COMPARATIVE CODE UNDERSTANDING OF INFORMATION MODELS (SPECIAL EDITION)

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

June 2003

By
Andrew Philip Broad
Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

This thesis concerns comparative code understanding, applied to information models (written in the EXPRESS modelling language) as a particular type of code. Information models are similar to database schemata (they contain entities, attributes, integrity constraints, etc.), but are intended for communicating the semantics of a domain, rather than the efficient physical representation of data.

A method is presented for comparing pairs of EXPRESS information models, with particular emphasis on the constraints specified in those models. This method is embodied in an experimental comparative code-understanding system (CCUS). Four aspects of the CCUS's behaviour constitute contributions to knowledge. **Firstly**, it infers correspondences between the two models (using name equivalence and semantic equivalence correspondence heuristics). **Secondly**, it assesses the semantic equivalence and relative semantic strength between the constraints in the models. **Thirdly**, it combines comparison with the extraction of higher-level knowledge about the constraints, in order to assess semantic equivalence at a higher level of abstraction when it fails to do so at the code level. **Fourthly**, this extraction is selective in that the CCUS only extracts those higher-level constraints (HLCs) that it needs, rather than extracting all the HLCs that it knows how to extract.

The CCUS is evaluated with respect to a specific application context: reasoning about potential violations of constraints when instances are mapped from one data repository to another, where the two repositories conform to different EXPRESS models. Comparative code understanding could also have applications in such areas as versioning, integration, and transformation.

Although the implemented CCUS is limited to EXPRESS information models, the four contributions listed above should generalise to all kinds of code (including program code in particular). By bringing these ideas together, this work aspires to establish comparative code understanding as a field in its own right.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

I would like to begin by thanking the three people without whom it would not have been possible for me to submit this thesis: my supervisor Nick Filer, and my parents.

I would also like to thank Hilary Kahn for being my advisor. I would like to thank David Brée and all those students in CS710 who made useful comments on an early draft of this thesis.

I am very grateful to the Department of Computer Science for accepting me as a student and providing facilities – both as an undergraduate and as a postgraduate. I have really enjoyed my time here.

Finally, I wish to acknowledge those whose brilliance inspires me in everything I do: my idol David Bowie; Radiohead; Björk; my tennis heroines Monica Seles, Daniela Hantuchová, Jelena Dokić, Iva Majoli, Karina Habšudová and Iroda Tulyaganova; everyone involved in the *Manic Miner/Jet Set Willy* community; and all those of you who know who you are.

About the Author

The Author graduated from the University of Manchester in July 1997, gaining the degree of Bachelor of Science in Computer Science with First Class Honours, and winning the Williams/Kilburn Medal for being the outstanding final year student.

He has been a research student in the Department of Computer Science at the University of Manchester since September 1997, gaining the degree of Master of Philosophy in 1999.

Glossary

abstract correspondence A code-level correspondence which can only be inferred by extracting and comparing HLCs (Section 6.3.3).

aggregation A set, bag, list or array of elements (Section 3.4).

antonyms Two words which are opposites, e.g. "male" and "female" (Section 4.2.4).

bout of extraction The CUS extracts all the HLCs it can from a given extraction prompt, which can be anything from a whole model to an individual expression (the CUS works inwards). The HLC CCUS, in its attempt to resolve a difference, calls the CUS to perform localised bouts of extraction from the vicinity of the difference in both models (Section 6.3.1).

case A prescription for solving a problem, consisting of a *problem part*, and a *solution part* which can be applied in the current situation if and only if the current situation matches the problem part (Appendix B). (Hyponyms: extraction case, navigation case, SE/S case, subexpression case, pretty-print case.)

CCUS *Comparative Code-Understanding System.* The CCUS compares two EXPRESS models (with particular emphasis on the constraints in those models) (Chapter 3).

code Anything written in a formal language, such as a program or an information model. The word "code" is used in this thesis as a count noun ("a code" can be read as "a piece of code, such as a program or an information model") as well as a mass noun.

code-fragment A piece of a code (such as an entity, a function or an expression) rather than the whole code.

code generation Generating code to implement a given specification – the inverse of *code understanding* (Section 2.1.1).

code level The internal (frame-based) representation of the code itself – not the actual source code, and not including any higher-level knowledge extracted from the code (Section 3.2).

code-level CCUS A version of the CCUS which compares at the code level only, without extracting HLCs (Chapter 3).

code understanding Extracting higher-level knowledge from a code – the inverse of *code generation* (Section 2.1).

comparative code generation Generating code to bridge semantic differences between two existing codes, which entails *comparative code understanding* (Section 1.3).

comparative code understanding Comparing two codes, inferring correspondences and assessing semantic equivalence, extracting higher-level knowledge in order to do so when it cannot be done at the code level (Section 1.2).

comparison (as opposed to matching) Comparing two codes, or data in general, which are 'equal in status' (Chapter 2).

completeness A correspondence heuristic (or a decision procedure in general) is *complete* if it always says that two objects correspond when they do (i.e. no false negatives) (Section 4.1.1).

computational mapping The computational (as opposed to *structural*) part of a mapping, which specifies how values are to be converted (Section 4.1.1).

constraint A condition in an information model, which must not be violated by instances of entities in that model. The constraint is either *satisfied* (if the condition evaluates to true or unknown) or *violated* (if the condition evaluates to false) (Section 1.1).

control-flow graph A graph of the control dependences between statements in a program (Section 2.1).

correspondence An assertion that an object in one code corresponds to an object in another code (Chapter 4). (Hyponyms: abstract correspondence, plausible correspondence.)

correspondence heuristic A rule of thumb for inferring correspondences (not guaranteed to be sound or complete) (Section 4.2). (Hyponyms: frame-class equivalence, name equivalence, semantic equivalence/strength.)

count noun A noun for objects that can be counted individually (e.g. "chairs", "objects", "codes"), as opposed to a mass noun.

cover An HLC covers (some of) the semantics of a code-fragment – typically the code-fragment from which it was extracted (Section 6.3.3).

current abstraction level The level at which the CCUS is currently operating: either the code level, or a level of HLCs.

CUS *Code-Understanding System.* The CUS extracts HLCs from a single EXPRESS model (Appendix B).

data-flow graph A graph of the data dependences between statements in a program and values (Section 2.1).

data repository A set of instances, conforming to a schema (Sense 1) such as an EXPRESS model.

defined type A named type which is defined by the user as opposed to built into EXPRESS (Section 1.2).

delocalised difference A difference which can only be resolved by comparing HLCs from non-corresponding bouts of extraction, and thus requires remembrance (Section 7.2).

disjointness Two semantically incomparable constraints are *disjoint* if it is impossible, in all cases, to satisfy both of them (Section 5.1.3).

domain rule An explicit constraint in an EXPRESS model, expressed as a logical expression which must evaluate to true or unknown for the constraint to be satisfied (Section 1.1).

entity A specification of the attributes and constraints of a class of instances. Entities are written in a schema (Sense 1) such as an EXPRESS model. Entity is to instance as frame-class is to frame.

equivalence Two objects are equivalent if they are completely interchangeable (see *semantic equivalence*).

EXPRESS model A model written in the EXPRESS information-modelling language. Essentially, an EXPRESS model consists of a set of entities (partitioned into a set of schemata (Sense 2)).

external instance-number The instance-number of a frame in an FIL file (Section 3.2.1).

extract an HLC To instantiate an HLC for a particular code-fragment (Section 6.3).

extraction case A case for extracting HLCs from an EXPRESS model. The problem part of the case is a template for a fragment of EXPRESS code, and the solution part specifies the HLC(s) to be extracted (Section 3.1).

extraction prompt A frame from the representation of a code-fragment, submitted to the CUS (possibly internally), from which it attempts to extract HLCs (Section 6.3.1).

extraneous HLCs HLCs extracted which are not needed to resolve a difference (Section 6.3.1).

FIL *Frame-Instance Language.* A language for the external representation of frames as instances of EXPRESS entities (Section 3.2.1).

FIL-C *Frame-Instance Language for Cases.* An extension of FIL for representing cases (Appendix B).

file integration Merging two concurrently-modified versions of a file (Section 2.2.1).

filler The value of a slot for a particular frame (Section 3.2.1).

frame A collection of slot-fillers (attributes and their values) to represent an object. Each frame is an instance of a frame-class (Section 3.2.1).

frame-class A specification for a particular class of frames, specifying the slots (and their filler-types) that are filled by frames of that frame-class. Each frame-class is represented by an entity in the frame-class model. Frame-class is to frame as entity is to instance (Section 3.2.1).

frame-class equivalence A necessary but not sufficient correspondence heuristic – it vetoes correspondence between frames of different frame-classes (that are not declared to be comparable in the frame-class-thesaurus) (Section 4.2.2).

frame-class model The set of frame-classes (written as an EXPRESS model) (Appendix D).

frame-class-thesaurus A thesaurus for the names of frame-classes; frames of two different frame-classes may be compared if and only if the names of those frame-classes are declared to be synonyms in the frame-class-thesaurus (Section 4.2.4).

graph A set of nodes with links between them.

HLC *Higher-Level Constraint.* A small chunk of knowledge extracted from an EXPRESS model, which expresses some constraint semantics at a higher level of abstraction (Chapter 6).

HLC CCUS A version of the CCUS which, upon encountering a difference that it cannot resolve at the current abstraction level, extracts HLCs and compares those (Chapter 6).

homonyms Identical words with different meanings, or syntactically identical expressions that are not semantically equivalent (Section 4.2.4).

hypernym A word which is a superclass of another word, e.g. "animal" is a hypernym of "cat" (cf. *hyponym*).

hyponym A word which is a subclass of another word, e.g. "cat" is a hyponym of "animal" (Section 4.2.4).

information model A formal description of the entities, relationships and constraints in a domain – like a database schema, but intended for communicating the semantics of a domain rather than for efficient implementation. This thesis concerns EXPRESS information models.

instance A particular object conforming to the entity of which it is an instance. A data repository contains a set of instances. Instance is to entity as frame is to frame-class.

instance-number A frame is uniquely identified by its frame-class and instance-number, e.g. `person#1` (Section 3.2.1). (Hyponyms: internal instance-number, external instance-number.)

inter-entity constraint An inter-instance constraint which must be satisfied by sets of instances of different entities (Section 8.3.1.2).

inter-instance constraint A constraint which must be satisfied by sets of instances collectively, rather than by each instance individually (Section 8.3.1.2).

internal instance-number The instance-number of a frame in memory (Section 3.2.1).

Jex The system used to parse EXPRESS models, which provides access to the resulting internal representation (Section 3.2).

locality A kind of remembrance heuristic which assumes that only HLCs from localised parts of the models can correspond, and thus discards HLCs when the HLC CCUS has finished comparing those parts (Section 7.2.5).

mapping (Sense 1) transferring instances from one data repository to another (Section 1.1); (Sense 2) a specification of how to do so, consisting of a *structural mapping* and a *computational mapping* (Section 4.1.1).

mass noun A noun for objects that are not counted individually (e.g. "butter", "water", "knowledge"), as opposed to a count noun.

matching (as opposed to comparison) Comparing a 'rule' with a situation to see if it is applicable, as opposed to comparing two situations (Chapter 2).

memorise an HLC To store an HLC in the remembrance structure for the model from which it was extracted (Section 7.2).

meronym A word which has a part-of relationship with another word, e.g. "finger" is a meronym of "hand" (Section 4.2.4).

monogamy A kind of remembrance heuristic which assumes only one-to-one correspondences between HLCs, and thus discards HLCs when they are inferred to correspond to other HLCs (Section 7.2.3).

named object An object that has a name; in the CCUS, named objects are represented as frames with the distinguished slot `name` (Section 4.2.1).

name equivalence A class of correspondence heuristics which sanction or veto correspondence between named objects based on their names (Section 4.2). (Hyponyms: name identicality, name equivalence with a thesaurus.)

name equivalence with a thesaurus A correspondence heuristic which sanctions correspondence between two named objects if their names are identical or declared in a name-thesaurus to be synonyms (Section 4.2.4).

name identicality A correspondence heuristic which sanctions correspondence between two named objects if and only if their names are identical (Section 4.2.3).

name-thesaurus A thesaurus for names; named objects may be inferred to correspond if their names are declared to be synonyms in the name-thesaurus (Section 4.2.4).

navigation case A case for navigating an EXPRESS model for the extraction of HLCs. The problem part of the case is a template for a fragment of EXPRESS code, and the solution part specifies the frames to be submitted to the CUS as extraction prompts (Section 6.3.1).

necessary slot A slot named in the solution part of an SE/S case, which specifies that if a given slot has different fillers for two frames, then those two frames are semantically incomparable and fail to correspond (Section 5.3).

overlap Two semantically incomparable constraints are *overlapping* if it is possible, in some cases, to satisfy both of them (Section 5.1.3).

plausible correspondence A correspondence between two frames which have to be compared *before* the CCUS can decide whether they correspond or not. A plausible correspondence is retracted if the comparison fails (Section 4.2.5).

pretty-print case A case for pretty-printing a code-fragment (Appendix E, Section E.5).

program dependence graph (PDG) A combined control- and data-flow graph (Section 2.1).

proper subset/superset A proper subset/superset is one which cannot be equal to itself, i.e. it must have strictly fewer/more elements.

reengineering Modifying a code through a reverse-engineering step to understand the code, followed by a forward-engineering step to implement the necessary changes (Section 2.1).

remembrance Remembering HLCs from earlier bouts of extraction so that they can be compared with HLCs from later bouts of extraction. Remembered HLCs are stored in a remembrance structure (Section 7.2).

remembrance heuristic A policy for which HLCs to remember and which not to remember (Section 7.2). (Hyponyms: locality, monogamy, unselective remembrance.)

resolve a difference To infer that two syntactically different objects (that fill the same slot) correspond. Some differences can be resolved at the code level, others can be resolved by extracting and comparing HLCs, and then inferring abstract correspondences (Section 4.2.1).

reuse reengineering Reengineering code to make it reusable (Section 2.1).

revision A version of a code produced by modifying it, as opposed to transforming it, or as opposed to different models of the same thing (Section 1.4.2).

sanction a correspondence To allow a correspondence to be inferred (Section 4.2.1).

satisfied A constraint is *satisfied* if the condition evaluates to true or unknown, otherwise it is violated (Section 1.1).

schema (Sense 1) A specification of the format (structure and constraints) of a data repository (EXPRESS is one language in which it could be written); (Sense 2) one of a set of schemata into which an EXPRESS model is partitioned.

schema integration Merging two or more local schemata of a federated database system to form a single global schema (Section 2.2.4).

schema-to-schema mapping Translating instances from (the schema of) one data repository to another (Section 1.1).

selective comparison Comparing HLCs selectively (Section 7.2).

selective extraction Extracting knowledge selectively from code (Section 7.1).

semantic equivalence Two constraints are *semantically equivalent* if whenever either of them is satisfied, the other is satisfied. To generalise, two expressions are *semantically equivalent* if they always evaluate to the same value as each other (Section 5.1.1).

semantic equivalence/strength A correspondence heuristic which infers correspondence between two expressions if they are semantically equivalent, or if one is semantically stronger than the other (Section 4.2.5, Chapter 5).

semantic heterogeneity Differences in data-formats that have to be overcome when translating data from one format to the other, including differences in structure and differences in constraint semantics (Section 1.1).

semantic incomparability Two constraints are *semantically incomparable* if they are neither equivalent, nor is one stronger than the other (Section 5.1.3). (Hyponyms: overlap, disjointness.)

semantic strength One constraint is *semantically stronger* (more constrained) than another constraint if whenever the first constraint is satisfied, then the second constraint is satisfied, but it is possible to satisfy the second and violate the first (Section 5.1.2).

semantic-strength slot A slot named in the solution part of an SE/S case, which specifies that a given slot affects the relative semantic strength of two frames, and does so either positively (the stronger frame has a greater value for this slot) or negatively (Section 5.3).

SE/S case *Semantic-Equivalence/Strength case.* A case for assessing the semantic equivalence or semantic strength between two frames of a particular frame-class. The problem part of the case specifies a frame-network which both of the frames being compared must match, and the solution part specifies a set of necessary slots and a set of semantic-strength slots (Section 5.3).

slice A subset of the statements of a program that (the subset) fulfils a given slicing criterion (Section 2.1).

slot An attribute (field) of a frame, which is filled with a particular value for each frame of the frame-class that has that slot. Each slot has a name and a filler-type, as specified in the frame-class (Section 3.2.1).

soundness A correspondence heuristic (or a decision procedure in general) is *sound* if it never says that two objects correspond when they do not (i.e. no false positives) (Section 4.1.1).

structural mapping The structural (as opposed to *computational*) part of a mapping, which specifies the correspondences between the source schema and the target schema (Section 4.1.1).

subexpression case A case for aligning subexpressions with top-level expressions for comparison, enabling the CCUS to compare such pairs of expressions as `age >= 0` and `(age >= 0) AND (age < 130)`. The problem part of the case specifies the frame-class of the top-level expression, and the solution part specifies the subexpressions with which to compare the other expression (Section 4.3).

synonyms Words that mean the same thing, or expressions that are semantically equivalent (Section 4.2.4).

synset A set of synonyms. A thesaurus consists of a set of synsets (Section 4.2.4).

thesaurus A data-structure that declares sets of names to be synonymous, and may also declare other relationships between them (Section 4.2.4). (Hyponyms: name-thesaurus, frame-class-thesaurus.)

transformation A version of a code produced by transforming it (Section 1.4.2).

unnamed object An object that does not have a name; in the CCUS, unnamed objects are represented as frames which do not have the distinguished slot `name` (Section 4.2.1).

unnecessary slot A slot which is not specified in the solution part of a given SE/S case. If an unnecessary slot has different fillers for two frames, this does not fail the correspondence between those frames (Section 5.3).

unselective remembrance A kind of remembrance heuristic which remembers all HLCs extracted from a model (Section 7.2.2).

version A code related to another code, either by *revision*, *transformation*, or different models of the same thing (Section 1.4.2).

veto a correspondence To forbid a correspondence to be inferred (Section 4.2.1).

vicinity of a difference Given two frames with a slot difference, the HLC CCUS has to decide which frames to submit to the CUS as extraction prompts so as to extract the HLCs needed to resolve the difference, but to minimise the extraction of extraneous HLCs (Section 6.3.1).

violated A constraint is *violated* if the condition evaluates to false, otherwise it is satisfied (Section 1.1).

WordNet The system used to implement the thesauri (Section 4.2.4).

Chapter 1

Introduction

Comparative code understanding is a cross-fertilisation of code comparison and code understanding (where "code" means code written in *any* formal language). A comparative code-understanding system compares two codes, inferring correspondences, finding differences, and assessing semantic equivalence. When semantic equivalence cannot be assessed at the code level, it can often be assessed at a higher level of abstraction, which is where code-understanding techniques come in (their purpose here is *not* to extract a complete understanding of the code). Comparative code understanding could have applications in such areas as versioning, integration, and transformation.

This thesis explores comparative code understanding, using information models written in EXPRESS as a test-bed (information models are like database schemata except that they are concerned with expressing the semantics of a domain, rather than with the efficient physical representation of data). Schema-to-schema mapping is the specific application context that has motivated the development of the Comparative Code-Understanding System, an experimental prototype for comparing, and extracting higher-level knowledge from, EXPRESS models.

Comparative code understanding is the primary contribution of this thesis, which has four subcontributions:

- inferring correspondences between the components of two codes;
- assessing the semantic equivalence, and the relative semantic strength, of the constraints in two codes (this is used as a correspondence heuristic);
- combining comparison with the extraction of higher-level knowledge from

codes, in order to assess semantic equivalence at a higher level of abstraction when it cannot be assessed at the code level;

- selective extraction and selective comparison of higher-level knowledge from the two codes (as opposed to extracting all the knowledge that can be extracted, and comparing every piece extracted from one code with every piece extracted from the other code).

1.1 Schema-to-Schema Mapping

In this world, data are constantly being moved around. Organisations share data: for example, student data are exchanged between departments of a university and the student records office; data are exchanged between tennis tournaments and the ATP/WTA professional tennis associations for the purposes of players entering tournaments, putting information on tournament websites and so on; when two companies amalgamate, they merge their data, which are in different formats; police departments in different countries exchange information about known criminals; millions of emails are sent around the world; the list seems endless.

Whenever data are transferred, the sender and the receiver are expected to be compatible such that the receiver will interpret the data in the same way as the sender. Assumptions are made about the way the data are supposed to be interpreted, for example 12/2/2001 is interpreted as a date, namely the Twelfth of February AD 2001 by Europeans, whereas an American might interpret it as December the Second. If an email from abroad asks whether a 1.024 Mb file can be accepted, it might be assumed that it is slightly larger than a megabyte, but many European languages use ‘,’ as a decimal point and ‘.’ to separate large numbers, so it might be a one-gigabyte (1024 Mb) file that is going to exceed the filestore quota. If a text file written in Slovak is downloaded, the accents may come out wrong because the same ASCII code is used by software in different countries to represent different characters. If it is attempted to ‘play’ the data on a CD-ROM using a conventional CD-player, they will sound like random noise.

This *semantic heterogeneity* problem occurs frequently in computing. Machine-code instructions which move 16-bit words around a byte-addressable memory might store the high byte followed by the low byte, so a routine that interprets them the other way round is asking for trouble. A routine that expects two’s

```
SCHEMA person1;
  ENTITY person;
    name : STRING;
    nationality : OPTIONAL STRING;
    age : INTEGER;
  END_ENTITY;
END_SCHEMA;
```

```
SCHEMA person2;
  ENTITY person;
    name : STRING;
    nationality : STRING;
    age : INTEGER;
  WHERE
    age_must_be_sensible:
      (age >= 0) AND (age < 130);
  END_ENTITY;
END_SCHEMA;
```

Figure 1.1: Two simple EXPRESS models of people.

complement binary numbers cannot be called with sign-and-magnitude binary numbers and be expected to give the correct result.

It is often necessary to interchange data between applications. For example, if a Computer-Aided Design (CAD) tool is to pass its designs to an independently-developed Computer-Aided Manufacturing (CAM) tool, the data from the CAD tool will almost certainly have to be translated to the format of the CAM tool, unless the tools were developed using a standard data-format to promote interoperability.

Data often have to be translated from one format to another, overcoming the semantic heterogeneity [DKM+93] between the two formats. The source format and the target format can be described by schemata. A *schema* describes *entities* (e.g. `person`) and their *attributes* (e.g. `name`, `nationality`, `age`). A schema also specifies *integrity constraints*: any *instance* of an entity (i.e. a specific person) must satisfy the constraints specified for that entity (e.g. a person's age must be between 0 and 130 years).

Consider the first schema in Figure 1.1, for example. It is an information model of people written in the EXPRESS information-modelling language [ISO94], an international standard (ISO 10303-11).

An *information model* specifies the entities, relationships and constraints in a domain (Section 1.1 of [Kas96]). So information models are similar to database schemata, but the purpose of an information model is to communicate the semantics of a domain, rather than the efficient physical representation of data.

The first schema[1] in Figure 1.1 defines the *entity* `person`, which has the

[1]Technically, an EXPRESS model consists of one or more schemata; however this thesis uses the term *schema* primarily to mean a specification of a data-format, so sometimes an EXPRESS model is referred to as a schema (even though it could contain more than one EXPRESS schema).

explicit attributes `name`, `nationality` and `age`. The attributes `name` and `age` are *mandatory* – an instance of `person` must have values for these attributes, whereas `nationality` is an *optional attribute*, for which the value can be *indeterminate* (null). An instance's values of `name` and `nationality` must (if present) be of type `STRING`; its value of `age` must be of type `INTEGER`.

Consider the second schema in Figure 1.1. Assuming that the same names represent the same concepts, it has two differences from the first schema:

- `nationality` is now mandatory;
- `person` now has a *domain rule* whose *label* is `age_must_be_sensible`[2] and whose *top-level expression* is `(age >= 0) AND (age < 130)`, which must be true for each instance of `person`.

The mandatory attributes, the types of the attributes, and the domain rule, are all examples of constraints. A *constraint* is a condition in a schema that must not be violated by instances of entities in the schema. For a given instance, each constraint is either:

- *satisfied* (if the condition is true – or even unknown, which is possible in EXPRESS), or
- *violated* (if the condition is false).

A *valid* instance is one which satisfies all of the constraints specified for the entity of which it is an instance.

The reader who is unfamiliar with EXPRESS should refer to Appendices A and F for more examples, or to Section 2.2 of [Kas96], or [SW94], for an overview of EXPRESS.

Schema-to-schema mapping is the task of translating instances from one schema to another (copying rather than moving). An oft-ignored problem in schema-to-schema mapping is that translating an instance may violate constraints in the target schema:

- Consider mapping from the first schema to the second schema an instance of `person` with 'indeterminate' as the value of `nationality`. This instance

[2] A domain rule need not have a label, and although it is good modelling practice for domain rules to have labels, subsequent examples in this thesis will omit them, and will often not distinguish between the top-level expression of a domain rule and the domain rule itself.

violates the constraint in the second schema that `nationality` is a mandatory attribute.

- If an instance of `person` in the first schema has a value of `age` less than 0, or greater than or equal to 130, then mapping this instance to the second schema will violate the domain rule.

Note that mapping valid instances from the second schema to the first schema could never violate the constraints on the first schema, because every constraint in the first schema is in the second schema. It could be said that the second schema is *semantically stronger* (more constrained) than the first.

1.2 Comparative Code Understanding

Comparing the semantics of the constraints in EXPRESS models is an application of *comparative code understanding*. This entails comparing the two models, inferring correspondences and reporting differences, assessing the semantic equivalence and relative semantic strength of the constraints in the models.

Inferring correspondences is trivial for the two models in Figure 1.1: the entities `person` correspond, the attributes `name` correspond, and so on; the domain rule in the second model does not correspond to anything in the first model.

In the implemented Comparative Code-Understanding System for EXPRESS (CCUS), these entity-correspondences and attribute-correspondences are inferred because they have the same names. A thesaurus can (at the option of the user) be used to infer correspondence between appropriate named objects which have different names (*synonyms*), and could be used to veto correspondence between objects which have the same name but represent different real-world concepts (*homonyms*).

The *correspondence problem* is to infer correspondence between objects that represent the same real-world concept – attempting to deduce this from the codes themselves, possibly with the help of additional knowledge such as thesauri.

Semantic equivalence (as opposed to name equivalence) is used to infer correspondence between domain rules (and between expressions in general). For example, `(age >= 0) AND (age < 130)` can be written in different ways which are *semantically equivalent*, such as:

- `(age < 130) AND (age >= 0)`;

```
SCHEMA gender_by_enumeration;
  ENTITY person;
    name : STRING;
    gender : gender_type;
  END_ENTITY;

  TYPE gender_type = ENUMERATION OF (male, female);
  END_TYPE;
END_SCHEMA;
```

```
SCHEMA gender_by_subtypes;
  ENTITY person ABSTRACT SUPERTYPE OF (ONEOF (man, woman));
    name : STRING;
  END_ENTITY;

  ENTITY man SUBTYPE OF (person);
  END_ENTITY;

  ENTITY woman SUBTYPE OF (person);
  END_ENTITY;
END_SCHEMA;
```

Figure 1.2: Two EXPRESS models of people, representing gender in different ways.

- `(0 <= age) AND (130 > age)`;
- {`0 <= age < 130`};
- {`-1 < age <= 129`} (for integers only);
- {`0 <= (age*1 + 0) < 130`}...

It is also appropriate to infer correspondence between two expressions such that one is *semantically stronger* (more constrained) than the other: for example, `age >= 21` is stronger than `age >= 18`. This means that an instance which satisfies `age >= 21` can always be mapped to a schema with the constraint `age >= 18` without violating *that* constraint, but 18-to-20-year-olds could not validly be mapped from the second schema to the first because they would violate `age >= 21`. So the CCUS infers correspondence between `age >= 21` and `age >= 18` because one is stronger than the other (this thesis is based on the principle that comparison is bidirectional: it should not matter which way round two codes are compared).

Note that there are many pairs of constraints such that neither constraint is equivalent to, nor stronger than, the other constraint. Such constraints are called *semantically incomparable*, e.g. `age >= 0` is semantically incomparable to `age < 130` because neither implies the other. These are the constraints between which no correspondences are inferred.

The most important problem addressed in this thesis is that constraints can be written in different but semantically equivalent ways, and it is not always possible to see their equivalence by comparing the codes themselves. Consider the two models in Figure 1.2 for example. Both models express the constraint that a person must have exactly one gender, but in radically different ways:

- In the first model, `person` has an attribute `gender` of the *defined type* `gender_type`. That this attribute is mandatory means that a person must have at least one gender. That `gender_type` is an enumeration type means that a person must have at most one gender, namely `male` or `female`.

- In the second model, `person` has two *subtypes*: `man` and `woman`. That `person` is declared to be an *abstract supertype* means that a person must have at least one gender, since an instance of `person` cannot exist without being bound to an instance of `man` or an instance of `woman` (to form what is known as a *complex instance*). The `ONEOF` *supertype expression* means that a person must have at most one gender (note that the supertype expression is optional in EXPRESS – without it, a person could be both a man and a woman).

These two models are semantically equivalent in the sense that mapping valid instances from one to the other cannot violate any constraints. To see that they are semantically equivalent means to see, at an abstract level, that both models have the constraint that a person must have exactly one of two genders. This entails:

- inferring that an instance of `person` in the first model maps to a complex instance of `person` and `man` when the value of `gender` is `male`, and to a complex instance of `person` and `woman` when the value of `gender` is `female` (the associations between `male` and `man` and between `female` and `woman` are given in a thesaurus);

- extracting from the first model a *higher-level constraint (HLC)* which says that `person` must have exactly one of the properties {`male`, `female`}, and from the second model an HLC which says that `person` must have exactly one of the properties {`man`, `woman`}. Since the correspondences between `male` and `man` and between `female` and `woman` are sanctioned by a thesaurus, the two HLCs are assessed to be semantically equivalent and inferred to correspond.

Semantic strength can be assessed at the HLC level, too:

- If the `gender` attribute in the first model were optional, or if the `person` supertype in the second model were not abstract, then a person would only be constrained to have *at most one* gender, which is a semantically weaker constraint than having to have *exactly one* gender.

- If `gender_type` were defined as `ENUMERATION OF (male, female, third_gender)`, then a person must have one of three genders, which is a semantically weaker constraint than having to have one of two genders.

This combination of comparison with the extraction of higher-level knowledge is what makes it *comparative code understanding* as opposed to mere comparison – *code understanding* entails extracting higher-level knowledge from code. Note that the 'understanding' in comparative code understanding is not so much extracting a model of the code (as in conventional program-understanding systems – Section 2.1) as extracting some common semantic abstractions to facilitate comparing semantics at a higher level than the code.

Comparative code understanding is not limited to EXPRESS models, of course, but could be applied to codes written in any formal language (or languages, in the case of inter-language comparative code understanding). For example:

- The semantic equivalence of two programs that compute the same function using different algorithms (e.g. bubblesort and shellsort) could be assessed using comparative code understanding.

- Two database queries can be assessed to see whether they would return the same set of records for all possible databases (semantic equivalence), or whether one would always be a subset of the other (the 'superset' query is semantically weaker than (*contains*) the 'subset' query).

- A *model-oriented specification* of a stack (which specifies the preconditions and the postconditions of the 'push' and 'pop' operations) could be assessed to be equivalent to an *algebraic specification* (where the same constraints are expressed in the axioms which relate the operations). These specifications are shown in Figure 1.3. There is a semantic difference, in fact: the algebraic specification allows the operations `pop` and `top` to be applied to an empty stack, but the model-oriented specification does not.

- Two grammars could be assessed to see whether they describe the same language.

<pre>stack = list of int
invariant: length(stack) >= 0
initially: stack = []
operations:
  push (s : stack, x : int) -> void
    pre: true
    post: s' = x::'s
  pop (s : stack) -> void
    pre: length('s) > 0
    post: 's = x::s'
  top (s : stack) -> h : int
    pre: length(s) > 0
    post: s = h::t
  is_empty (s : stack) -> e : boolean
    pre: true
    post: e = (s = [])</pre>
	<pre>stack
	operations:
	  new () -> stack
	  push (stack, int) -> stack
	  pop (stack) -> stack
	  top (stack) -> int
	  is_empty (stack) -> boolean
	axioms:
	  is_empty(new) = true
	  is_empty(push(s,x)) = false
	  pop(new) = new
	  pop(push(s,x)) = s
	  top(new) = 0
	  top(push(s,x)) = x</pre>

Figure 1.3: Model-oriented and algebraic specifications (adapted from Figures 11.6 and 11.5 in [vV93]).

Comparative code understanding adds to code comparison the ability to assess the extent to which two codes are semantically equivalent, at an abstract level when it is not possible at the code level. This would be useful for many applications, as discussed in Section 1.4; the next section focuses on one such application.

1.3 Transferring Constraints for Schema-to-Schema Mapping

Transferring constraints for schema-to-schema mapping is the application context with respect to which the work presented in this thesis should be judged. It has not been implemented, but this task (first introduced in Section 1.3.2 of [Bro99]) is what has motivated the development of the CCUS from the start.

The goal of schema-to-schema mapping is: given two schemata – the *source schema* and the *target schema* – to generate automatically a program to map instances from the source schema to the target schema, as opposed to writing the mapping program by hand.[3] Often, a specification of the mapping has to be given as an additional input, although comparing schemata and inferring correspondences can obviate the need for this (this work assumes that instances would be mapped according to the correspondences inferred by the CCUS).

[3]This is an example of *code generation* – the inverse of code understanding.

As discussed in Section 1.2, semantic equivalence and semantic strength are defined in terms of the possible violation of constraints in the target schema when valid instances are mapped from the source schema. Any constraints in the target schema that do not hold in the source schema could be violated by mapping valid instances from a source repository (that conforms to the source schema[4]) which are invalid when mapped to the target schema. For example, mapping a stateless 131-year-old from `person1` to `person2` in Figure 1.1 would violate two constraints in `person2` even though that person is valid with respect to `person1`.

It would be useful to deduce which constraints could be violated when mapping from a given source schema to a given target schema. Existing schema-to-schema mapping methods (Section 2.4.2) have tended to ignore the problem that constraints could be violated, concentrating only on performing the structural mapping.

A naïve solution to this problem would be to map an instance from the source repository to the target schema, and then to check whether the mapped instance is valid with respect to the target schema before accepting it into the target repository. This has the following drawbacks:

- The mapping has to be performed before the target schema's constraints can be checked for the mapped instance, which would be particularly disadvantageous for expensive mappings, many mappings, many instances which are invalid with respect to the target schema, and mapping across a network.

- All of the constraints for the appropriate entity in the target schema have to be checked, even if they already hold on the source schema (note that checking a target-schema constraint is redundant if that constraint has a semantically equivalent or stronger constraint in the source schema).

A solution which overcomes these drawbacks would be to check that each instance in the source repository is valid with respect to the target schema *before* mapping it to the target repository. This entails transferring constraints from the target schema to the source schema (Figure 1.4) so that they can be checked prior to mapping each instance – and only constraints that are not implied by the constraints in the source schema need to be transferred.[5]

[4] A data repository *conforms* to a schema if it contains only valid instances of entities in that schema.

[5] Assuming that all instances in the source repository are valid with respect to the source schema.

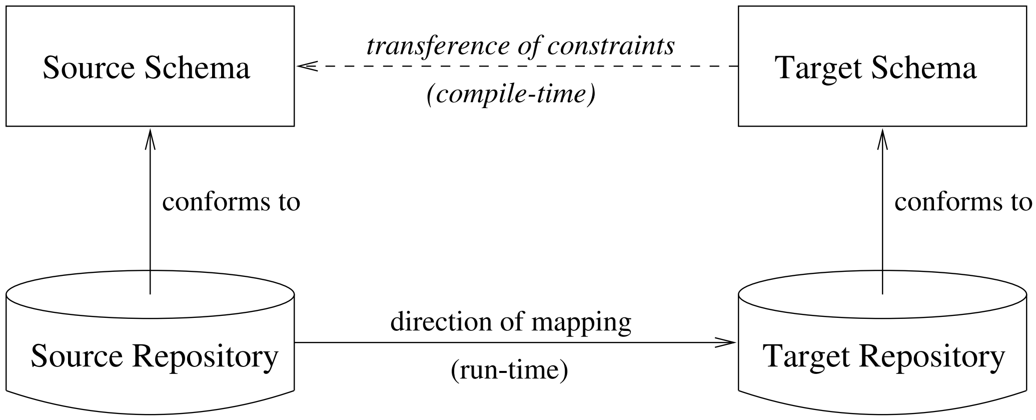


Figure 1.4: Transference of constraints for schema-to-schema mapping (adapted from Figure 1-6 in [Bro99]).

This is where comparative code understanding comes in: the CCUS infers correspondences between constraints in the two schemata, and assesses whether corresponding constraints are semantically equivalent or one is stronger than the other. Using the CCUS as a building block, a constraint-transference system would transfer, from the target schema to the source schema, just those target-schema constraints which do not have corresponding constraints in the source schema, or which have corresponding but semantically weaker constraints in the source schema.

To transfer a constraint from the target schema to the source schema really means to plant code in the generated mapping program to check that constraint on each source instance before mapping it to the target schema (if any constraints are violated, the generated mapping program should raise an exception instead of performing the mapping). However, constraint transference can be thought of as adding constraints from the target schema to the source schema, adapting the constraints if necessary according to the structural differences from the target schema to the source schema:

- When mapping from `person1` to `person2` in Figure 1.1, the domain rule `(age >= 0) AND (age < 130)` is transferred to `person1` as it is, since there is no structural difference between the schemata (i.e. code to check this constraint would be planted in the mapping program); the optional attribute `nationality` becomes mandatory (i.e. a non-null check would be planted). No constraint transference is needed when mapping from `person2` to `person1`.

- When mapping from `temperature_in_fahrenheit` to `temperature_in_celsius` in Figure 1.5, the domain rule {`0 < temperature < 100`} is transferred to `temperature_in_fahrenheit`. That the units of temperature are different constitutes a *computational mapping* between the two schemata, which has to be taken into account when transferring constraints: thus the domain rule is adapted to {`32 < temperature < 212`} (using the conversion formula $F = 32 + \frac{9}{5}C$).

- If corresponding named objects have different names in the two schemata, then any references to those objects also have to be renamed (e.g. `(years_alive >= 0) AND (years_alive < 130)`).

- No constraint transference is needed when mapping between `gender_by_enumeration` and `gender_by_subtypes` in Figure 1.2 because the constraints are semantically equivalent. However:

  - if `gender_type` were optional then it would have to become mandatory when mapping from `gender_by_enumeration` to `gender_by_subtypes` (i.e. a non-null check would be planted);

  - if `person` in `gender_by_subtypes` were non-abstract then it would have to become abstract when mapping from `gender_by_subtypes` to `gender_by_enumeration` (i.e. a check that each instance of `person` in `gender_by_subtypes` is bound to an instance of `man` or of `woman` would be planted).

  That these checks do not have to be performed anyway (with Figure 1.2 as it stands) is the difference that extracting HLCs makes – using code-level comparison alone, the CCUS would fail to resolve the syntactic differences (i.e. to see that the models are semantically equivalent) and hence assume that the constraints were semantically incomparable, so the constraint-transference system would have no option but to transfer these constraints.

Transferring constraints for schema-to-schema mapping is an example of *comparative code generation*: generating code to bridge the semantic differences between two codes. Comparative code generation clearly entails comparative code understanding – to find the differences to be bridged.

```
SCHEMA temperature_in_fahrenheit;
  TYPE fahrenheit = REAL;
  END_TYPE;

  ENTITY water;
    temperature : fahrenheit;
  END_ENTITY;
END_SCHEMA;
```

```
SCHEMA temperature_in_celsius;
  TYPE celsius = REAL;
  END_TYPE;

  ENTITY water;
    temperature : celsius;
  WHERE
    {0 < temperature < 100};
  END_ENTITY;
END_SCHEMA;
```

Figure 1.5: Two EXPRESS models to illustrate computational mapping.

1.4 Motivations

This section discusses the motivations for code understanding (Section 1.4.1) and for comparative code understanding (Section 1.4.2).

1.4.1 Motivations for Code Understanding

Code understanding, which entails extracting knowledge from code, can be used to make code more understandable to humans or to computers. A literature review of code understanding is in Section 2.1.

The usual motivation for code understanding is *reengineering*: understanding a code at a higher level of abstraction (*reverse engineering*) so that it can be reimplemented (*forward engineering*). Some tasks which entail reengineering are:

- propagating changes in the requirements for a software system to the specification, to the design and to the implementation, without having to reimplement the system from scratch;
- rewriting program code to be more efficient, particularly high-level optimisations such as using a different algorithm;
- translating code to another language or (in the case of executable code) to run on another hardware platform – particularly reengineering legacy systems to modern languages or to run on modern platforms;
- *reuse reengineering*: reengineering code to make it reusable, e.g. reengineering an unstructured legacy system into a set of reusable components.

However, the motivation for code understanding in this thesis is to support comparative code understanding – combining code comparison with the extraction of higher-level knowledge adds the ability to infer *abstract correspondences* that cannot be inferred by comparing at the code level alone, and thus to resolve more of those syntactic differences that actually constitute semantic equivalences.

Code understanding of information models differs from program code understanding in that it is concerned with information semantics rather than behaviour. In particular, it is concerned with the semantics of the constraints in the models. Some roles of constraints are:

- to capture the semantics of information models by expressing restrictions – constraints express the necessary conditions to be an instance of an entity, which (together with sufficient conditions) are an integral part of the intensional definition of a concept (cf. description logics [Hor97]);
- to enforce the semantic integrity of conformant databases/repositories (by checking constraints);
- constraints can also apply to database updates as well as database states (e.g. "score can only increase") – these *transitional constraints* are usually enforced by procedural *triggers* rather than declarative constraints [CPM96];
- constraints in information models and database schemata are analogous to *invariants* (conditions that must be true before and after operations) in model-oriented specifications (where operations are specified in terms of their preconditions and postconditions), and to conditions in program code (although these tend to control behaviour rather than enforce integrity).

Other possible applications of constraint understanding are:

- to generate code to check constraints in an implementation derived from an information model [Teg97];
- to check whether the constraints are sensible, e.g.
  - can be satisfied both individually and collectively, unlike `age < 16` and `age >= 65`;
  - are not *redundant*, i.e. implied by another constraint in the same model.

1.4.2 Motivations for Comparative Code Understanding

Comparative code understanding could support the following tasks, which are discussed in detail with literature review in Chapter 2:

- *schema-to-schema mapping*, as already discussed;
- *program equivalence* (*program differencing*): this tends to focus on mathematical (not necessarily arithmetic) expressions (Section 5.2.3.4);
- *semantics-preserving transformations*, which will be discussed in Section 1.4.2.1;
- *schema integration*: a *federated database system* consists of several local schemata, and a global schema which can be derived by integrating the local schemata, which includes *schema comparison* as a subprocess (Section 2.2.4);
- *mediation*: a *mediator* provides an integrated view of several heterogeneous databases; mediation entails selecting appropriate information-sources to answer a given query, rewriting the query in terms of the source schemata so that it can be executed, and integrating the results from the sources to present to the user (Section 2.4.4);
- *data warehousing*: a *data warehouse* stores data from several sources, often in summarised form, which raises the problem of keeping the warehouse up-to-date with the data-sources (Section 2.3.2);
- *versioning*: three kinds of versions can be distinguished, in increasing order of the difficulty of comparing versions:
  - *revisions*: modifying a code produces a new *revision* of that code – revisions are the easiest codes to compare (unless the modifications are radical);
  - *transformations*: applying a transformation to a code and comparing the transformed version with the original – this is more difficult, particularly if the comparison method is not informed of which transformation was applied;

  – different models of the same domain (e.g. Gerber [STEa] has a conceptual model, an implementation model and a syntactic model; EDIF [Lau96] and VHDL [STEc] are separate languages for describing electronic designs, but their information models have much semantic overlap) – these are the most difficult codes to compare, as they differ radically;

- *program integration*: integrating concurrent modifications to a program, which can interfere if semantics are not handled properly (Section 2.2.3).

1.4.2.1 Transformations

The purpose of semantics-preserving transformations is typically to improve efficiency, as in compiler optimisations (Chapter 10 of [ASU86]). Information models can be transformed, too – from rich conceptual models to data models that support efficient processing. Examples of information-model transformations are: [KFWW01]

- *flatten-up*: transform a subtype hierarchy (the left model in Figure 1.6) by removing the subtypes, transferring their attributes and constraints up into the supertype, and adding attributes to distinguish between subtypes (the upper-right model in Figure 1.6) – the domain rule `(gender = woman) XOR NOT EXISTS(pregnant)` means that if an instance of `person` is a woman then `pregnant` must have a value (i.e. `EXISTS(pregnant)` is true), otherwise `pregnant` must be indeterminate (`EXISTS(pregnant)` is false);

- *flatten-down*: transform a subtype hierarchy by removing the supertype, transferring its attributes and constraints down into the subtypes (the lower-right model in Figure 1.6);

- adding an *object identifier (OID)* to entities so that they can be populated in an object-oriented database.

The application of comparative code understanding to transformation could be to verify that (or assess the extent to which) a transformation of a code preserves the semantics of the original code. If it does not, the transformed code could be modified to respect as much of the semantics of the original as possible, according to the semantic differences found – another example of comparative

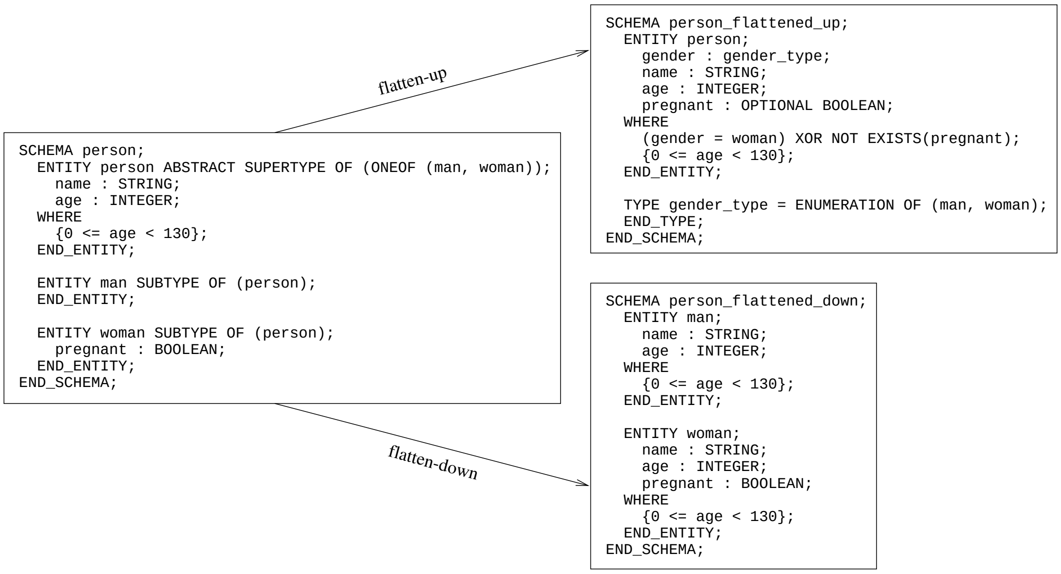


Figure 1.6: An EXPRESS model, flattened up and flattened down.

code generation. It really is not possible to compare non-trivial transformations without comparing the codes at a higher level of abstraction, thus it is again necessary to combine comparison with extraction.

Comparative code understanding could also be used to infer which transformation has been applied.

1.5 Contributions

The primary contribution of this thesis is *comparative code understanding*, applied to EXPRESS information models in particular.

Comparative code understanding has its roots in comparison methods and code understanding, and this work is more of a contribution to the former, although comparative code understanding deserves to be a field in its own right.

The application of code understanding to information models as opposed to program code (as in most of the existing code-understanding systems) means that it is concerned with information semantics rather than with behaviour; in particular, it is concerned with the semantics of constraints. The constraints in information models are pure and declarative (unlike program code), which makes information models a test-bed that facilitates trying out the ideas of comparative code understanding without such complications as state and flow of control.

The CCUS has been implemented as an experimental prototype for comparative code understanding of EXPRESS models. The CCUS compares two EXPRESS models, inferring correspondences and reporting differences.

The *extrinsic evaluation criterion* of the CCUS is how well it could support transferring constraints for schema-to-schema mapping (Section 1.3) – the application context which has motivated the development of the CCUS.

The *intrinsic evaluation criteria* of the CCUS are the soundness and completeness – and to a lesser extent the efficiency – of inferring correspondences and of assessing semantic equivalence/strength. The CCUS errs on the side of soundness; the big issue is completeness. Note that improving completeness does not necessarily compromise soundness, although it can under some circumstances (for example, when adding synonyms to a thesaurus in order to sanction correspondence between objects with synonymous names, this does open new avenues for unsoundness).

This thesis has four subcontributions, which are identified and briefly discussed in the following subsections.

1.5.1 Inferring Correspondences

This subcontribution provides *correspondence heuristics* which sanction (allow) or veto (forbid) inferring correspondence between given pairs of objects in the codes being compared. Many of the existing comparison methods have to be provided with correspondences, rather than inferring them automatically.

The correspondence heuristics implemented are:

- *frame-class equivalence*: a quick initial test to veto correspondence between objects that are not even of the same 'type' (frame-class);
- *name equivalence*: inferring correspondence between named objects based on their names;
- *semantic equivalence/strength*: inferring correspondence between objects if they are semantically equivalent, or if one is semantically stronger than the other – this correspondence heuristic is used for all unnamed objects (usually expressions; in the absence of semantic knowledge, this correspondence heuristic reduces to syntactic identicality).

Ideally, these correspondence heuristics would be:

- *sound*: never infer correspondence between two objects that truly do not correspond (i.e. represent the same real-world concept);
- *complete*: always infer correspondence between two objects that truly do correspond.

However, the following two problems ensure that this could never be:

- *synonyms*: different names for the same real-world concept, or syntactically different but semantically equivalent expressions – the synonymy problem militates against completeness;
- *homonyms*: the same name for different real-world concepts, or syntactically identical but semantically different expressions – the homonymy problem militates against soundness.

The CCUS tackles the synonymy problem by means of thesauri, and by assessing the semantic equivalence and strength of expressions. It assumes that syntactically identical expressions are semantically equivalent – an assumption which is only false in cases of computational mapping (Section 1.3).

The bottom line is that name equivalence can always be fooled, and semantic equivalence is undecidable in general. The correspondence heuristics can only be made more and more complete by adding knowledge: entries to the thesaurus, and knowledge of semantic equivalence and strength as discussed in the next subsection.

1.5.2 Assessing Semantic Equivalence and Strength

This subcontribution aims to show that two expressions are semantically equivalent, that one is stronger than the other, or that they are semantically incomparable. Many existing comparison methods ignore the semantics of what they are comparing.

However, the current CCUS does not use traditional equational reasoning (it is only intended as an experimental prototype, after all); rather, it is a test-bed for some new ideas. Assessing semantic equivalence and strength is also novel in the way that it is used as a correspondence heuristic, as discussed in the previous subsection.

The CCUS facilitates the assessment of semantic equivalence through an internal representation which aims to be semantics-oriented as opposed to syntax-oriented: for example, commutative and associative expressions such as additions and multiplications are represented as flat, unordered bags (thus $(a + b) + c$ and $(b + c) + a$ are both represented by a bag $\{a, b, c\}$).

The CCUS uses *semantic equivalence/strength (SE/S) cases* to ignore irrelevant syntactic differences (e.g. $a > b$ is equivalent to $b < a$) and to assess that one expression is semantically stronger than another (e.g. $a > 10$ is stronger than $a > 5$).

However, the biggest way in which the CCUS assesses the semantic equivalence of syntactically different expressions is by comparing them at a higher level of abstraction: as discussed in the next subsection, it extracts and compares higher-level knowledge from the expressions when they differ in ways that cannot be resolved at the code level.

1.5.3 Combining Comparison with Extraction

This contribution aims to extend the completeness of assessing semantic equivalence, by extracting higher-level knowledge from two codes – specifically *higher-level constraints (HLCs)* from information models, as introduced in Section 1.2. As the example there demonstrates, this applies not just to expressions but to arbitrary code-fragments.

The use of combining comparison with the extraction of higher-level knowledge in order to assess semantic equivalence is highly novel – it could replace or enhance the use of traditional equational reasoning, which can show the semantic equivalence of two expressions. There are few examples of combining comparison with extraction in computer science.

The CCUS uses, as a building block, a *Code-Understanding System (CUS)* for EXPRESS, which was first presented in [Bro99]. The CUS extracts HLCs from an EXPRESS model – it is called by the CCUS to perform localised bouts of extraction from the vicinity of each difference in both models (rather than extracting all HLCs unselectively from each model). The CCUS compares the HLCs thus extracted from the two models; when it infers a correspondence between two HLCs, it infers an *abstract correspondence* (meaning a correspondence inferred at a higher-level of abstraction) between the code-fragments covered by the corresponding HLCs. For example, to infer that `(age >= 0) AND (age < 130)` corresponds to `{0 <= age < 130}`, it extracts from each expression a `range` HLC on `age` with an inclusive lower bound of 0 and an exclusive upper bound of 130; it compares these `range` HLCs, and when it infers that they correspond, it infers an abstract correspondence between the expressions they cover (i.e. between `(age >= 0) AND (age < 130)` and `{0 <= age < 130}`).

The completeness of assessing semantic equivalence (and hence of inferring correspondences) through the extraction and comparison of HLCs is determined by the set of extraction cases which the CUS uses to extract HLCs. An *extraction case* associates a code-fragment with a (usually singleton) set of HLCs to extract if the code-fragment in the case is matched by a code-fragment in an input model. The comparison of HLCs is performed using the same algorithm as the code-level comparison; completeness can be enhanced by adding extraction cases (which, although they are written by a human, are supposed to be correct by construction to ensure soundness).

The extraction cases and the SE/S cases are written in a declarative frame language called FIL-C, rather than being hardwired into the program – the CCUS has the objective of being data-driven in order to make it more flexible, more understandable to humans, and more extensible.

1.5.4 Selective Extraction and Comparison

This subcontribution pertains to efficiency in that it avoids unnecessary effort in the extraction and comparison of HLCs.

Selective extraction means extracting only those HLCs that are needed to resolve differences that cannot be resolved at the code level, rather than extracting all the HLCs that the system knows how to extract. Most code-understanding systems extract knowledge unselectively because they are not driven by a purpose – comparative code understanding is able to extract selectively, driven by what is needed for comparison.

The CCUS achieves selective extraction of HLCs in a passive manner: whenever it encounters a syntactic difference that it cannot resolve at the code level, it performs a localised bout of extraction from the vicinity of the difference in both models (hence two corresponding bouts of extraction).

The comparison of the HLCs thus extracted is also selective: rather than comparing every HLC extracted from one model with every HLC extracted from the other model, the CCUS compares a sensible subset:

- Only HLCs of the same frame-class are compared – this is achieved by storing the HLCs in a hash-table partitioned by frame-class.

- In the simplest mode of HLC-comparison, only HLCs from corresponding bouts of extraction are compared.

However, it is possible for corresponding HLCs to be in non-corresponding bouts of extraction, due to delocalised differences. Therefore it is necessary to remember HLCs from earlier bouts of extraction so that they can be compared with HLCs from later bouts of extraction:

- The simplest solution is to remember all HLCs unselectively.

- It is more efficient to use *remembrance heuristics* which remember HLCs selectively, based on certain assumptions about the HLCs that could correspond (completeness suffers when these assumptions do not hold). One such remembrance heuristic, *monogamy*, has been implemented: it assumes that HLC-correspondences are one-to-one, and thus forgets HLCs once they have been inferred to correspond.

1.6 Thesis Overview

This chapter has introduced comparative code understanding of information models, focusing on schema-to-schema mapping as a specific application context. It has discussed motivations for code understanding and comparative code understanding, and declared the contributions of the thesis.

The thesis has a contribution-oriented structure: each of the subcontributions has its own chapter, with literature review, ideas and implementation distributed among the subcontribution chapters rather than in chapters of their own. The subcontributions are intertwined with each other, so there are inevitably some circular dependences and a lot of cross-references.

Chapter 2 contains the main literature review – even though this thesis has a distributed literature review, not all topics fit into one of the subcontribution chapters. Chapter 2 covers literature pertaining to code understanding, code comparison, general comparison methods, and the related topics of data translation, schema-to-schema mapping, integration and mediation.

Chapter 3 gives an overview of the CCUS, including its internal representation of EXPRESS models and HLCs, a detailed walk-through of a simple example to illustrate the behaviour of the main CCUS algorithm, and a discussion of its aggregation-matching policies. Chapter 3 establishes an operational framework on which to hang the subcontribution chapters that follow.

Chapters 4 to 7 correspond to the four subcontributions identified in Section 1.5:

- Chapter 4 concerns inferring correspondences (Section 1.5.1).
- Chapter 5 concerns assessing semantic equivalence and semantic strength (Section 1.5.2). It intertwines with Chapter 4 in the way that semantic equivalence is used as a correspondence heuristic.
- Chapter 6 concerns combining comparison with the extraction of higher-level knowledge (Section 1.5.3). It depends on Chapter 4 and Chapter 5 in as much as inferring correspondences and assessing semantic equivalence and strength apply at the HLC level too, but the earlier subcontribution chapters do not concern themselves with HLCs.
- Chapter 7 concerns selective extraction and selective comparison of HLCs (Section 1.5.4), and cannot reasonably be read before Chapter 6.

Chapter 8 summarises the thesis, discusses the extent to which the contributions declared in Section 1.5 have been achieved, how they are evaluated and how they could generalise, and discusses avenues for future work.

Figure 1.7 shows the basic dependences between chapters, which constrain the order in which they should be read, although readers can read Chapter 8 immediately after Chapter 1 if they so wish.

1.7 Publications

This section briefly discusses the previous publications written by the Author.

The CUS, which extracts HLCs unselectively from a given EXPRESS model, was first presented in [Bro99]. It was billed as a *case-based reasoning (CBR)*

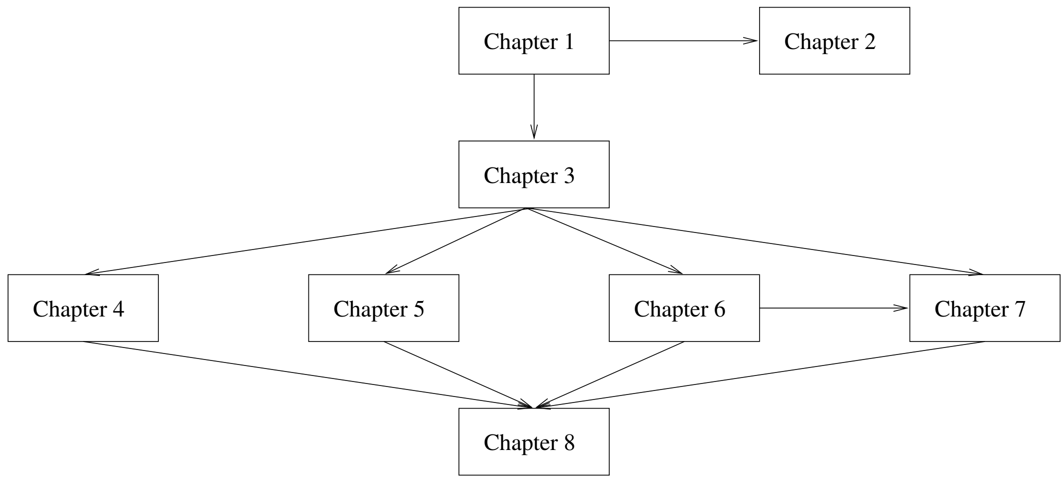


Figure 1.7: Chapter dependence graph.

(Section 2.3.3) approach to code understanding. However, it was decided to deemphasise the CBR aspects of the work in this thesis, because although the method of extracting HLCs (by matching actual code to cases) does involve simple retrieval and adaptation, it is too rudimentary to justify being categorised as CBR work, it turned out to be a case of trying to drive a square peg into a round hole, and the mechanics of extracting HLCs are much less important in a thesis on comparative code understanding which uses the CUS as a building block. Nevertheless, some CBR terminology remains (*extraction cases*, and there are now *navigation cases*, *semantic-equivalence/strength cases*, *subexpression cases* and *pretty-print cases*, all of which base their implementation on the CBR engine from the CUS).

The CUS was also presented in [BF00], which focuses on extracting knowledge about constraints from code.

The applicability of CBR to code understanding and code generation is discussed in [BF99], which distinguishes between *explicit CBR* and *covert CBR* (methods that can be regarded as CBR, but are not labelled as such). It is into the latter category that this thesis falls.

Chapter 2

Comparative Code Understanding

This chapter contains the main literature review of this thesis on comparative code understanding. It surveys the related areas of code understanding (Section 2.1), code comparison (Section 2.2), comparison in other fields including data warehousing, analogical and case-based reasoning (Section 2.3), and finally some areas that build on comparison, namely data translation, schema-to-schema mapping, information integration and mediation (Section 2.4).

This thesis makes the following distinction between *comparison* and *matching*:

- The term *comparison* is used when comparing two codes, or data in general, which are 'equal in status' as opposed to one having to match the other. An example is comparing the assertions `man(andrew)` and `man(philip)`.

- The term *matching* is used when comparing a situation with a 'rule' to see if it is applicable, as opposed to comparing two situations. Rather than being 'equal in status', the situation must match the rule. The rule 'dominates' the situation in the sense that every part of the rule's antecedent must be matched by a situation (but not vice versa) for the rule to be applicable in that situation. An example is matching the assertion `man(andrew)` to the rule `IF man(?x) THEN mortal(?x)`.

2.1 Code Understanding (Reverse Engineering)

This section serves two orthogonal purposes: to review the relevant literature on code understanding for its own sake, and also for the sake of comparison – in the guise of matching the code to be understood to rules for extracting knowledge from code.

Code understanding means extracting knowledge from code – which often means program code (in which case code understanding is usually referred to as *program understanding* [BMW94] or *reverse engineering* [CI90]), but can mean code written in *any* formal language, such as database schemata and queries, frame languages, rule languages, constraint languages, and grammars. For example, an entity-relationship (ER) diagram can be reverse-engineered from a relational database schema, using ER-to-relational mapping (Section 6.8 of [EN94]) in reverse. Code understanding can even be applied to code representing hardware rather than software (e.g. reverse-engineering abstractions from digital logic circuits [Mir93]). This thesis is concerned with understanding EXPRESS information models as a particular kind of code.

The motivations for code understanding tend to relate to *software maintenance* (Chapter 18 of [vV93]) – tasks ranging from debugging, adapting the software to new environments (hardware platforms or operating systems), and adapting the software to new requirements for functionality. These tasks entail first understanding the code (*reverse engineering*), then making the necessary changes (*forward engineering*). Such combinations of reverse engineering and forward engineering are known as *reengineering* [CI90].

Reengineering is often applied to *legacy systems*: old software written in old languages to run on old platforms. The code of a legacy system may have to be not only translated, but also restructured (since old software often tends to be badly structured, due to the language it is written in and also due to efficiency considerations which were important for slow machines and lack of compiler optimisations – optimising code tends to make it more abstruse and less reusable) and packaged as reusable components. Reengineering code to make it reusable is known as *reuse reengineering* [CLM95b].

The motivation for code understanding does not necessarily have to entail modifying the code, though. It may be desirable to abstract the ideas from the implementation detail for academic interest (perhaps to see how to go about writing a similar code), or to extract domain knowledge that is not documented elsewhere (such as business rules [NEK94]). Code understanding can also be used to discover how rival systems work, for example disassembling or reverse-engineering machine code (which can be illegal [Sam90]), which is why some code is *obfuscated* (transformed to be difficult to understand whilst preserving the semantics of the original code) [CTL97].

The motivation for code understanding in this thesis is to facilitate semantic comparison at an abstract level when code-level comparison does not suffice. There is little existing work on combining comparison with the extraction of higher-level knowledge (see Chapter 6).

Various kinds of knowledge, both structural and semantic, can be extracted from code (Section 2.2.1 of [Bro99]). For example:

- *Subsystem hierarchies* can be extracted from the nested substructures of a code [MOTU93].
- *Call graphs* can be extracted from programs to capture the structure of function calls [CLM95b].
- *Control-flow graphs* can be extracted from programs to capture the control dependences between statements ('this statement must be executed before that statement') [CLM95b].
- *Data-flow graphs* can be extracted from programs to capture the data dependences between statements and values ('this statement requires this value, which is produced by that statement'). They are used, for example, for data-flow analysis in optimising compilers [FOW87].
- *Program dependence graphs* are combined control- and data-flow graphs [FOW87].
- *Slices* are subsets of the statements of a program that fulfil a given slicing criterion (such as computing the value of a given output variable) [NEK94]. Slices can be computed from program dependence graphs, which facilitate slicing and are language-independent [TL98]. Slicing is particularly useful for reuse reengineering, because a slice that computes just one external function can be packaged as a reusable component [CLM95a]. Slicing is discussed in Section 2.1.2.

- *Clichés* (*plans*) are common data-structures (e.g. linked list, binary tree) and algorithms (e.g. sorting, inserting an element into a data-structure if it is not there already) that can be recognised in code [RW90]. Thus code-fragments can be clustered into higher-level abstractions. Clichés are discussed in Section 2.1.1.

2.1.1 Cliché-based Code Understanding

The cliché-based approaches to code understanding are the most important for this thesis, because clichés are very similar to the higher-level constraints (HLCs) extracted from EXPRESS models (Chapter 6) – small chunks of knowledge that cover the abstract semantics of code-fragments. The big difference (other than the fact that the existing cliché-based approaches are applied to program code rather than to information models) is in the purpose of extraction:

- The purpose of extracting clichés from program code is typically to build up a complete understanding of the program and ultimately discover its overall goal (e.g. that it is an implementation of a sorting algorithm). A tree of clichés for the program is built up, whose leaves are code-fragments that match the first-level clichés, and whose root is a cliché representing the overall goal of the program (Figure 2.1).

- The purpose of extracting HLCs from EXPRESS models is to assess whether the constraints in the models are semantically equivalent despite syntactic differences at the code level. Constraints are very localised, and it would be a much more vague and difficult problem to abstract the overall goal of an EXPRESS model (e.g. that it is a model of electronic design) than to do so for a program, which has concrete functionality. Thus the HLCs extracted from an EXPRESS model tend to form a forest of tiny trees, each one or two levels deep.

Cliché-extraction is performed as follows. There is a library of *overlays*, each overlay consisting of:

- an *implementation cliché* to which fragments of the input code (or lower-level clichés) are matched;

- a *specification cliché* which is extracted from a code-fragment if the code-fragment successfully matches the implementation cliché;

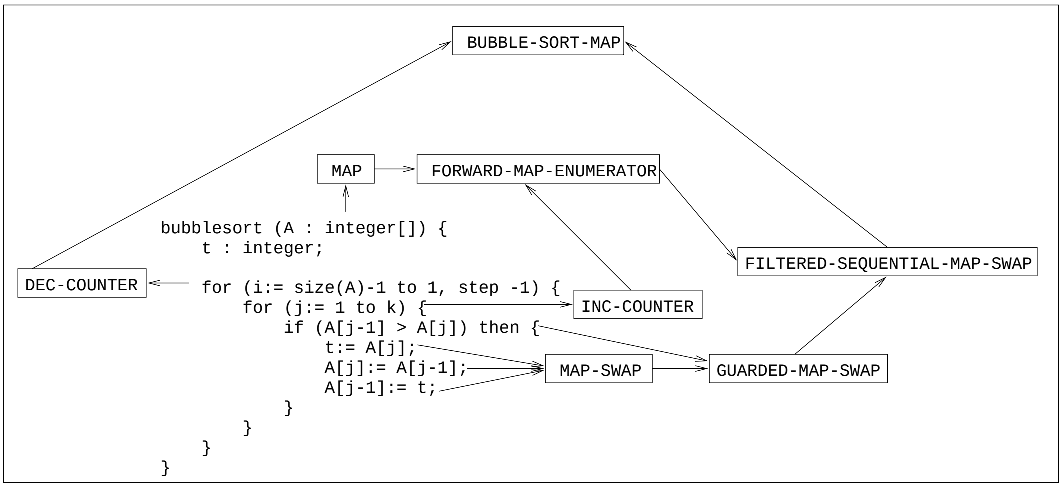


Figure 2.1: A program and its tree of clichés (adapted from [HN90]).

- *correspondences* between the implementation cliché and the specification cliché.

So the clichés in Figure 2.1 were extracted by matching code-fragments and lower-level clichés to the implementation clichés of overlays, and instantiating the specification clichés of those overlays they match (Figure 2.1 does not show the details of the clichés – they would be tagged with information such as which variables they pertain to, and other values picked up from the source code).

When a specification cliché is extracted, it is instantiated by substituting values for variables[1] which were bound during the matching of the code-fragment to the implementation cliché.

Analogously, HLCs are extracted from EXPRESS models by matching code-fragments to the problem parts of *extraction cases*, extracting HLCs according to the solution part if the match succeeds (Appendix B, Section B.1). Thus extraction cases are analogous to overlays, and the problem and solution parts of an extraction case are analogous to the implementation and specification clichés in an overlay.

Cliché-extraction can be either:

- *bottom-up*: working from the code towards the top-level cliché, as assumed above, or

[1]Variables in the overlay as opposed to the variables in the program.

- *top-down*: start by hypothesising the top-level cliché, and extract lower-level hypothesised clichés until the code can be matched to them directly.

Bottom-up extraction can suffer from combinatorial explosion, the worse the bigger the code, especially with a large cliché library where one code-fragment can match many first-level clichés, which can match many second-level clichés, and so on. However, this combinatorial explosion (which has not proved to be a problem in extracting HLCs from EXPRESS models – Section 8.2) can be alleviated by the following techniques [Qui94]:

- *Indexing* overlays so that only the relevant ones will be retrieved when considering a given code-fragment, rather than matching that code-fragment to all overlays that contain it. The Code-Understanding System (CUS) used to extract HLCs from EXPRESS models (Appendix B) indexes each extraction case by the frame-class of the top-level frame of the code-fragment in its problem part, rather than matching every code-fragment to every extraction case.

- *Specialising* each extracted cliché as far as possible before attempting to extract further clichés, e.g. specialising `MULTIPLY(x,x)` to `SQUARE(x)`.

- Inferring clichés whose existence is *implied* by the clichés already extracted (e.g. on recognising an occurrence of Pythagoras's Theorem, infer that the variables represent two points).

Top-down extraction is only applicable when clichés can be hypothesised. The user could provide hypotheses interactively, hypotheses could be triggered by *beacons* in the code (e.g. recognising a swap operation could trigger a sorting algorithm to be hypothesised) [vMV95] or, in comparative code understanding, clichés could be extracted bottom-up from one code and top-down from the other (as discussed in Section 6.2.2).

There are various methods of cliché-based code understanding – for example Rich and Wills [RW90] view it as graph-parsing and define *plans* as a specific (language-independent) representation of clichés, while Quilici [Qui94] calls clichés *plans*; Harandi and Ning [HN90] view clichés as *events* and represent the tree of clichés extracted from a program as a justification-based truth-maintenance system – but they are all essentially the same.

Clichés can be used for *code generation* as well as for code understanding, as in the Programmer's Apprentice [RW88]. A code-generation system is given a specification of a program to implement, and it does so by matching the specification to the specification clichés of overlays, and extracting the implementation clichés of the successfully matched overlays.

Cliché-based code understanding also relates to code comparison (Section 2.2) in the way that code-fragments are matched to implementation clichés (recall the comparison/matching distinction from the beginning of this chapter).

2.1.2 Slicing

This section introduces slicing, because slicing is a prerequisite for the discussion of program integration in Section 2.2.3 (slicing itself belongs to Section 2.1 because it applies to one code as opposed to comparing two codes, which is the subject of Section 2.2).

Slicing is a kind of *program segmentation* [NEK94], the purpose of which is to isolate functionally-related code-fragments from a program. Program segmentation extracts groups of related statements, even if the statements are far apart in the code (which is helpful for recognising *delocalised plans* [LS86], possibly using cliché-based techniques – Section 2.1.1). Program segmentation facilitates program comprehension, and is useful for reengineering-tasks such as debugging and automatic parallelisation [TL98].

A *slice* is a subset of the statements in a program that fulfil a given *slicing criterion*, and is itself a program. Slicing can be used to isolate the external functions of a program so that they can be packaged as reusable components – an example of reuse reengineering [CLM95a].

Three varieties of slicing [NEK94] are:

- *Backward slice*: Given an output variable, include in the slice all statements that contribute to the computation of that variable.

- *Forward slice*: Given an input variable, include in the slice all statements that could be affected by that variable.

- *Conditional slice*: Given a condition (i.e. a logical expression), include in the slice all statements that could be executed under that condition.

`input n;`
`s:= 0; f:= 1; i:= 1;`
`while (i <= n) {`
`  s:= s + i;`
`  f:= f * i;`
`  i:= i + 1;`
`}`
`print s;`
`print f;`
	`input n;`
	`f:= 1; i:= 1;`
	`while (i <= n) {`
	`  f:= f * i;`
	`  i:= i + 1;`
	`}`
	`print f;`

Figure 2.2: A program and a slice of that program.

Figure 2.2 gives a simple example of program-slicing: the program is sliced backwards with respect to `f`, thus producing a program that computes a single external function (factorial).

Slicing could be extended to code other than programs. For example, an EXPRESS information model could be sliced in order to isolate the set of entities associated with a given entity, or to simplify the constraints in a model under a given condition.

A more detailed review of program segmentation and slicing is given in Section 2.3.1 of [Bro99].

Section 2.1 has discussed *code understanding*, which entails extracting knowledge from a single code. Section 2.2 looks at *code comparison*, which entails comparing two codes.

2.2 Code Comparison

This section reviews the literature which pertains to comparing two codes. It discusses systems which compare code on one of three levels: text, structure or semantics (Sections 2.2.1 to 2.2.3). Section 2.2.4 then discusses systems which perform *schema comparison* – database schemata being a particular kind of code which has much in common with information models. Finally, Section 2.2.5 discusses systems which compare the executions of two programs rather than the codes of those programs.

2.2.1 Textual Code Comparison

The most low-level way to compare two codes is to compare two files byte-for-byte. This may be appropriate for comparing two versions of a machine-code program (particularly if there is no relocation). Byte comparison is the simplest and least flexible method of comparison, suitable only for comparing data that are in a rigid, fixed-length format, or as a quick test of whether two files are identical (this test can fail immediately if the files are of different lengths, or upon encountering the first difference).

The next level up is to compare the source codes as text, using a standard text-comparison tool such as UNIX's `diff` [dif]. This adds the ability to align corresponding substrings of text. For example, in comparing "we hope you choke" and "we hope that you choke", a text-comparison algorithm (working from left to right) would report the insertion of "that " and proceed to compare "you choke".[2]

The `diff` tool is useful for comparing two codes which are minor variants (e.g. revisions) of each other, where the differences are at the level of lines (e.g. if an EXPRESS model is modified by inserting or deleting an attribute or an entity, or if a program is modified by inserting a statement). The alignment algorithm is prone to quirky features such as depending on which way round the files are compared (e.g. comparing "we hope that you choke" and "we hope you that choke" reports an insertion of "that " and a deletion of "that ", but comparing them the other way around reports an insertion of "you " and a deletion of "you ") – comparison is supposed to be commutative. It cannot abstract the fact that "that" corresponds to "that" and "you" corresponds to "you" but they are in different orders.

Text comparison has application in version-control systems such as the Concurrent Versions System (CVS) [Col], where files (particularly code) are stored in a central repository. To modify a file F, a CVS user A signs out a local copy of F, edits it locally to produce a modified version F_A, and then submits F_A back into the repository. However, another user B might have taken a local copy of F before it was upgraded to F_A, and produced another modified version F_B. When user B submits F_B back to the CVS repository, it would lose the changes from F to F_A if it simply replaced the file with F_B (an instance of the *lost-update problem* – Section 17.1.3 of [EN94]). So CVS merges the changes from F to F_A with the changes from F to F_B to produce an integrated file F_{AB} which incorporates both sets of changes. This process is called *file integration* (Figure 2.3).

Textual file integration suffers from the problem that the two sets of changes

[2] `diff` only aligns text and reports differences on a line-by-line basis, so for this example to work, it is necessary to replace spaces with newlines.

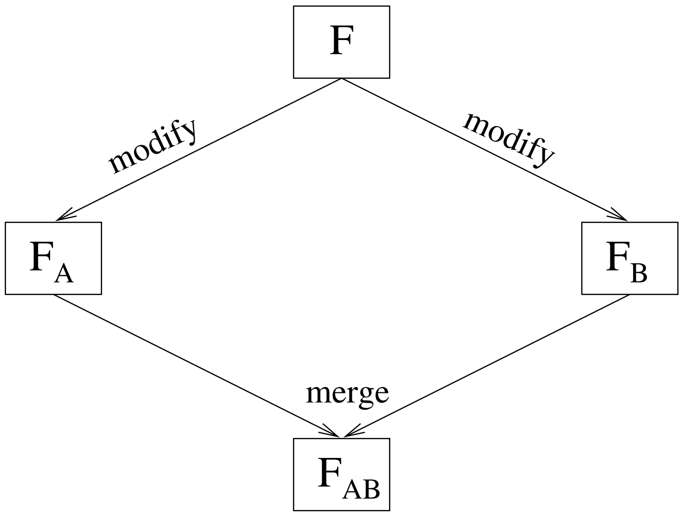


Figure 2.3: File integration.

may interfere if they are too close – to overcome this problem, it is necessary to consider the semantics of the codes to be integrated (Section 2.2.3).

Text comparison is related to *string-searching* – finding a *target string* (e.g. "choke") in a *source text* ("we hope that you choke"), as in the Boyer-Moore algorithm [BM77], which efficiently examines **fewer** characters in the source text, the **longer** the target string. The relation of text comparison to string-searching is an example of the comparison/matching distinction made at the beginning of this chapter – simply applying the Boyer-Moore algorithm to text comparison with the shorter text as the target string would be such a coarse-grained approach that it would find at most one correspondence (although the Boyer-Moore algorithm could be useful for quickly locating corresponding substrings using keywords as target strings).

Text comparison does nothing to exploit the fact that the texts being compared are codes. It does not exploit the structure of the codes, let alone the semantics. It can align corresponding substrings when there are insertions or deletions, but anything more complicated causes it to report a lot of differences – differences that it cannot resolve (i.e. infer that two substrings correspond even though they are not in corresponding positions). The next level up is to compare the codes structurally rather than textually.

2.2.2 Structural Code Comparison

This section concerns methods that go beyond aligning text, to the hierarchical comparison of nested structures. The codes to be compared are parsed into hierarchical representations such as abstract syntax trees (Section 5.2 of [ASU86]), which facilitate a hierarchical structure comparison as opposed to a sequential text comparison. Such a comparison algorithm walks the representations of the codes in parallel, recursing in and out of corresponding substructures (e.g. in comparing two entities, it compares each corresponding pair of their attributes; in comparing attributes, it compares their types, which may themselves be entities) rather than comparing the codes sequentially from beginning to end. This is more flexible than textual code comparison – it handles, with great facility, differences in order where the order does not matter (e.g. the order in which entities are declared in EXPRESS models).

The foundation of the Comparative Code-Understanding System (CCUS) is an algorithm that structurally compares two EXPRESS models, as described in Chapter 3. The EXPRESS models are parsed into networks of frames (Section 3.2), which the CCUS walks in parallel, inferring correspondences (between entities, between attributes, between expressions, and so on) and finding differences.

Paul Pun [Pun91] presents a method for deducing mappings between knowledge bases and application data-structures (the former are complex structures for rich, semantic concepts; the latter simple structures for efficient processing). His method generates a *knowledge interpretation* from a knowledge-base schema and a *data interpretation* from an application data-structure – these interpretations consist of entities and inter-entity relationships. Each entity is either *atomic* (of an elementary type), a *composition* (defined in terms of other entities, as in a record) or a *stream* (a sequence of objects of the same type). An *entity-function tree* is generated for the knowledge interpretation, and one for the data interpretation (nodes represent entities, links represent inter-entity relationships). A mapping is deduced between the two entity-function trees using tree isomorphism – after the entity-function tree for the data interpretation has been reduced to an equivalent terminal form[3] (using two reduction rules: combining two compositions, and combining a stream with a composition).

[3] Pun assumes that the knowledge interpretation is in terminal form without the need for reduction.

Pun's method only compares the *structure* of objects, which can lead to the ambiguity of multiple mappings, between which a human must choose. Other correspondence heuristics could be used to choose between them, such as name equivalence and semantic equivalence (Section 4.2).

Pun's work can be seen as an instance of code comparison. The generation of entity-function trees is analogous to generating frame-representations of EXPRESS models, and the deduction of mappings using isomorphism is analogous to comparing the frame-representations and inferring correspondences. The CCUS adds the ability to assess semantic equivalence between the constraints in two models (Chapter 5) – whereas Pun effectively considered only entities and attributes – and the CCUS combines comparison with the extraction of higher-level knowledge in order to assess abstract semantic equivalence (Chapter 6). On the one hand, Pun facilitates the deduction of structural mappings by the use of reduction, but on the other hand his method appears to be very inflexible in that if the entity-function trees are not isomorphic then the knowledge interpretation and the data interpretation are simply declared to be incompatible and no mappings are deduced. Knowledge-base schemata and application data-structures are two different kinds of code being compared, whereas the current CCUS compares only EXPRESS models (although the CCUS could be extended to compare different kinds of code – see Section 8.3.2).

The Structure-Mapping Engine [FFG89] performs structural comparison, though not of code. It will be discussed in the context of analogical reasoning in Section 2.3.3.

Structural comparison does not exploit the semantics of the code being compared – it only exploits the structure. Considering the semantics of the codes being compared is the next level up.

2.2.3 Semantic Code Comparison

This section pertains to methods for comparing code *semantically* – comparative *understanding* as opposed to mere comparison.

The CCUS attempts to assess the semantic equivalence of code when it encounters differences during structural comparison (based on the assumption that where the code is syntactically identical, it is semantically equivalent):

- It attempts to assess semantic equivalence (e.g. `age >= 18` is equivalent to

`18 <= age`) and semantic strength (e.g. `age >= 21` is semantically stronger than `age >= 18`) at the code level using *semantic-equivalence/strength cases* (Section 5.3).

- Failing that, it attempts to assess semantic equivalence at an abstract level, by extracting and comparing HLCs (e.g. `(age >= 0) AND (age < 130)` can be seen to be equivalent to {`0 <= age < 130`} by extracting an HLC of type `range` from each expression, and comparing these HLCs) (Chapter 6).

There are systems that compare code semantically by assessing the semantic equivalence of expressions: for example, comparing two specifications by comparing their preconditions and comparing their postconditions [ZW97] (Section 5.2.2.4), assessing the equivalence and containment of database queries [CV94] (Section 5.2.2.5), and assessing the operational equivalence of expressions in a functional programming language [Tal98] (Section 5.2.3.4). However, these systems specialise in comparing expressions, whereas the CCUS performs a structural comparison of the whole codes (including expressions) and attempts to assess semantic equivalence when it encounters differences (as mentioned above). The CCUS could, however, be enhanced with the ability to perform equational reasoning using conventional mathematical equivalence laws (see Section 6.1 for a discussion of the relative merits of equational reasoning, and extracting and comparing HLCs to assess equivalence).

Program integration [BHR95] entails merging the changes from two independently-modified revisions of a base program to produce an integrated program which incorporates the changes from the base program to the first revision, and the changes from the base program to the second revision – it is a specialisation of file integration (Section 2.2.1).

The problem with applying textual file integration to program integration (à la CVS) is that there are no guarantees about execution behaviour – if a change from one revision is too close to a change from another revision then the changes may interfere. Figure 2.4 gives a correct example of program integration – Revision A renames the variable `f` to `factorial`, and Revision B adds a statement to print the final value of `f` – but textual file integration, which knows nothing of variables, would produce an integrated program with `print f` instead of `print factorial` as the final statement.

Binkley, Horwitz and Reps [BHR95] present a semantics-based algorithm for program integration, which uses slicing (Section 2.1.2) and operates on program

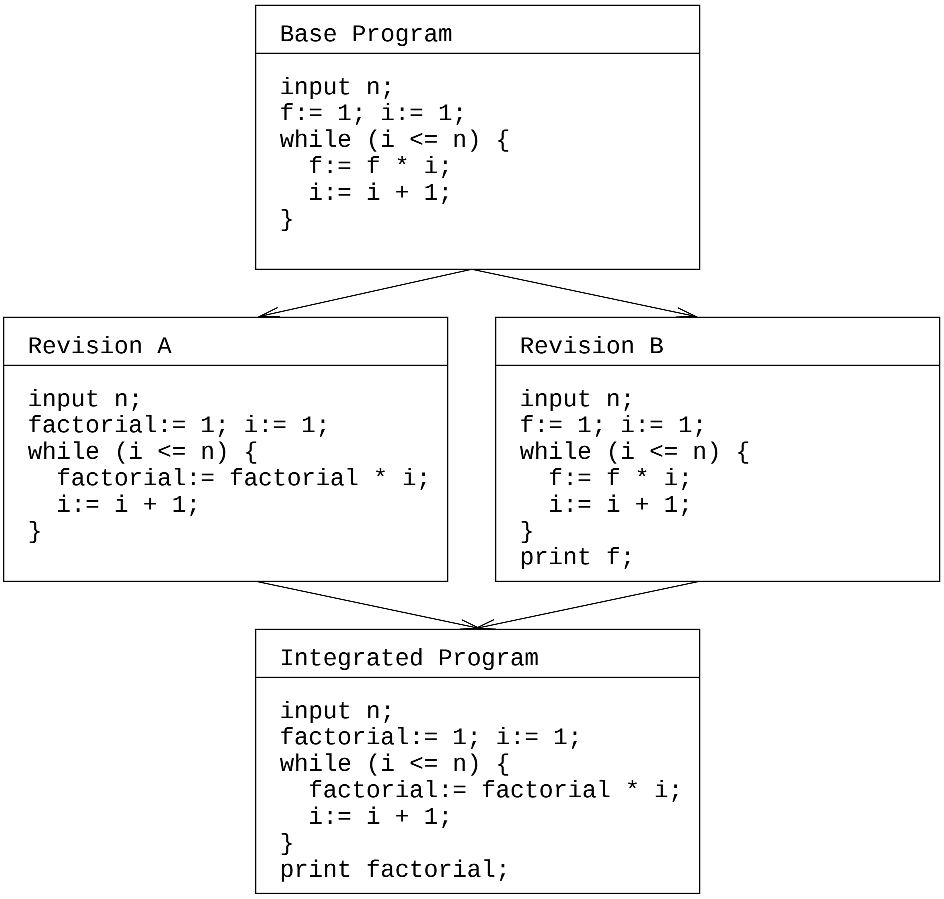


Figure 2.4: An example of program integration.

dependence graphs (PDGs) (Section 2.1) as an intermediate code-representation. The algorithm proceeds as follows:

- It compares the (backward) slice of each node in the PDG for the base program with the slice of the corresponding[4] node in the PDG for the first revision,[5] and thus partitions the nodes into a set of *affected nodes* (those for which the slices differ) and a set of *preserved nodes*.
- Likewise, it computes the affected and preserved nodes from the base program to the second revision.
- It combines these slices (the preserved nodes of the base program and both revisions, the affected nodes from the base program to the first revision, and the affected nodes from the base program to the second revision) using graph union to form the PDG of the integrated program – if and only if it does not detect any interference between the affected nodes (*Type I interference*).
- It converts the integrated PDG into an actual program (*program reconstitution*). This can fail if it is not possible to order the program components to respect all of the dependences (*Type II interference*).

Semantics-based program integration is *sound* (if it succeeds and returns an integrated program, then it is guaranteed to be a correctly integrated program – unlike textual program integration); the big issue is *completeness* (a more complete program integration algorithm succeeds and returns an integrated program in cases where a less complete program integration algorithm would report failure). For example, one contribution to completeness was to accommodate semantics-preserving transformations [YHR92]; another was the extension to programs with multiple procedures [BHR95] (which has to take different calling contexts into account, and entails interprocedural slicing).

It might be possible to improve the completeness of program integration further by combining comparison with the extraction of higher-level knowledge (Chapter 6) – comparing at a higher level of abstraction should facilitate the accommodation of more and more semantics-preserving transformations.

[4]They use a special editor which automatically tags program components such that corresponding components have the same tag.

[5]Comparing slices entails comparing sets of nodes in the PDGs representing the slices.

2.2.4 Schema Comparison

This section concerns systems that compare two database schemata. Schema comparison warrants its own section because it is the closest thing to comparing information models, which are like database schemata except that their purpose is to communicate the semantics of a domain, rather than efficient physical data representation.

There is a commercial tool called Schema Comparison [Pro] for comparing two ORACLE database schemata. It performs very low level, basic comparison, just comparing corresponding components of the two schemata (tables, indexes, constraints, sequences, synonyms, roles, comments, triggers, views, procedures, packages and privileges) with no extraction of higher-level knowledge. In this regard it does for ORACLE schemata what the code-level CCUS (Chapter 3) does for EXPRESS models. It is less flexible than the CCUS in that it requires corresponding tables to have the same names and identical structures, whereas the CCUS reports differences without necessarily vetoing correspondence, and can use a name-thesaurus to sanction correspondence between objects with synonymous names (Section 4.2.4).

However, the Schema Comparison tool does not just compare two schemata, it can also *synchronise* the schemata, i.e. copy objects (definitions) from one schema to the other, which is useful for schema-to-schema mapping (Section 1.1). To use an EXPRESS example, consider comparing schemata `a` and `b` in Figure 2.5. This reports the difference that the attribute `nationality` is missing in schema `a`, so synchronising the two schemata copies `nationality` from `b` to `a`. Thus instances of `b.person` can be mapped to `a.person` without losing their values for `nationality`. This is an example of comparative code generation (Section 1.3), and is like file integration (Section 2.2.1) except that differences are copied from one schema to the other (with or without asking the user at the time that each difference is reported) rather than producing a single integrated schema.

Schema comparison is the first step in the process of *schema integration* [BLN86] (merging two or more local schemata to form a single global schema – see also Sections 4.2.4, 5.2.2.1 and 6.1):

1. *schema comparison*: analysing and comparing schemata in order to determine correspondences (different representations of the same concepts in particular);

SCHEMA a;
  ENTITY person;
    name : STRING;
    age : INTEGER;
  END_ENTITY;
END_SCHEMA;
	SCHEMA b;
	  ENTITY person;
	    name : STRING;
	    age : INTEGER;
	    nationality : STRING;
	  END_ENTITY;
	END_SCHEMA;

Figure 2.5: An example for synchronisation with two EXPRESS models.

2. *schema conforming*: modifying one or both of the schemata until each phenomenon in the universe of discourse is represented in the same way in both schemata;

3. *schema merging*: superimposing the schemata to obtain a single, integrated schema.

Sari Hakkarainen's work on schema comparison [Hak98] (see also Section 6.1) focuses on comparing the *dynamic aspects* of conceptual models (specifically comparing event-type specifications in the Delphi modelling language). Schema comparison had been successful for structural, terminological and relational conflicts, but not for behavioural conflicts.

Event comparison facilitates the identification of incomplete *correspondence assertions* at a high level of abstraction – the temporal aspect must be taken into account. Schema comparison establishes two kinds of associations: *correspondences* and *discrepancies*, just as the two basic outputs of the CCUS are the correspondences inferred and the differences reported. These associations facilitate forming a single united repository (for schema integration) or establishing exchange connections (for schema-to-schema mapping).

Hakkarainen solves the problem of semantic heterogeneity in schema comparison by *semantically enriching* the schemata. Since the use of semantic enrichment in schema comparison is a rare example of combining comparison with the extraction of higher-level knowledge, it is discussed in Section 6.1. Semantic enrichment also relates to code understanding (Section 2.1).

Hakkarainen's work is thus strongly related to the work presented in this thesis, but the CCUS compares information models rather than event specifications, focusing on constraints rather than dynamic schema structures. Hakkarainen's work relates to two of the four subcontributions of this thesis: inferring correspondences (Chapter 4) and combining comparison with extraction (Chapter 6), but does not appear to address assessing semantic equivalence and semantic strength (Chapter 5), nor selective extraction and comparison (Chapter 7).

2.2.5 Comparing Executions

This subsection concludes this section on code comparison with a brief discussion of systems that compare two running programs rather than comparing their source codes.

Relative debugging [AFMS96] entails comparing the executions of two running programs, where one program is a revision (modification) of the other. Modifying a program may introduce subtle errors; a *relative debugger* runs both revisions in parallel, comparing program states (internal data) and reporting differences. Relative debugging is not tedious and error-prone like running the programs in single debuggers and visually comparing the program states. Another approach is to insert print statements in both programs (whereas relative debugging does not require the programs to be edited) and `diff` the output (relative debugging does not need to store the whole program outputs). Unlike `diff`, relative debugging can take data-types into account, and can even handle different data-representations (which entails data translation – cf. Section 2.4) – the two programs do not even have to be written in the same language or run on the same platform. Relative debugging is useful for regression testing, and for demonstrating (rather than proving) the functional equivalence of two programs.

Another example of run-time comparison is checking the implementation of an abstract data-type against its algebraic specification [AH00]. The user has to add code to the implementation to map implementation-states to the abstract values of the specification (which in itself has the advantage of forcing the user to understand this mapping), so that the implementation can automatically check itself against the specification. To achieve this, an automatically-generated direct implementation of the specification is used, which can compute the abstract result of any sequence of the specification's operations – these abstract values are compared with the corresponding mapped implementation-states, and any violation of the specification will be detected. Although it is often assumed that the specification is an oracle with respect to which the implementation is supposed to be correct, this method is, in fact, as much a check of the specification as of the implementation, since both are code written by humans and could contain errors. The likelihood of common-mode failure is reduced by the fact that these codes differ so radically.

Section 2.2 has reviewed the literature pertaining to code comparison; the next section considers comparison in related fields.

2.3 Comparison in Related Fields

This section reviews general comparison methods, as opposed to the specifically code comparison methods in Section 2.2 – though the ideas in this section do relate to code comparison. Section 2.3.1 reviews the basic topic of comparing hierarchically-structured data; Section 2.3.2 reviews data warehousing; Section 2.3.3 reviews analogical and case-based reasoning, and structure mapping.

2.3.1 Comparing Hierarchically-Structured Data

This subsection pertains to methods that compare hierarchically-structured data, of which the comparison of abstract-syntax-tree representations of code (Section 2.2.2) is an example, but the methods reviewed in this section are intended for instance-data rather than data that represent code.

MH-DIFF [CGM97] compares data snapshots in order to detect changes. The 'H' in "MH-DIFF" stands for *hierarchical* (it compares hierarchically-structured data). The 'M' stands for *meaningful*: it reports the changes at as high an abstraction level as possible. For example, it reports an 'update node-label' operation rather than reporting a deletion and an insertion; it reports a 'copy subtree' operation rather than reporting a set of insertions; it reports a 'move subtree' operation rather than reporting a set of deletions and insertions. Whereas traditional change-detection algorithms report changes in terms of primitive operations (insert-node and delete-node), MH-DIFF uses a heuristic change-detection algorithm that computes near-minimal descriptions of the changes.

MH-DIFF automatically infers correspondences between nodes, whereas traditional change-detection algorithms rely on the provision of node identifiers. MH-DIFF infers correspondences based not just on node-labels, but on the structure of the trees. MH-DIFF computes a minimum-cost *edit script* (a sequence of transformations from one tree to the other) by computing a minimum-cost edge cover of a bipartite graph.

Inferring high-level changes from one tree to another does not constitute combining comparison with the extraction of higher-level knowledge in the sense of

Chapter 6, but it is an interesting idea in its own right – rather than extracting higher-level knowledge in order to infer correspondences at a high-level of abstraction, it is actually extracting higher-level correspondences. An application of this would be to compare an EXPRESS model with a transformed version of itself and infer what the transformation was (e.g. flattening – Section 1.4.2.1). Inferring higher-level correspondences raises the abstraction level of the changes from one tree (model) to the other, whereas combining comparison and extraction raises the abstraction level of the models in order to see abstract correspondences – it exploits the semantics of what it is comparing whereas MH-DIFF only exploits the structure.

TreeDiff [WSC+97] compares two documents in order to perform the following tasks:

- *structural matching*: comparing old and new versions of a document;
- *structural discovery*: finding common portions of two documents.

TreeDiff compares two LaTeX documents using approximate tree-matching. The following operators are provided for comparing two documents A and B:

- `symdifference(A,B)`: find all the differences between A and B (changed elements, and elements in one document but not the other – bidirectional comparison);
- `difference(A,B)`: find the changes from A to B (changed elements, and elements in A but not in B – unidirectional matching of B to A);
- `intersection(A,B)`: find the elements in both A and B;
- `union(A,B)`: find the elements in either A or B, or both;
- `mergeable(O,A,B)`: return true if A and B, which are both revisions of O, can be merged without interference – cf. file integration (Section 2.2.1);
- `merge(O,A,B)`: perform the merge of A and B, with A's changes superseding B's changes if `mergeable(O,A,B)` is false.

The first four operations can be thought of as specialised versions of what the CCUS does in comparing two EXPRESS models – the CCUS infers correspondences and reports all differences bidirectionally. However, TreeDiff just performs straightforward structural comparison (cf. Section 2.2.2) with no regard for semantics, whereas when the CCUS encounters a difference between two elements, it attempts to assess semantic equivalence and hence decide whether the two elements correspond (Section 4.2.5).

Wang, Zhang and Shasha [WZS95] use string comparison, graph comparison and tree comparison to perform tasks such as:

- finding approximately common regular expressions in protein/DNA sequences;
- finding approximately common substructures in RNA trees;
- aligning LaTeX/SGML documents, and programs, according to their hierarchical structure (cf. Section 2.2.2).

2.3.2 Data Warehousing

A *data warehouse* holds integrated data from multiple sources. It stores three levels of data [Squ95]:

- *detailed data*;
- *summarised data*;
- *metadata*: data about data (e.g. the contents of the data warehouse and where they came from, the context of the data, the meaning of the data, and the history of the data – a data warehouse typically stores large amounts of data, collected over many years).

Data is extracted from legacy systems, and passes into the data warehouse via integration and transformation programs [Squ95] – cf. Section 2.4. When the source-data are updated, there are three ways to infer the modifications to be made to the warehouse [LGM96]:

- The application sends modifications to the warehouse (i.e. the application code has to be edited just to do this).
- Parse the log file to obtain the modifications (typically, only database administrators have access to the log file).
- Infer the modifications by comparing snapshots of each data-source (as it was when the warehouse was last updated, and as it is now) using a *snapshot-differential* algorithm.

Labio and García-Molina [LGM96] present a snapshot-differential algorithm that is closely related to *joins* in relational databases.

- A *snapshot* is a file containing a set of records.
- Each *modification* inferred is either an *update*, a *deletion* or an *insertion*. A snapshot-differential algorithm must avoid *useless insert-delete pairs* because they compromise correctness, and should avoid *useless delete-insert pairs* because it is more efficient to report them as updates.

Labio and García-Molina use a *window algorithm* that works very well if the snapshots are not "very different", which is similar to the CCUS's unordered aggregation-matching policy (Section 3.4.1) in the way that it walks the snapshots in parallel, with wrap-around, preferring to compare records in corresponding positions (relative to the previous pair of corresponding records).

The snapshot-differential algorithm of Labio and García-Molina seems very similar to the CCUS comparison algorithm (Section 3.3), except that Labio and García-Molina are not concerned with hierarchical structures. The snapshot-differential problem is also related to the text-comparison problem (Section 2.2.1), but text comparison is concerned with *sequences* of text whereas the snapshot-differential algorithm handles *sets* of records in any order.

Data warehousing is closely related to *data mining* (a kind of *knowledge discovery*), which is about extracting, from data, implicit knowledge such as patterns and relationships [FPSM92, Ede97] (e.g. association rules [AIS93] such as "80% of customers who purchase a computer also purchase a printer" or "90% of times Monica Seles gets her first serve in, she wins the point").

Data mining is a bit like code understanding (Section 2.1), except that it operates at the object level (extracting knowledge from raw data) rather than at the meta level (extracting knowledge from code), and it extracts statistical patterns across the data rather than chunks of knowledge from specific code-fragments. "The data warehouse sets the stage for effective data mining" in that the data in a warehouse are (often claimed to be) clean, integrated and complete [Inm96].

Data mining is related to *On-Line Analytical Processing (OLAP)*, which performs multidimensional analysis of data, but the analysis and exploration of the data are driven entirely by a human analyst in OLAP (OLAP tools are used to verify given hypotheses), whereas in data mining, exploration is computer-driven (data-mining tools explore the data themselves and generate hypotheses) [Ede97]. OLAP *data marts* are similar to data warehouses, but contain local rather than global data (e.g. departmental data rather than university-wide data). The data in OLAP data marts tend to be more summarised than in data warehouses, so data mining is 'secondary' rather than 'primary' [Inm96].

2.3.3 Case-Based Reasoning, Analogical Reasoning and Structure Mapping

Case-based reasoning (CBR) [Kol93] is a reasoning paradigm in which new problems are solved by analogy with old ones. A detailed review of CBR can be found in Chapter 3 of [Bro99].

A CBR system has a *case library*, in which each *case* represents a problem-solving experience. Each case consists of:

- a *problem* (e.g. create a recipe for sweet-and-sour chicken and peas – a typical example from CHEF (Section 2.1 of [Kol93]), a case-based planner in the domain of recipe-creation);
- a *solution* to the problem (bone the chicken; shell the peas; stir-fry the chicken and peas together using a wok);
- any more information that will enable the reasoner to use the case for the goal it is intended to support, such as the *outcome* of solving the problem as the case suggests.

To solve a given problem in a case-based manner, the reasoner:

- *retrieves* a case of which the current problem reminds it;
- *adapts* the solution from the retrieved case to solve the current problem;
- may *store* the current problem and its solution as a case in the case library – CBR learns from experience.

For example, suppose that the given problem is to create a recipe for stir-fry beef and broccoli, and suppose that the above case is retrieved (being the most usefully similar case to the given problem in the case library). CHEF infers that chicken corresponds to beef (because they are both meat – CBR needs *semantic* *knowledge* in addition to the *episodic knowledge* embodied in the case library), and that peas correspond to broccoli (because they are both vegetables). CHEF adapts the recipe by:

- substituting beef for chicken, and broccoli for beef (*reinstantiation*);
- replacing "shell the broccoli" with "chop the broccoli into chunks" (using *object critics* – another example of semantic knowledge);
- "stir-fry the beef and broccoli together using a wok" causes the broccoli to go soggy, so (after simulating the recipe – again using semantic knowledge) CHEF *repairs* the recipe so that the beef and the broccoli are cooked separately.

CBR relates to the subject matter of this section (2.3) in the way that it compares the problem to be solved with the problem part of the case, inferring correspondences between objects in the current situation and objects in the case. The comparison in CBR differs from the comparison in comparative code understanding (CCU) in the following key ways:

- In CBR, the situation must match the case, whereas CCU entails comparing two 'situations', which are 'equal in status'.
- CBR seeks the case which the situation best matches – the case which best solves the problem. The traditional approach is to compute a similarity score between the situation and each case, selecting the case with the highest similarity score; *adaptation-guided retrieval* [SK98] selects the case which is the most easily adaptable to solve the current problem. CCU, on the other hand, finds the differences between two codes, reacting to them accordingly (the CCUS tries to assess semantic equivalence (Chapters 5 and 6) when it encounters a difference, and may have to retract correspondence between two objects which have a vital difference – Section 4.2.5).
- In CCU, the correspondences between two codes tend to be more concrete than in CBR, as the representations of code (Sections 2.2.2 and 3.2) are well-structured, so that corresponding objects tend to fill the same slot. CBR, on the other hand, tends to be applied to less well-defined situations, where correspondences are more difficult to infer. The CCUS uses name

equivalence and semantic equivalence as correspondence heuristics (Chapter 4), whereas CBR often has to use *common-sense reconciliation heuristics* (Section 9.2.1 of [Kol93]) to infer correspondence, based on factors such as structural similarity, functional role (e.g. using a needle to pick a lock), and common abstractions (e.g. chicken and beef are both kinds of meat).

- CCU entails combining comparison with the extraction of higher-level knowledge (Chapter 6) in order to see abstract correspondences between two codes. This is somewhat related to *situation assessment* in CBR (Section 10.1 of [Kol93]), where the given problem situation is *elaborated* by extracting index terms with which to retrieve similar cases – however, this process takes place *before* comparison (although it can also take place during or even after retrieval), rather than being driven by the differences encountered during comparison.

- CBR also relates to combining comparison with extraction in the way that it extracts a solution from the problem situation if it matches the case – but the comparison is part of the process of extraction rather than interleaved with it; moreover the extraction is triggered by successful matching rather than by differences.

CBR has been applied to software reuse [SC92, MSSM95, FCGCGAHY96, TWP98, GB99], which actually entails code comparison in that specifications are matched to the problem parts of cases, whose solution parts contain implementation code to extract if the match succeeds. In fact, software reuse is so much in the spirit of CBR that it can be considered as *covert CBR* (work that is not claimed to be CBR, but could reasonably be claimed to be) [BF99]. The cliché-based approaches to code understanding and generation in particular (Section 2.1.1) have much affinity with CBR, as indeed does the use of cases in the CCUS (Section 1.7).

CBR also relates to *rule-based reasoning* (Chapter 7 of [Win92]; Chapter 6 of [RK91]; Chapter 4 of [FdK93]), where assertions such as `man(andrew)` are matched to the antecedents of rules such as `IF man(?x) THEN mortal(?x)`, inferring conclusions (`mortal(andrew)`) if the match succeeds. However, rules are typically much smaller than cases, they must be matched exactly whereas cases can be matched partially and then adapted, and many rules tend to be chained together in the course of problem-solving whereas cases tend to approximate the whole solution at once (Section 3.4.1 of [Kol93]).

Analogical reasoning [FFG89] is like CBR except that it uses inter-domain analogies, whereas CBR is regarded as being restricted to intra-domain analogies [AP94]. Inter-domain analogies make inferring correspondences an even finer art than it is in CBR; in fact one of the major subtasks of analogical reasoning is *analogical mapping*, which infers correspondences between two analogous situations (e.g. water flow and heat flow).

To explore Gentner's *structure-mapping* theory of analogy, the Structure-Mapping Engine (SME) was built [FFG89]. SME "constructs all consistent ways to interpret a potential analogy".

- SME prefers inferring correspondences between relationships to inferring correspondences between individual objects, which can often be misleading in inter-domain analogies. For example, in the analogy between water flow and heat flow, water corresponds to heat – not to hot water. This is seen by inferring correspondence between relationships: "water flows to a place of lower pressure" corresponds to "heat flows to a place of lower temperature."

- SME tries to infer correspondences between as large a structure of relations as possible (as above), rather than preferring correspondences between isolated objects and between relationships that are not part of a larger relational structure (e.g. "the diameter of the beaker is greater than the diameter of the vial"). This is known as the *systematicity principle*.

- The correspondences inferred by SME are one-to-one and *structurally consistent*: if two predicates correspond then each of their corresponding arguments must correspond. The correspondences inferred by the CCUS tend to be one-to-one too, although polygamous correspondences are possible (Section 4.1.2); the recursive nature of the CCUS comparison algorithm (Section 3.3) is conducive to structural consistency.

- Structure-mapping evaluates the quality of the match using structural criteria: the number of similarities and differences, the degree of structural similarity, and what new knowledge the analogy provides.

Section 2.3 has focused on comparison; Section 2.4 reviews some related topics which (could) use comparison.

2.4 Schema Mapping and Data Translation

This section reviews literature on data translation (Section 2.4.1) and schema-to-schema mapping (Section 2.4.2), which is the main application context that motivates the comparative code understanding of information models (Section 1.1). The related topics of information integration and mediation are discussed in Sections 2.4.3 and 2.4.4.

2.4.1 Data Translation

This section considers systems that convert data from one format to another, given declarative specifications of the mapping rather than an imperative implementation of the mapping. Data translation allows data interchange between different applications, enabling them to interoperate – a single standard format for all data is unreasonable [LTV96]. Writing data-translation programs by hand is tedious, resulting in code that is not easy to modify if the representations change, does not automatically support the reverse translation, and is not reusable for translations with the same source or target format [LTV96] – these problems can be rectified by generating the translation program automatically, given a specification of the source format, the target format, and the mapping between them.

The Integrated Chameleon Architecture (ICA) [MBO93] is an elegant general solution to the data-translation problem, applied specifically to text-formatters (ICA was used to develop a LaTeX-to-Troff translator). ICA automatically generates translation code, although it has to be provided with a high-level specification of the translator, rather than being able to infer correspondences between specifications of the two data-formats automatically.

There are two approaches to translator design [MBO93]:

- the *string-matching* approach: the translator replaces a given source tag with a given target tag (cf. textual comparison – Section 2.2.1);

- the *structure-oriented* approach: the translator also accommodates rules about tag ordering or occurrence (cf. structural comparison – Section 2.2.2). This is the approach adopted by ICA.

There are two data-translation models [MBO93]:

- *pairwise translation*: a separate translator is used for every pair of data-formats, so $n(n-1)$ translators are needed for n data-formats;
- *intermediate-form translation*: data are translated from the source format to an intermediate format that covers all the data-formats, and from the intermediate format to the target format, so $2n$ translators are needed for n data-formats (from each of the n data-formats to the intermediate format, and from the intermediate format to each of the n data-formats). This is the model adopted by ICA.

The distinction between pairwise translation and intermediate-form translation is analogous to the distinction between equational reasoning and extract-and-compare (Section 6.1).

ALCHEMIST [LTV95, LTV96] is a general-purpose transformation generator that has been applied to such examples as transformations between text-formats, transforming from the ER model to the relational model (cf. Section 2.1), and generating C++ code from a knowledge-acquisition tool. ALCHEMIST allows an abstract specification of the transformation through a window-based interface, and automatically generates the transformation code – so the user still has to specify the mapping, albeit at a high level.

In ALCHEMIST, an individual transformation is called a *spell*. A spell is generated in three steps:

1. *Specification*: the user specifies grammars for the source and target representations, and specifies a mapping between these grammars (i.e. a set of correspondences between the grammars' symbols and between the grammars' productions).
2. *Code generation*: from the specifications of the grammars and of the mapping, ALCHEMIST generates C++ code for a file-to-object parser, an object-to-object mapper, and an object-to-file writer.
3. *Code compilation*: ALCHEMIST calls a C++ compiler to compile the codes generated in Step 2, linking them with a user interface and object-representation code.

The main shortcoming of ICA and of ALCHEMIST is that they have to be provided with the mappings between data-formats by humans, rather than being able to infer the mappings automatically from the specifications of the data-formats (or even from examples of data in both formats). Comparing representations and inferring correspondences automatically is one of the subcontributions of this thesis (Chapter 4). This issue of whether to infer correspondences automatically or to provide them as input is discussed in Section 4.1.1.

2.4.2 Schema-to-Schema Mapping

Schema-to-schema mapping (Section 1.1) is data translation in the case of translating instances between two data models (e.g. database schemata or EXPRESS models).

Mappings between EXPRESS models can be expressed in special mapping languages for EXPRESS, such as EXPRESS-M [Bai95], EXPRESS-X [Lab96] or EXPRESS-C [Sta94] – see Section 4.1.1, which includes a detailed discussion and examples of EXPRESS-M. See [Has96] for a detailed comparison of EXPRESS-M and EXPRESS-C. These are languages for humans to specify the mapping – once again, there is no automatic deduction of mappings.

Although EXPRESS-M and EXPRESS-C are very high-level languages, they are still very procedural in nature – [BFM95] presents a set of generic, very declarative operators for schema-to-schema mapping (see Section 4.1.1). These operators still have to be specified by humans, although [BFM95] does identify "semi-automation of the specification of mappings" as future work, which would imply schema comparison.

The COMBI integration framework [KS96] supports interoperability between different engineering data models, and product-data exchange between heterogeneous design tools. COMBI achieves interoperability in two ways:

- *Schema mapping between EXPRESS models*: COMBI has a special-purpose schema-mapping language called CSML (COMBI Schema-Mapping Language), since existing EXPRESS mapping languages such as EXPRESS-M, EXPRESS-V and VML do not meet the requirements for COMBI (the existing mapping languages were in experimental phases themselves at the time, did not provide programming-language bindings, and are not suitable for the specific integration situations targeted by COMBI).

- *Object matching* tracks modifications to the data-structures resulting from the mapping operations, since data in one repository may continue to change after being mapped to another repository. Objects may need to be reclassified, so *matching rules* are used to match instances to classes.

The CCUS infers correspondences between two EXPRESS models (Chapter 4), which could be used to perform schema-to-schema mapping provided that the correspondences are expressed concretely enough (Section 4.1.2).

Pun's method for deducing mappings between knowledge-based objects and application data-structures [Pun91] (Section 2.2.2) may be adaptable to deducing mappings between two schemata, but to do the job properly it must consider semantics in order to ensure that integrity constraints will not be violated (cf. Section 1.3).

Dearnley and Smith [DS99] present a method for deducing mappings between schemata, which is discussed in Section 4.2.6.5. An application of the deduced mappings is to transform SQL queries that are expressed in terms of the global schema of a federated database system to the local schemata so that they can actually be answered.

2.4.3 Information Integration

Infomaster [GKD97] is "an *information integration* system that provides integrated access to multiple distributed heterogeneous information-sources on the Internet, thus giving the illusion of a centralized, homogeneous information system." Infomaster is a "virtual data warehouse" (cf. Section 2.3.2).

To resolve differences between multiple data-sources, Infomaster performs translation to harmonise the heterogeneities (cf. Section 1.1) among these sources:

- *notational heterogeneity*: different databases, different languages (e.g. one source may be a Sybase database using SQL (Structured Query Language); another may be an Object store using OQL (Object Query Language));
- *conceptual heterogeneity*: differences in the structure of the schemata, or differences in vocabulary (e.g. synonyms and homonyms – cf. Section 4.2.4).

Infomaster determines which sources contain the information necessary to answer a query efficiently, designs a strategy for answering the query, and performs translations to convert source information to a global virtual database schema. Infomaster uses rules and constraints to describe information-sources and translations among them. Translation rules (written by humans) describe how each source schema relates to the global schema – again, correspondences are provided rather than inferred.

InfoSleuth [JBB+97] performs semantic integration of heterogeneous information-sources too. Each information-source is represented as an *agent*, which advertises its capabilities. *Information brokers* semantically match specified information-needs with the currently available information-sources.

For example, an advertisement for a healthcare information-source might advertise that it pertains to patients aged 40 to 75, that it uses SQL as its query language and HTTP as its transfer protocol (along with its Internet address, of course). An information-need might ask for a source that pertains to patients over 65 and uses SQL. Semantic matching of the information-need to the information-source thus entails assessing semantic equivalence and semantic strength (cf. Chapter 5).

The capabilities of information-sources, and the information-needs, are both specified in terms of the concepts in a domain model (ontology). InfoSleuth specifies a *common ontology* for a domain, and *local mappings* from the individual database schemata of the information-sources to the common ontology.

2.4.4 Mediation

Mediation is a topic strongly related to information integration. A *mediator* provides integrated access, and database-like query capabilities, to information distributed over heterogeneous sources [AK00]. Each source has a local *source model* (of the contents of that source), and there is one global *domain model* (of the application domain). A mediator translates queries specified in terms of the domain model so that they are written in terms of the source models – transparently to the user, who only sees things in terms of the domain model. To perform this translation, the mediator uses *integration statements*, which relate the terms in the domain model to the terms in the source models.

There are two approaches to specifying the integration statements (Section 2.1 of [AK00]):

- *Local-as-view*: the integration statements define the terms of the source models in terms of the domain model. This makes it easier to add sources, as the new integration statements do not interact with the existing ones.

- *Global-as-view*: the integration statements define the terms of the domain model in terms of the source models. This allows more efficient query-processing, because rewriting queries from terms of the domain model to terms of the source models entails straightforward substitution (as opposed to having to look up integration statements whose definitions contain a given domain term as in local-as-view – a problem which can be obviated by precompiling a set of global-as-view inverse formulae from the human-written local-as-view integration statements).

One problem to be solved in mediation is *query planning*: how to generate a cost-efficient plan that computes a user query from the relevant information-sources. One solution to this problem is *planning by rewriting* [AK00]: transforming an easy-to-generate but possibly suboptimal initial plan into a high-quality plan. Query rewriting entails substituting terms using the integration statements – in order to transform the queries from the domain model to the source models – and optimising queries by, for instance, pushing complex queries into subexpressions so that they can be evaluated remotely, and exploiting commutativity, associativity and distributivity. Semantic differences among the sources are reconciled using the integration statements.

A possible application of comparative code understanding would be to derive the transformations from queries on the domain model to queries on the source models. Using comparative code understanding, the integration statements could be derived automatically rather than being written by humans.

Context Interchange [GBMS99] is a strategy for mediated data access in which semantic conflicts are detected and reconciled through the comparison of context axioms. *Context* refers to the way that the data in an information-source should be interpreted – for example, whether a temperature value should be interpreted as being in degrees Celsius or degrees Fahrenheit.

In the Context Interchange strategy, query rewriting is facilitated by:

- *Context axioms*: Each source is associated with a context, which has a set of *context axioms* such as "a temperature value in this context is in degrees Celsius"; context axioms also include conversion functions for transforming values between different contexts. For example, suppose that in the context of the domain model, temperature is in degrees Fahrenheit, but that in the context of a particular source model, temperature is in degrees Celsius. Then degrees Fahrenheit must be converted to degrees Celsius when rewriting queries from the domain model to that source model.

- *Elevation axioms*: Each source model has a set of elevation axioms, which are the mappings from the attributes in that source model to the semantic types of the domain model (e.g. each attribute representing a temperature is mapped to a semantic object of type `temperature`). Elevation axioms also include the integrity constraints in each source model, which are used to optimise (simplify) queries.

2.5 Conclusion

This chapter has reviewed the literature in fields related to comparative code understanding:

- Section 2.1 reviewed the field of code understanding, including various forms of knowledge that can be extracted from program code in particular. Section 2.1.1 looked in some detail at cliché-based code understanding, which is the closest approach to the CUS's method of extracting HLCs from EXPRESS models (Appendix B), and Section 2.1.2 introduced slicing.

- Section 2.2 reviewed methods of code comparison, categorising them into textual (Section 2.2.1), structural (Section 2.2.2) and semantic (Section 2.2.3) methods, including a discussion of program integration. Schema comparison was discussed in Section 2.2.4, and comparing program-executions in Section 2.2.5.

- Section 2.3 reviewed comparison methods that do not pertain specifically to code. Section 2.3.1 discussed the basic topic of comparing hierarchically-structured data, Section 2.3.2 discussed the related topic of data warehousing, and Section 2.3.3 discussed case-based reasoning, analogical reasoning and structure mapping, with the emphasis on matching and inferring correspondences.

- Section 2.4 reviewed the literature on data translation (Section 2.4.1), schema-to-schema mapping (Section 2.4.2), information integration (Section 2.4.3) and mediation (Section 2.4.4).

The contribution of this chapter is that it has imposed a new structure on the existing fields of code understanding and comparison, as this research aspires to

System/method	Reference	IC	SE/S	C & E	SE/SC	Section
`diff`	[dif]	yes				2.2.1
Talcott	[Tal98]		yes			5.2.3.4
Chaudhuri+Vardi	[CV94]		yes			5.2.2.5
Context Interchange	[GBMS99]		yes			2.4.4
InfoSleuth	[JBB+97]		yes			2.4.3
Schema Comparison	[Pro]	yes				2.2.4
Labio+García-M.	[LGM96]	yes				2.3.2
MH-DIFF	[CGM97]	yes				2.3.1
TreeDiff	[WSC+97]	yes				2.3.1
Pun	[Pun91]	yes				2.2.2
SME	[FFG89]	yes				2.3.3
CBR	[Kol93]	yes				2.3.3
Dearnley+Smith	[DS99]	yes				4.2.6.5
Türker	[Tür99]		yes			5.2.2.1
Binkley et al.	[BHR95]	yes	yes			2.2.3
Zaremski+Wing	[ZW97]	yes	yes			5.2.2.4
Hakkarainen	[Hak98]	yes		yes		2.2.4
CCUS	this thesis	yes	yes	yes	yes	

Table 2.1: A comparison of comparison methods.

establish comparative code understanding as a field in its own right. Comparative code understanding brings together ideas that were previously unrelated. Table 2.1 summarises the literature on comparison by identifying which of the subcontributions of this thesis each method addresses:

- IC: inferring correspondences (Chapter 4);
- SE/S: assessing semantic equivalence and semantic strength (Chapter 5);
- C & E: combining comparison with the extraction of higher-level knowledge (Chapter 6);
- SE/SC: selective extraction and selective comparison (Chapter 7).

Remaining literature which pertains specifically to one of the four subcontributions will be reviewed in Chapters 4 to 7 where it is relevant. First, however, the Comparative Code-Understanding System for EXPRESS models has to be introduced: Chapter 3 gives an overview of the CCUS, establishing a framework on which to hang the subcontributions.

Chapter 3

The Comparative Code-Understanding System

This chapter presents an overview of the *Comparative Code-Understanding System (CCUS)*. The CCUS compares two given EXPRESS models with the particular aim of comparing the semantics of the constraints in the models (Section 1.2).

The chapter begins with an architectural overview of the system (Section 3.1). Section 3.2 then introduces a frame-based representation of EXPRESS models, which is a prerequisite to understanding the behaviour of the CCUS. Section 3.3 explains the basic behaviour of the CCUS, presenting a walk-through of a simple run of the CCUS on a pair of simple EXPRESS models. Section 3.4 extends Section 3.3 by discussing the aggregation-matching policies used in the CCUS.

This chapter relates to the primary contribution of this thesis, *comparative code understanding*. There are four subcontributions, presented in subsequent chapters:

- inferring correspondences between two codes (Chapter 4);
- assessing the semantic equivalence of two constraints (Chapter 5);
- combining comparison with the extraction of higher-level knowledge (Chapter 6);
- selective extraction and selective comparison (Chapter 7).

This chapter establishes an overall framework into which these subcontributions fit.

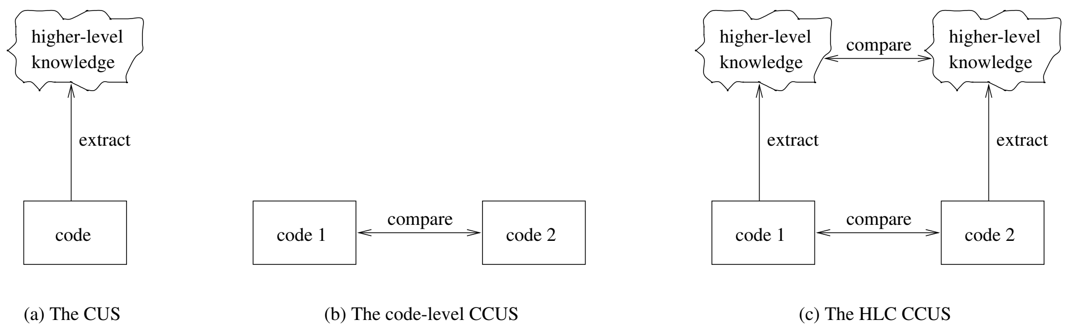


Figure 3.1: A top-level 'architecture' for comparative code understanding.

3.1 Architectural Overview

Figure 3.1 depicts a high-level view of the CCUS, which combines a system for extracting higher-level knowledge from code with a system for comparing two codes. While the CCUS applies to EXPRESS models as a particular kind of code, Figure 3.1 applies to comparative code understanding in general, not just of EXPRESS models in particular.

The *Code-Understanding System* (CUS, Figure 3.1(a)) extracts *Higher-Level Constraints* (HLCs) from a **single** EXPRESS model. The CUS was first presented in [Bro99], and a description is given in Appendix B.

An HLC is a small chunk of higher-level knowledge which characterises the semantics of a constraint at a higher level of abstraction, so that semantically equivalent constraints which are written differently can be seen to be equivalent (e.g. `(age >= 0) AND (age < 130)` can be seen to be equivalent to {`0 <= age < 130`} by extracting an HLC of type `range` from both of these expressions, and comparing the HLCs). HLCs will be discussed at greater length in Chapter 6.

The CUS extracts HLCs using *extraction cases*. Each extraction case consists of:

- a *problem part*: a template to which code-fragments from an input model are matched;
- a *solution part*: an HLC (or HLCs) to extract from a code-fragment if it does match the problem part.

The CUS closely resembles the cliché-based code-understanding systems reviewed in Section 2.1.1.

The basic *code-level Comparative Code-Understanding System* (code-level CCUS, Figure 3.1(b)) compares two EXPRESS models at the code level, inferring correspondences between corresponding parts of the models (Chapter 4), and finding differences. It can reason about the semantic equivalence and relative semantic strength of two expressions (Chapter 5), but the code-level CCUS does not extract HLCs.

The *HLC-extracting CCUS* (HLC CCUS, Figure 3.1(c)) combines comparison with extraction of higher-level knowledge. The HLC CCUS is the subject of Chapter 6. In a nutshell, when the CCUS encounters a difference that it cannot resolve at the current abstraction level, it attempts to extract HLCs from the vicinity of the difference in both models, and then compares the HLCs that it has extracted. If the HLCs are assessed to be equivalent, it then infers correspondences not only between the HLCs, but also between what those HLCs cover at a lower abstraction level (so-called *abstract correspondences*). This method of combining comparison with extraction is justified, with a discussion of alternative methods, in Section 6.2.

Selective extraction and comparison (Chapter 7) come into play in the HLC CCUS:

- *Selective extraction* means that the only HLCs that are extracted are those that are required to resolve differences at the current abstraction level, whereas the CUS (which operates on a single model) extracts all the HLCs that it knows how to extract (since it has no purpose to guide it other than extracting HLCs for their own sake).

- *Selective comparison* means that it is unnecessary to compare every HLC extracted from one model with every HLC extracted from the other model – if certain assumptions are made, then the number of HLC-comparisons can be reduced.

The rest of this chapter explains in detail the behaviour of the main CCUS algorithm, with the aid of a walk-through on two simple EXPRESS models (Section 3.3). Before the behaviour of the CCUS can be understood, however, its internal frame-based representation of EXPRESS models needs to be introduced, and that is the role of the next section.

```
INSTANCE person#1;
  name = "Andrew Broad";
  age = 25;
END_INSTANCE;
```

Figure 3.2: A frame.

3.2 Representation of EXPRESS Models

Before an EXPRESS model is processed by the CUS (Appendix B), or a pair of EXPRESS models is processed by the CCUS (Section 3.3), they are converted into a frame-based representation: each model is represented as a network of *frames*, each frame representing a syntactic component of the model (an entity, an attribute, an expression, and so on – a model is a hierarchical set of components).

The frame-representation of EXPRESS models is presented in Section 3.2.3, following an introduction to the frame system in Section 3.2.1, and a discussion in Section 3.2.2 of abstract syntax trees, the parsing of EXPRESS models and their conversion to the frame-representation.

3.2.1 The Frame System

This subsection introduces the frame system, which has already been described in detail in Chapter 5 of [Bro99], so this subsection aims to give the reader enough introduction to frames to understand the details of the CCUS which follow.

A *frame* is a collection of slot-fillers (attributes and their values) to represent an object. Figure 3.2 gives a simple example of a frame.

Every frame is an instance of a *frame-class*, which represents the type of object it is, and specifies the *slots* (attributes) it can have. For example, the frame in Figure 3.2 is the first instance of the frame-class `person` (defined in Figure 3.3), which has the slots `name` and `age`. Each frame is uniquely identified by its frame-class and its *instance-number* within that frame-class (e.g. `person#1`).

Frame-classes and slots are themselves data (as opposed to being hard-wired into the program code, they are defined in a *frame-class model* written in a *frame-class language*), so the frame system provides a data-driven, object-centric environment upon which generic algorithms (namely for extraction and comparison) can be built. So the EXPRESS model in Figure 3.3 is used as the frame-class model for the frame in Figure 3.2.

```
SCHEMA test;
  ENTITY person;
    name : STRING;
    age : OPTIONAL INTEGER;
  WHERE
    age >= 0;
  END_ENTITY;
END_SCHEMA;
```

Figure 3.3: A simple EXPRESS model.

It was decided to use EXPRESS itself as the frame-class language, defining frame-classes as entities and slots as attributes (Section 5.2 of [Bro99]), because:

- EXPRESS provides all the features required of a frame-class language for the purposes of code understanding and comparative code understanding.
- The CUS and CCUS need to read EXPRESS anyway (for the input models), so EXPRESS might as well be used as the frame-class language, since it is more than adequate, and the same parser can be used. This has the advantage of uniformity as well as convenience.

Because the frame-class model for the frame-representation of EXPRESS models (Appendix D) is itself written in EXPRESS, it constitutes an EXPRESS *metamodel*, i.e. an EXPRESS model of EXPRESS (cf. [Haz94]). This means that every frame-class is itself represented by a frame (of frame-class `entity_type`), which is elegant, albeit not exploited in the current system.

Since EXPRESS does not itself include instances,[1] a separate language is required in which to write the frames themselves (instances of the frame-classes, i.e. instances of EXPRESS entities). The Author has invented the *Frame-Instance Language* (FIL) for this purpose, which is described in complete detail in Section 5.3 of [Bro99]. The frame in Figure 3.2 is written in FIL.

FIL was invented instead of using the existing instance language for EXPRESS, the ISO 10303-21 Physical File Format (*Part 21*) (Section 2.3 of [Kas96]), because:

- Whereas FIL represents the slot-fillers of a frame as an unordered set of attribute-value pairs, Part 21 represents them as an ordered list of values

[1] EXPRESS-I [Wil95] includes instances, but is not intended to support the efficient implementation of instances.

without the attribute names, which is less human-readable, and also gives rise to complicated and contentious ordering rules where inheritance is involved.

- Part 21 does not support inverse attributes (Appendix A, Section A.5); FIL does.

- Part 21 does not support the extensions from FIL to FIL-C, the language in which cases[2] are written (Appendix B).

FIL is a language for the external representation of frames (as text files). Frames can be either:

- read in from FIL files[3] (when reading the cases, which are written by humans in a variant of FIL called FIL-C);

- generated internally (when building the frame-representation of an EXPRESS model, and when extracting HLCs).

At the end of a run of the CUS or CCUS, the user can elect to have all the frames in memory dumped as FIL files (or just the frames pertaining to the input models and their HLCs, suppressing the frames for the frame-class model and the cases). The FIL files are dumped in HTML format [MK97] to facilitate browsing (with hypertext links for the frame-references). There is thus a distinction between *input FIL files* and *output FIL files*.

Although the slots in frames can be filled in any order, they are iterated over in the order in which they were filled,[4] which ensures that the CUS and the CCUS are *deterministic* (always give the same output for the same input).

[2]Extraction cases (Section 3.1) and other cases, namely *navigation cases* (Section 6.3.1), *semantic-equivalence/strength cases* (Section 5.3), *subexpression cases* (Section 4.3) and *pretty-print cases* (Appendix E, Section E.5) – recall the case-based reasoning origins of the CUS (Section 1.7).

[3]Note that each frame is assigned an *internal instance-number* when it is created in memory, which is not necessarily the same as its *external instance-number* in the case that it is read from an FIL file.

[4]This is achieved by representing frames as *ordered hash-tables*: the keys are slots, the values are fillers, and each frame maintains an ordered list of its slots as they are filled.

3.2.2 Abstract Syntax Trees and Parsing the EXPRESS Models

Representing an EXPRESS model as a network of frames corresponds to the traditional notion of using *abstract syntax trees* (Section 5.2 of [ASU86]) as an internal representation of a code, typically to facilitate compilation. Such a representation has the following advantages:

- An internal representation means that the code need not be processed sequentially, but rather according to its structure, with the appropriate cross-references. This facilitates navigating the code for extraction and comparison, which entails going in and out of substructures rather than walking the code from beginning to end (Section 2.2.2).

- An abstract syntax tree of a code abstracts away unnecessary detail such as the concrete syntax, which could actually hinder a comparative code-understanding system if it were left in.

- Converting to a frame-representation provides opportunities to massage the code into a more semantics-oriented form to facilitate comparison (cf. the discussion of standard forms in Section 5.2.3.1). In particular, the CCUS:

  - represents the addends of an addition as a bag[5] (e.g. both $a + b$ and $b+a$ can be represented as a bag $\{a, b\}$), since addition is commutative;
  - similarly represents the multiplicands of a multiplication as a bag;
  - represents the conjuncts of a conjunction (AND), or the disjuncts of an inclusive disjunction (OR), as a set (duplicate elements do not count since $c \wedge c \equiv c$ and $d \vee d \equiv d$);
  - similarly represents the expressions in an equation as a set;
  - represents a subtraction as an addition whose second operand is negated, i.e. $a - b$ has the same representation as $a + -b$;
  - represents a division as a multiplication whose second operand is reciprocated, i.e. c/d has the same representation as $c(1/d)$;
  - represents a comparator so as to normalise out which way round it is written (e.g. $a > b \equiv b < a$) (see Section 5.3).

[5] A *bag* is an unordered aggregation which is the same as a set except that duplicate elements are significant.

- A frame-representation can be annotated with information beyond the code itself, namely the higher-level knowledge extracted, and the correspondences inferred between two codes.

The CUS and the CCUS actually read EXPRESS models using an existing parser/API (Application Programming Interface) called Jex [KFWW01]. Jex is written in the object-oriented programming language Java[6] [Gra97], as are the frame system, the CUS and the CCUS.

Jex converts an EXPRESS model to a collection of Java objects, but these are not suitable for use by the CUS and CCUS directly. In contrast to the semantics-oriented representation discussed above, the Jex representation of an EXPRESS model is extremely syntax-oriented, resembling the structure of a concrete parse tree (for example, an arithmetic expression is represented as an abstruse network of expressions, simple expressions, terms, factors, and so on).

So the Jex objects are converted to frames for processing by the CUS/CCUS.[7] Frames are themselves represented as Java objects, but as opposed to having a Java class for each frame-class (like Jex), each frame is represented as an object of type `Frame`, and each frame-class is represented as an object of type `FrameClass`. Recall that the frame-classes are defined in the frame-class model as opposed to being hard-wired into the Java program.

3.2.3 The Frame-Representation of EXPRESS Models

The CUS builds a frame-network to represent an input EXPRESS model, and the CCUS does so to represent two such models. The FIL frame-representation of the EXPRESS model in Figure 3.3[8] is shown below (Figure 3.4 is a diagram of the frames and their relationships to aid the reader in visualising it).

```
INSTANCE model_frame#3;
  corresponds_to = {model_frame#4};
  schemata = {schema_frame#4};
END_INSTANCE;

INSTANCE schema_frame#4;
  corresponds_to = {schema_frame#5};
  name = "test";
```

[6] Java is a trademark of Sun Microsystems, Inc.

[7] Using a variant of FIL called FIL-J [Bro] which specifies, in a data-driven manner, the mapping from each Jex object to a frame (although some Jex-to-frame mappings have to be hard-wired into the program due to the limited expressive power of FIL-J).

[8] Figure 3.3 also happened to be used as a frame-class model in Section 3.2.1, but here it is used as an input model.

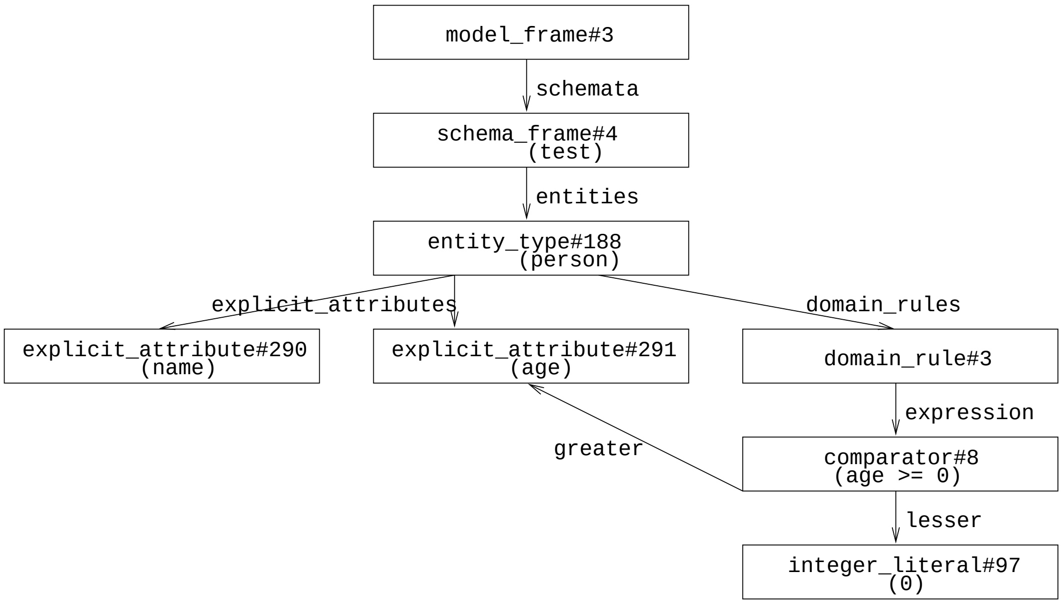


Figure 3.4: Diagram of the frame-representation of the model in Figure 3.3.

```
  entities = {entity_type#188};
  the_model = model_frame#3; (* set indirectly *)
END_INSTANCE;

INSTANCE entity_type#188;
  corresponds_to = {entity_type#189};
  name = "person";
  domain_rules = {domain_rule#3};
  is_abstract = false;
  explicit_attributes = {explicit_attribute#290, explicit_attribute#291};
  the_schema = schema_frame#4; (* set indirectly *)
END_INSTANCE;

INSTANCE explicit_attribute#290;
  corresponds_to = {explicit_attribute#292};
  tipe = simple_type(string_type);
  is_variable = true;
  name = "name";
  is_optional = false;
  the_entity = entity_type#188; (* set indirectly *)
END_INSTANCE;

INSTANCE explicit_attribute#291;
  corresponds_to = {explicit_attribute#293};
  tipe = simple_type(integer_type);
  is_variable = true;
  name = "age";
  is_optional = true;
  the_entity = entity_type#188; (* set indirectly *)
END_INSTANCE;

INSTANCE domain_rule#3;
  corresponds_to = {correspondence#2};
  expression = comparator#8;
  the_type = entity_type#188; (* set indirectly *)
```

```
END_INSTANCE;

INSTANCE comparator#8;
  description = "age >= 0";
  corresponds_to = {correspondence#3};
  tipe = simple_type(logical_type);
  greater = explicit_attribute#291;
  lesser = integer_literal#97;
  inclusive = inclusive_or_exclusive(inclusive);
  discrete = discrete_or_continuous(discrete);
  which_way_round = way_round(greater_first);
END_INSTANCE;

INSTANCE integer_literal#97;
  corresponds_to = {correspondence#4};
  tipe = simple_type(integer_type);
  is_variable = false;
  literal_value = 0;
END_INSTANCE;
```

Notes on the above frame-representation:

- The frame-classes of these frames are defined in the EXPRESS frame-class model (Appendix D).
- The `corresponds_to` slots are unfilled initially – they are not filled during the building of the frame-networks, they are filled by the CCUS as it infers correspondences between the two input models: Section 3.3 will explain how the model in Figure 3.3 is compared with another model.
- Unfilled slots are simply omitted (e.g. the frame-class `entity_type` also has the slots `supertype_of`, `subtype_of`, and so on, which are unfilled in `entity_type#188`).
- The slots in a frame include slots inherited from superclasses (e.g. `entity_type#188` fills the slots `name` and `domain_rules`, which `entity_type` inherits from `named_type`).
- Inverse slots are filled automatically by the frame system (there is no need to specify their fillers in an input FIL file, although inverse slots *can* be filled directly (by specifying their fillers explicitly)). For example, in `schema_frame#4`, the inverse slot `the_model` is set indirectly to `model_frame#3` when `schemata` is set to {`schema_frame#4`} in `model_frame#3`. Inverse slots that are set indirectly are commented as such in output FIL files (comments in FIL are delimited by `(*` and `*)`).

- `simple_type(string_type)` is an example of a *defined-type slot-filler*, where the defined type (`simple_type`) is an enumeration type, and `string_type` is an enumeration symbol.

- The slot `is_variable` is used to distinguish between *variables* (different for different instances, e.g. attributes) and *constants*, so that the system understands which operand of a comparator is the constrained variable and which is the limit (see Section 5.3 for a discussion of the representation of comparators).

- Reserved words in EXPRESS cannot be used as identifiers in FIL, since the entity and attribute names must match those defined in the frame-class model. That is why "type" is spelled as "tipe".

Note that FIL is not limited in its expressive power to represent *any* code, since an EXPRESS model of any formal language could be written, and used as the frame-class model to represent any code written in that language as a frame-network. FIL representations are very long-winded compared with the code they represent, though.

This section has explained how EXPRESS models are represented internally as frame-networks. To summarise, the CUS and the CCUS use Jex to read EXPRESS models, Jex returns a collection of objects which represent an EXPRESS model (in a very syntax-oriented manner), and the CUS or CCUS converts the Jex representation of the input EXPRESS model(s) to a frame-representation (which is more semantics-oriented), upon which the CUS or CCUS operates directly:

- The CUS extracts HLCs from a frame-network that represents a single EXPRESS model;

- The CCUS compares two frame-networks representing EXPRESS models.

The next section presents the main CCUS algorithm which compares two frame-networks.

3.3 The Main Comparison Algorithm

This section explains the behaviour of the basic CCUS frame-comparison algorithm, which starts with the frame-networks representing the two EXPRESS

<pre>SCHEMA test;
  ENTITY person;
    name : STRING;
    age : OPTIONAL INTEGER;
  WHERE
    age >= 0;
  END_ENTITY;
END_SCHEMA;</pre>
	<pre>SCHEMA test;
	  ENTITY person;
	    name : STRING;
	    age : INTEGER;
	  WHERE
	    age >= 0;
	  END_ENTITY;
	END_SCHEMA;</pre>

Figure 3.5: A pair of simple EXPRESS models to compare.

models (Section 3.2) and compares them, inferring correspondences between the corresponding parts of the two models, and reporting differences. Inferring correspondences and assessing semantic equivalence are discussed in detail in Chapters 4 and 5 respectively.

Here in the main text, the behaviour of the CCUS is illustrated through worked examples with simple EXPRESS models. Pseudocode algorithms are in Appendix C, Section C.1.

The example used in this section will be comparing the EXPRESS model in Figure 3.3 (which is replicated in the left half of Figure 3.5) with the slightly modified version of it in the right half of Figure 3.5.[9] The frame-representation of this second model follows below (Figure 3.6 is a diagram of the frames and their relationships to aid the reader in visualising it).

```
INSTANCE model_frame#4;
  corresponds_to = {model_frame#3};
  schemata = {schema_frame#5};
END_INSTANCE;

INSTANCE schema_frame#5;
  corresponds_to = {schema_frame#4};
  name = "test";
  entities = {entity_type#189};
  the_model = model_frame#4; (* set indirectly *)
END_INSTANCE;

INSTANCE entity_type#189;
  corresponds_to = {entity_type#188};
  name = "person";
  domain_rules = {domain_rule#4};
  is_abstract = false;
  explicit_attributes = {explicit_attribute#292, explicit_attribute#293};
  the_schema = schema_frame#5; (* set indirectly *)
END_INSTANCE;

INSTANCE explicit_attribute#292;
  corresponds_to = {explicit_attribute#290};
```

[9]The differences from the first model are minimal, but the point of this example is to have just enough to illustrate every aspect of the behaviour of the main CCUS algorithm.

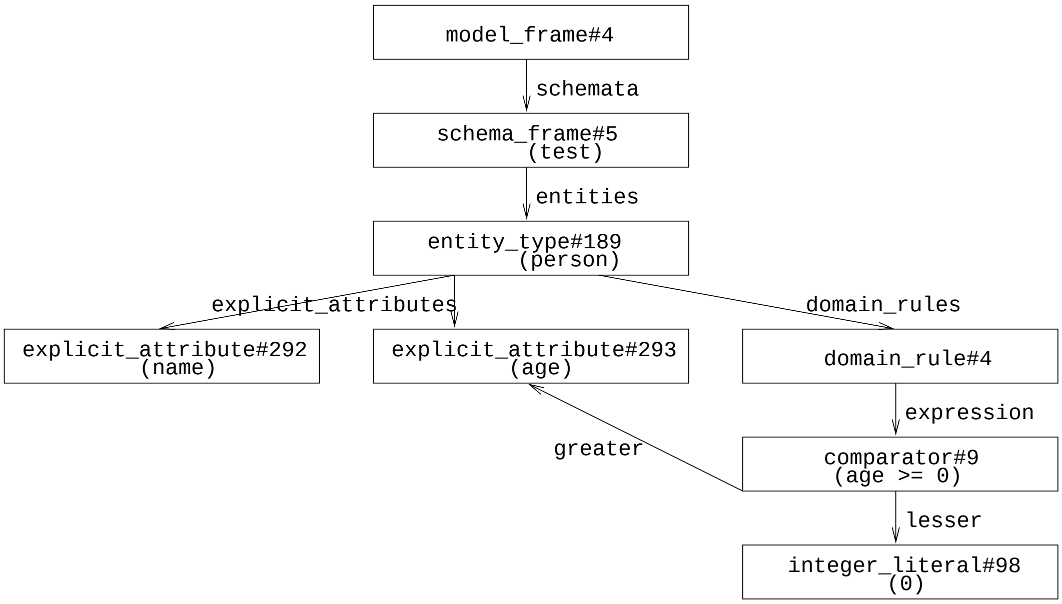


Figure 3.6: Diagram of the frame-representation of the second model in Figure 3.5.

```
  tipe = simple_type(string_type);
  is_variable = true;
  name = "name";
  is_optional = false;
  the_entity = entity_type#189; (* set indirectly *)
END_INSTANCE;

INSTANCE explicit_attribute#293;
  corresponds_to = {explicit_attribute#291};
  tipe = simple_type(integer_type);
  is_variable = true;
  name = "age";
  is_optional = false; (* <---------------------------------- THE DIFFERENCE *)
  the_entity = entity_type#189; (* set indirectly *)
END_INSTANCE;

INSTANCE domain_rule#4;
  corresponds_to = {correspondence#2};
  expression = comparator#9;
  the_type = entity_type#189; (* set indirectly *)
END_INSTANCE;

INSTANCE comparator#9;
  description = "age >= 0";
  corresponds_to = {correspondence#3};
  tipe = simple_type(logical_type);
  greater = explicit_attribute#293;
  lesser = integer_literal#98;
  inclusive = inclusive_or_exclusive(inclusive);
  discrete = discrete_or_continuous(discrete);
  which_way_round = way_round(greater_first);
END_INSTANCE;

INSTANCE integer_literal#98;
  corresponds_to = {correspondence#4};
```

```
  tipe = simple_type(integer_type);
  is_variable = false;
  literal_value = 0;
END_INSTANCE;
```

The `corresponds_to` slots are special slots filled by the CCUS whenever it infers that two frames correspond to each other. They are initially unfilled, which should be borne in mind when following the progress of the algorithm with the frame-representations given above.

The actual output of the CCUS – which reports the correspondences inferred and the differences found – for this example is shown below.

```
model_frame#3 corresponds to model_frame#4.
.schema_frame#4 (test) corresponds to schema_frame#5 (test).
..entity_type#188 (person) corresponds to entity_type#189 (person).
...explicit_attribute#290 (person.name) corresponds to
   explicit_attribute#292 (person.name).
...explicit_attribute#291 (person.age) corresponds to
   explicit_attribute#293 (person.age).
...Difference in slot "is_optional":-
...- explicit_attribute#291.is_optional == true
...- explicit_attribute#293.is_optional == false
...domain_rule#3 plausibly corresponds to domain_rule#4.
....comparator#8 (age >= 0) plausibly corresponds to comparator#9 (age >= 0).
.....integer_literal#97 (0) plausibly corresponds to integer_literal#98 (0).
.....Correspondence between integer_literal#97 and integer_literal#98
     confirmed.
....Correspondence between comparator#8 and comparator#9 confirmed.
...Correspondence between domain_rule#3 and domain_rule#4 confirmed.
```

The CCUS frame-comparison algorithm walks the two models in parallel, in a top-down manner. It starts by comparing the frames `model_frame#3` and `model_frame#4`, which are known to be the top-level frames of the models. The `matchFrames` method (Section C.1.1) starts by recording correspondence between the `model_frame` frames (i.e. it adds them to each other's `corresponds_to` slots), and then proceeds to compare the frames slot by slot.

The only slot which is filled in `model_frame#3` and `model_frame#4` is `schemata`,[10] which is filled with the sets {`schema_frame#4`} and {`schema_frame#5`} respectively. *Aggregation-matching* (for sets, bags, lists and arrays) will be discussed in Section 3.4; for now, it suffices to understand that the frames `schema_frame#4` and `schema_frame#5` are compared.

Whenever `matchFillers` (Section C.1.2) compares two frames, it must get their correspondence sanctioned before calling `matchFrames` (which presupposes

[10]`corresponds_to` does not count because it is a special slot rather than part of the models, so it is marked with `--IGNORE` in the frame-class model (Appendix D), which tells the CCUS to ignore it.

that they correspond, records the correspondence, and then compares them slot by slot). The `sanctionCorrespondence` method uses various *correspondence heuristics* (Section 4.2) to sanction (allow) or veto (forbid) the correspondence, returning one of the following:

- `CORRESPONDENCE`: The frames correspond – proceed with comparison;
- `PLAUSIBLE_CORRESPONDENCE`: The frames may or may not correspond – proceed with comparison, which may have to retract the correspondence if the comparison fails;
- `NO_CORRESPONDENCE`: The frames do not correspond – do not compare them.

`sanctionCorrespondence(schema_frame#4, schema_frame#5)` returns `CORRESPONDENCE` – the correspondence is sanctioned by the *name identicality* correspondence heuristic (Section 4.2.3) because the schemata are both named objects (i.e. the frame-class `schema_frame` has the slot `name`) and their names (i.e. "`test`") are identical. So `matchFrames` is called on `schema_frame#4` and `schema_frame#5` in `CORRESPONDENCE` mode.

In comparing `schema_frame#4` and `schema_frame#5`, the CCUS records that they correspond, and compares the slots `name` (their names are string-equal) and `entities` (it compares `entity_type#188` and `entity_type#189` after sanctioning correspondence between them – again using name identicality). It does not compare the schemata along the slot `the_model`, because that is an inverse slot that was set indirectly, and by comparing `model_frame#3` and `model_frame#4` again, the CCUS would be working back outwards (towards the top of the model) and thus repeating comparisons, instead of working inwards (towards the leaves of the model).

In the same way, `entity_type#188` is compared with `entity_type#189` (`person`), `explicit_attribute#290` with `explicit_attribute#292` (`name`), and `explicit_attribute#291` with `explicit_attribute#293` (`age`).

The only difference encountered is that `age` is an optional attribute in the first model (`is_optional = true`), and a mandatory attribute in the second model (`is_optional = false`). This is the hook to the HLC CCUS (Chapter 6), where differences trigger the extraction of HLCs.

When comparing `domain_rule#3` and `domain_rule#4`, `sanctionCorrespondence` returns `PLAUSIBLE_CORRESPONDENCE` – since they do not have the `name` slot, they cannot be inferred to correspond using name equivalence heuristics. So comparison must proceed with a *plausible correspondence* – one which may have to be retracted if the comparison fails (Section 4.2.5).

A plausible correspondence is represented as a frame of frame-class `correspondence`, with which the corresponding frames' `corresponds_to` slots are filled. This is done instead of simply filling their `corresponds_to` slots with each other, so that plausible correspondences can be tagged with extra information – namely about semantic equivalence/strength (Section 5.3). The `correspondence` frames (generated by the CCUS) referred to in the above frame-networks are:

```
INSTANCE correspondence#2;
  frame1 = domain_rule#3;
  frame2 = domain_rule#4;
  semantic_equivalence = semantic_equivalence(equivalent);
END_INSTANCE;

INSTANCE correspondence#3;
  frame1 = comparator#8;
  frame2 = comparator#9;
  semantic_equivalence = semantic_equivalence(equivalent);
END_INSTANCE;

INSTANCE correspondence#4;
  frame1 = integer_literal#97;
  frame2 = integer_literal#98;
  semantic_equivalence = semantic_equivalence(equivalent);
END_INSTANCE;
```

Similarly, plausible correspondences are also inferred between `comparator#8` and `comparator#9`, and between `integer_literal#97` and `integer_literal#98` (note that in comparing `comparator#8` and `comparator#9`, `explicit_attribute #291` matches `explicit_attribute#293` immediately, because they have already been inferred to correspond[11]). Since there are no differences, the plausible correspondences are confirmed – in the sense that they are not retracted – the correspondence heuristic which has sanctioned their correspondence is *semantic equivalence* (Section 4.2.5).

In summary:

- `model_frame#3` corresponds to `model_frame#4` because they are the top-level frames;
- `schema_frame#4` corresponds to `schema_frame#5` (`test`) by name identicality;

[11]Unless the CCUS happens to compare domain rules before attributes, in which case it would infer the attributes to correspond when it encountered the attribute-references in the domain rules.

- `entity_type#188` corresponds to `entity_type#189` (`person`) by name identicality;
- `explicit_attribute#290` corresponds to `explicit_attribute#292` (`name`) by name identicality;
- `explicit_attribute#291` corresponds to `explicit_attribute#293` (`age`) by name identicality;
- `domain_rule#3` corresponds to `domain_rule#4` (`age >= 0`) by semantic equivalence;
- `comparator#8` corresponds to `comparator#9` (`age >= 0`) by semantic equivalence;
- `integer_literal#97` corresponds to `integer_literal#98` (0) by semantic equivalence.

If, for the sake of argument, the comparison of `comparator#8` and `comparator#9` failed,[12] the correspondence between those two frames would be retracted, and so would the correspondence between `domain_rule#3` and `domain_rule#4`, since they would then fail to match along the slot `expression`, whose fillers `comparator#8` and `comparator#9` would not correspond (this is a form of *dependency-directed backtracking*, as will be discussed in Section 4.2.5). But they do correspond, so the comparison succeeds.

Note that differences do not necessarily imply failure to match, since the whole point of comparative code understanding is that there can be things which are syntactically different but semantically equivalent.

A fundamental assumption here is that two syntactically identical code-fragments cannot have different semantics. It would fool the CCUS if they could, but this problem is unlikely to occur in practice.

- A realistic example of this is that entities which describe different concepts can have the same name – Section 4.2.4 discusses how the correspondence between such homonyms could be vetoed. The only way to legislate for the possibility of homonymy would be to analyse corresponding code-fragments in depth even when they are identical at the code level, and the expense

[12] This would happen if they were semantically incomparable, as explained in Section 5.3.

of doing so would almost certainly outweigh the benefit of detecting any homonymy that might occur.

- This problem would also occur in cases of computational mapping (see Section 1.3 or 4.1.1).

When two code-fragments are syntactically different but semantically equivalent, the CCUS has two major ways to assess this:

- assessing semantic equivalence at the current abstraction level, using *semantic-equivalence/strength cases* (Section 5.3);
- extracting HLCs, comparing the HLCs to see that they are equivalent, and then inferring so-called *abstract correspondences* back at the current abstraction level (Section 6.3).

The CCUS frame-comparison algorithm is very similar to the CUS case-matching algorithm (Appendix B), but with the following major differences:

- Whereas the CCUS frame-comparison algorithm compares two codes, which are 'equal in status', in the CUS case-matching algorithm the input code-fragment must match the problem part of a case (but not vice versa). Thus the CCUS is performing bidirectional comparison, whereas the CUS is performing unidirectional matching.
- If the CUS case-matching algorithm encounters a difference (i.e. if a slot-filler specified in the problem part of a case is not matched by the corresponding slot-filler in the input), then the match fails immediately. The CCUS frame-comparison algorithm, on the other hand, is comparing two codes to find as many differences and correspondences as it can, and the only situation in which it fails a match is in plausible-correspondence mode, when a difference is encountered that cannot be resolved (i.e. semantic equivalence despite the difference cannot be shown).
- These differences are reflected in the aggregation-matching policies used by the CUS case-matching algorithm, as contrasted in Section 3.4 with those used by the CCUS frame-comparison algorithm.

3.4 Aggregation-Matching Policies

When the CCUS is comparing slot-fillers which are aggregations (vectors), it must match corresponding elements in the two aggregations. The CCUS uses different aggregation-matching policies, depending on whether the aggregations are *unordered aggregations* (sets or bags) or *ordered aggregations* (lists or arrays).

Aggregation-matching policies use *correspondence heuristics* (Section 4.2) to infer correspondences between the elements of one aggregation and the elements of the other, in the case that the elements are frames – to generalise, two elements are matched using the `matchFillers` method (described in the previous section).

3.4.1 Unordered Aggregations

The aggregation-matching policy chosen for unordered aggregations (such as the set of schemata in an EXPRESS model, the set of entities in a schema, or the set of attributes in an entity) is to attempt to match each element in the first aggregation to some element in the second aggregation, and vice versa. The algorithm is given in Appendix C, Section C.2.2.

To take toy examples with sets of integers,

- When matching {1,2,3} and {2,1,3}, it would match 1 with 1, 2 with 2, and 3 with 3, and not report any differences.
- When matching {1,2,4} and {2,3,4}, it would match 2 with 2, 4 with 4, report that 1 does not correspond to anything in the second set, that 3 does not correspond to anything in the first set, and finally that {1,2,4} and {2,3,4} are different.

Although the algorithm is, in general, required to compare every element in the first aggregation with every element in the second aggregation and vice versa, for some element types it is desirable to cut the match after finding the first pair of matching elements. These are types where correspondences are known to be one-to-one, in particular the *named* objects (such as schemata, entities and attributes – schema names must be unique within a model, entity names must be unique within a schema, and attribute names must be unique within an entity). Unnamed objects (e.g. domain rules and expressions), on the other hand, can have many-to-many correspondences between them. Therefore, `matchUnorderedVectors` is passed the element type as an argument, and if it is a frame-class with the slot `name`, then for each element in one aggregation, once it has found one matching element in the other aggregation, it stops looking.

A naïve algorithm would be to start at the beginning of the second aggregation for each element in the first. However, `matchUnorderedVectors` starts at the position in the second aggregation where it left off at the end of matching the previous element from the first aggregation (during the iteration, it wraps around to the start of the second aggregation if necessary) – a similar technique is used in Labio and García-Molina's window algorithm [LGM96] (Section 2.3.2). When it is in the mode of cutting after the first pair of matching elements, this reduces the time complexity of the algorithm from quadratic (in the worst case) to linear in the case where corresponding elements are in the same order in both aggregations. This will indeed tend to be the case, particularly when one of the two models (or code in general) is a revision (modification) of the other – there is usually no reason to change the order when code is modified, or even when it is transformed.

The unordered-aggregation-matching policy described here corresponds to Policy 1 in Section D.2.3 of [Bro99], but with an important difference:

- Policy 1 is an aggregation-matching policy for *case matching*[13] in the CUS (matching a code-fragment to an extraction case), where every element in the aggregation from the case must be matched by an element in the aggregation from the retrieval prompt but *not* vice versa.

- On the other hand, the CCUS frame-comparison algorithm is comparing two models, which are 'equal in status' as opposed to one model having to match the other – bidirectional comparison as opposed to unidirectional matching.

3.4.2 Ordered Aggregations

The aggregation-matching policy chosen for ordered aggregations[14] is to walk the two aggregations in parallel, matching elements in corresponding positions. The algorithm is given in Appendix C, Section C.2.3.

To take toy examples with lists of integers,

[13] This is another example of the case-based reasoning origins of the CUS (Section 1.7).

[14] If the CCUS has to compare an ordered aggregation with an unordered one, it selects the unordered-aggregation-matching policy, as it would not make sense to assume that the elements in the unordered one are ordered.

- When matching [1,2,3] and [1,2,3], it would match 1 with 1, 2 with 2, and 3 with 3, and not report a difference.
- When matching [1,2,3] and [1,3,2], it would match 1 with 1, report that 2 and 3 are different, that 3 and 2 are different, and that [1,2,3] and [1,3,2] are different.

When devising the ordered-aggregation-matching policy, two ideas were considered:

1. Idea 1: Require the aggregations to be the same size, and match only elements in corresponding positions. For example, matching [a, b, c] to [a, b, c, d] would fail immediately, without even looking at the elements.
2. Idea 2: Match and find correspondences as for unordered aggregations, but report differences if the elements are in different orders. For example, given [a, b, c, d] and [d, a, c, b], it would infer four correspondences but report three differences.

The final algorithm implemented is based on Idea 1, except that it still proceeds to match the elements if the sizes are different, for the sake of finding correspondences. For example, given [a, b, c] and [a, b, c, d], it would infer correspondences for a, b and c, and ignore d. This is a sensible ordered-aggregation-matching policy for comparative code understanding, because what the ordered aggregations tend to represent are function calls, where it only makes sense to infer correspondences between arguments in corresponding positions, and some languages even allow some of the last arguments to a function call to be omitted, supplying default values.

Idea 2 would be appropriate for constructs that are ordered, but for which it makes sense to infer correspondences even when some elements are missing, or in a different order. An obvious example is statements in procedural code. Another example is that enumerations in EXPRESS (and other languages, such as C – Section 6.2 of [CLL92]) order their symbols, even though they clearly tend to have set semantics in the minds of the modellers (so it would be a shame to fail to match `ENUMERATION OF (male, female)` and `ENUMERATION OF (female, male)`, for example).[15]

[15]In fact, the HLC CCUS gets around this particular problem by converting the list to a set when extracting the `choice_set` HLC.

The ordered-aggregation-matching policy described here corresponds to Policy 2 in Section D.2.3 of [Bro99], except that:

- Policy 2 returns false immediately if the two aggregations are of unequal length because, as an aggregation-matching policy for *case matching* as opposed to comparative code understanding, its sole aim is to succeed or fail the matches.

- The CCUS, on the other hand, aims to infer as many correspondences and find as many differences as possible, rather than cutting the matches immediately when they fail.

3.5 Conclusion

This chapter has introduced the comparative code-understanding system which has been implemented for comparing two EXPRESS models. Section 3.1 gave an architectural overview, Section 3.2 introduced the frame system and the frame-representation of EXPRESS models, Section 3.3 gave an overview of the behaviour of the CCUS frame-comparison algorithm, and Section 3.4 discussed the aggregation-matching policies used in the CCUS.

The primary contribution of this thesis is comparative code understanding. This chapter has established an overall framework on which to hang the four major subcontributions:

- inferring correspondences (Chapter 4);

- assessing semantic equivalence and semantic strength (Chapter 5);

- combining comparison with the extraction of higher-level knowledge (Chapter 6);

- selective extraction and comparison (Chapter 7).

Chapter 4

Inferring Correspondences

The previous chapter presented an overall framework for comparative code understanding, which entails walking two codes in parallel, inferring correspondences between components of one code and components of the other code, comparing corresponding components and assessing whether they are semantically equivalent despite any syntactic differences.

This chapter focuses on the inference of correspondences, which is one of the four subcontributions of this thesis presented in Chapters 4 to 7. When the Comparative Code-Understanding System (CCUS) first considers two objects (represented as frames, as in Section 3.2.3) for comparison, it attempts to infer a correspondence between them before it compares them slot by slot (see Section 3.3). Correspondences are sanctioned or vetoed by *correspondence heuristics*. The CCUS must first select an appropriate correspondence heuristic for the given pair of objects, and then apply it.

Section 4.3 presents a mini-contribution: how to align an expression in one code with a corresponding subexpression in the other code, enabling the CCUS to infer appropriate correspondences between expressions such as `age >= 0` and `(age >= 0) AND (age < 130)`.

4.1 Representing and Inferring Correspondences

Before two code-fragments can be compared, they must be known (or at least hypothesised) to correspond.

A *correspondence* is an assertion that an object in one code corresponds to an object in another code, meaning that it is appropriate to compare them. This does not necessarily mean that they are *equivalent* (i.e. that for all possible input values, they always evaluate to the same value as each other) – indeed, it is often necessary to compare them *in order to assess whether they are equivalent*, which means that correspondences may have to be retracted.

For example, the CCUS:

- assumes an overall correspondence between the two EXPRESS models it is given to compare;
- infers correspondences between named objects such as entities and attributes based on their names, which are often the best available indicator of whether they represent the same real-world concept and thus *truly* correspond;
- hypothesises that expressions in corresponding positions correspond, and verifies such correspondences by comparing the expressions to assess whether they are semantically equivalent (e.g. `(age >= 0) AND (age < 130)` is semantically equivalent to {`0 <= age < 130`}). Semantic equivalence is the subject of Chapter 5.

There are several other areas of computer science that entail inferring correspondences. For example:

- Schema comparison [Hak98] entails inferring correspondences between two schemata. Schema comparison is a process which can be used to support tasks such as schema integration [BLN86] (Section 2.2.4).
- Detecting changes in hierarchically-structured data [CGM97] (Section 2.3.1) entails inferring correspondences between the nodes in two trees (based on the nodes' labels and their positions in the trees).
- Even text-comparison programs such as `diff` [dif] (Section 2.2.1) can be thought of as inferring correspondences in the way that they align corresponding sequences of text.
- The use of clichés [RW88] in code understanding and code generation (Section 2.1.1) is another area that entails inferring correspondences. Recall that an *overlay* consists of a specification cliché and an implementation cliché. Matching actual code to an implementation cliché (in code understanding), or matching a specification to a specification cliché (in code

generation), entails inferring correspondences between the cliché and the code or specification.

- Case-based reasoning (CBR) [Kol93] (Section 2.3.3) entails inferring correspondences between the problem to be solved and the problem part of a case. The correspondence heuristics required for CBR (Section 9.2.1 of [Kol93]) differ from those required for code comparison in that:
  - A rigidly-structured representation, where corresponding values tend to fill the same named slot, is possible (and thus desirable) for code.
  - In CBR, cases are often represented in an unstructured way, such that it may be necessary to infer correspondences between the fillers of different slots. This can be done by, for example, preferring correspondences between objects that are structurally similar, or that play the same functional role (e.g. a staple can play the same functional role as thesis-binding, even though they are structurally very different).
- Analogical reasoning [FFG89] (Section 2.3.3) entails *analogical mapping* to determine the correspondences between two analogous situations (such as water flow and heat flow). Analogical mapping tends to prefer inferring correspondences between *relationships* ("water flows to a place of lower pressure" corresponds to "heat flows to a place of lower temperature") to inferring correspondences between *objects* (water corresponds to heat, not to hot water). It also prefers correspondences between relations that are part of a larger structure, to correspondences between isolated relations (the *systematicity principle*).

There are actually three orthogonal dimensions of correspondences:

- *Horizontal correspondences* between a situation and the case or rule to which it is matched, as discussed above (e.g. between a code-fragment and an implementation cliché, or between a problem situation and the problem part of a case). Horizontal correspondences are inferred at run-time.
- *Vertical correspondences* between the problem and solution parts of a case, between the antecedent and consequent of a rule, or between the implementation and specification clichés in an overlay. Vertical correspondences are provided statically in the case library, the rule base, or the cliché library.

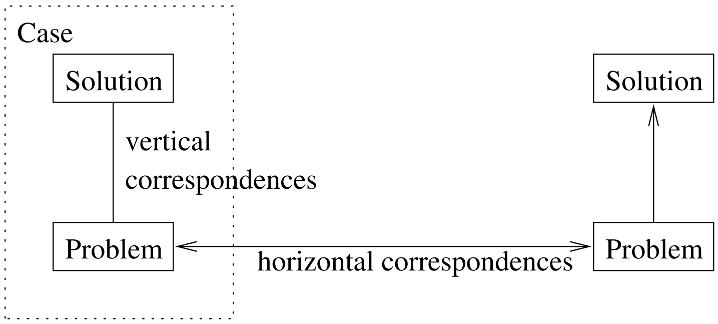


Figure 4.1: Horizontal and vertical correspondences.

Figure 4.1 illustrates the distinction between horizontal correspondences and vertical correspondences in the context of CBR.

- When (code) comparison itself is the problem to be solved – as in combining comparison with the extraction of higher-level knowledge (Chapter 6), or the use of semantic-equivalence/strength cases (Section 5.3) – the correspondences between the codes being compared constitute an orthogonal third dimension of correspondences (the correspondences between code-fragments, as opposed to the correspondences between code-fragments and cases, or the correspondences between the problem and solution parts of cases).

This section first addresses the issue of whether correspondences should be inferred automatically, or whether they should be provided as an additional input to the two codes to be compared (Section 4.1.1). It then discusses how correspondences are represented (Section 4.1.2).

4.1.1 Correspondences: To Infer, or to Provide as Input?

There are three ways that a comparative code-understanding system could acquire its knowledge of the correspondences between the parts of the two codes to be compared:

- It could infer correspondences automatically.
- It could be provided with the correspondences as an extra input (along with the two codes to be compared).

```
ENTITY vegetarian
  SUPERTYPE OF (vegan);
END_ENTITY;

ENTITY vegan
  SUBTYPE OF (vegetarian);
END_ENTITY;
```

```
ENTITY demi_vegetarian
  SUPERTYPE OF (vegetarian);
END_ENTITY;

ENTITY vegetarian
  SUBTYPE OF (demi_vegetarian);
END_ENTITY;
```

Figure 4.2: Two EXPRESS models illustrating synonymy and homonymy.

- A hybrid approach is also possible, whereby it infers some of the correspondences automatically, but is given some guidance for inferring the other correspondences. Such guidance could range from an explicit statement of some specific correspondences for the particular codes being compared, to more generic knowledge such as a thesaurus (Section 4.2.4).

Inferring correspondences automatically is clearly desirable in that it removes the tedium and human error associated with writing a specification of the correspondences by hand (since most existing codes do not come with a specification of correspondences with other codes). However, fully automatic deduction of correspondences has the drawback that without the benefit of a human oracle, the computer can err. In particular, it can suffer from [PSU98]:

- The *synonymy problem*: Corresponding objects may have different names, or differ syntactically, causing the system not to infer correspondence when it should (i.e. inference of correspondences can be *incomplete*).

- The *homonymy problem*: Objects may have the same name, or be syntactically identical, even though they do not correspond. Hence the system might infer correspondence when it should not (i.e. inference of correspondences can be *unsound*).

Figure 4.2 illustrates the synonymy and homonymy problems for a simple pair of EXPRESS models about vegetarianism. In the first model, a *vegetarian* is someone who does not eat meat, and a *vegan* is a vegetarian who abstains from all animal products. However, as reflected in the second model, some vegans call themselves *vegetarians*, and use the term *demi-vegetarian* for those who abstain from meat but eat fish and other animal products. The correct correspondences are:

- `vegetarian` in the first model corresponds to `demi_vegetarian` in the second model;
- `vegan` in the first model corresponds to `vegetarian` in the second model.

Using the *name identicality* correspondence heuristic (Section 4.2.3), the CCUS would infer the following incorrect correspondences and non-correspondences:

- `vegetarian` in the first model corresponds to `vegetarian` in the second model (the homonymy problem strikes);
- `vegan` in the first model does not correspond to anything in the second model (the synonymy problem strikes);
- `demi_vegetarian` in the second model does not correspond to anything in the first model (the synonymy problem strikes again).

The synonymy and homonymy problems do not exist only for name-based correspondence heuristics. The synonymy problem also arises when two semantically equivalent expressions are written differently; in principle, the homonymy problem could arise if two syntactically identical expressions actually had different semantics.

The synonymy problem is well-known in mathematics, in the guise of syntactically different but semantically equivalent expressions (e.g. $x(x+y)$ is equivalent to $x^2 + xy$, as both expressions evaluate to the same value as each other for all possible values of x and y). When both expressions involve the same variables, the synonymy problem reduces to mathematical equivalence, and the homonymy problem cannot arise.

However, it is not that simple in comparative code understanding, because the variables in the expressions being compared are not the *same* variables, but *corresponding* variables in the two codes (the variables must be inferred to correspond before the comparison of any two expressions involving those variables can be completed).

If corresponding variables are not equivalent, then the homonymy problem *could* arise, and conversion factors would need to be taken into account to assess correctly the semantic equivalence of two expressions. For example, Figure 4.3 shows two EXPRESS models, where temperature is interpreted to be in degrees Celsius in the first model and degrees Fahrenheit in the second (this knowledge is implicit, as the semantics of EXPRESS do not take units into account).

```
TYPE celsius = REAL;
END_TYPE;

ENTITY water;
  temperature : celsius;
WHERE
  {0 < temperature < 100};
END_ENTITY;
```

```
TYPE fahrenheit = REAL;
END_TYPE;

ENTITY water;
  temperature : fahrenheit;
WHERE
  {32 < temperature < 212};
END_ENTITY;
```

Figure 4.3: Two EXPRESS models illustrating computational mapping.

To see that the two domain rules are equivalent requires knowing that `temperature` in the first model corresponds to `temperature` in the second model, but with a conversion factor of $F = 32 + \frac{9}{5}C$. This is beyond the scope of the currently implemented CCUS, which assumes that corresponding attributes are equivalent.

In the context of transferring constraints for schema-to-schema mapping (Section 1.3), consider mapping from the second model to the first. If the second model did not already have the domain rule {`32 < temperature < 212`}, it could be derived by transferring the domain rule {`0 < temperature < 100`} from the first (target) model to the second (source) model, using the above conversion formula. However, if the conversion formula was not applied, so that both models had a domain rule {`0 < temperature < 100`}, then the homonymy problem would arise if these models were subsequently compared without taking the conversion factor into account.

As regards providing correspondences as input, as opposed to inferring them automatically, correspondences could be provided using a *mapping language*. Note the distinction between:

- *structural mapping*: this is concerned with the correspondences between the various parts of the codes, such as (in the case of EXPRESS models) entities, attributes, domain rules, expressions and their subexpressions;

- *computational mapping*: this is concerned with value-conversions, such as the Celsius/Fahrenheit example above, and is often ignored or not needed.

Mapping languages exist for EXPRESS, such as EXPRESS-M ([Bai95], Chapter 2 of [Has96]), EXPRESS-X [Lab96] and EXPRESS-C [Sta94], but they do not generalise to other languages, and most languages do not have mapping languages.

EXPRESS-M is now considered, in order to look at a particular mapping language in detail. Although EXPRESS-X is a more advanced, more recently developed mapping language, the examples here are clearer in EXPRESS-M.

EXPRESS-M is a language for specifying the structural mapping between EXPRESS models. It does so in a procedural manner, at a very concrete level of abstraction, specifying how to map an instance of an entity in one model to an instance of the corresponding entity in the other model. Figure 4.4 shows two EXPRESS models, and the EXPRESS-M mappings from the second to the first and vice versa.

The advantages of EXPRESS-M include:

- It has good expressive power (e.g. "`b.person` maps to `a.man` if `gender` is `male`, and to `a.woman` if `gender` is `female`").
- It can specify computational mappings, because the right-hand side of an attribute mapping can be an arbitrary expression (e.g. `temperature:= 32 + temperature*9/5;`).

The drawbacks of EXPRESS-M include:

- It is procedural in nature, couched more in terms of how to perform mappings than what the correspondences are, and it has to specify the mapping in both directions explicitly. Its lack of explicit, bidirectional correspondences is not helpful for comparative code understanding.
- The mappings of inherited attributes (e.g. `name` and `age` in Figure 4.4) have to be redundantly duplicated.

An alternative approach to providing correspondences is to specify them *declaratively*. For example, [BFM95] presents a set of generic operators for schema-to-schema mapping. These operators are still concerned with the direction of mapping, but are much more declarative in nature than EXPRESS-M, and many (though not all) of the operators are automatically reversible. The operators consist of:

- *Schema-level mapping operators*. These are concerned with correspondences between *entities*. There are operators for declaring that two entities are equivalent; for adding entities to, and deleting them from, the source model; for declaring one-to-many entity mappings (where an instance of *each* of the

```
SCHEMA a;
  ENTITY person ABSTRACT SUPERTYPE OF
                (ONEOF (man,woman));
    name : STRING;
    age : INTEGER;
  END_ENTITY;

  ENTITY man SUBTYPE OF (person);
  END_ENTITY;

  ENTITY woman SUBTYPE OF (person);
  END_ENTITY;
END_SCHEMA;
```

```
SCHEMA b;
  TYPE gender_type
    = ENUMERATION OF (male,female);
  END_TYPE;

  ENTITY person;
    name : STRING;
    age : INTEGER;
    gender : gender_type;
  END_ENTITY;
END_SCHEMA;
```

```
SCHEMA_MAP a <- b;
  MAP ONEOF (man,woman) <- person;
    IF gender = 'male' THEN
      MAP man <- person;
        name:= name;
        age:= age;
      END_MAP;
    ELSE
      MAP woman <- person;
        name:= name;
        age:= age;
      END_MAP;
    END_IF;
  END_MAP;
END_SCHEMA_MAP;
```

```
SCHEMA_MAP b <- a;
  MAP person <- man;
    name:= name;
    age:= age;
    gender:= 'male';
  END_MAP;

  MAP person <- woman;
    name:= name;
    age:= age;
    gender:= 'female';
  END_MAP;
END_SCHEMA_MAP;
```

Figure 4.4: Two EXPRESS models and an EXPRESS-M specification of the mappings between them.

target entities can be generated, or only of *one* of the target entities), and so on.

- *Entity-level mapping operators.* These are concerned with correspondences between *attributes* of entities. These are analogous operators to the schema-level operators, but for attributes rather than for entities.

However, these operators, though declarative rather than procedural, are not necessarily more (or even as) intuitive as EXPRESS-M. They also cannot handle conditional mappings such as the EXPRESS-M example given above, nor can they handle computational mappings.

For languages that do not have a mapping language (such as programming languages), one approach to providing correspondences is to *tag* code components so that corresponding components can be identified [BHR95]. For example, when a code is going to be compared with a revision (modified version) of itself, a special editor can be used which allocates these tags automatically.

In the context of schema-to-schema mapping (Section 1.1), the CCUS assumes that instances would be mapped according to the correspondences it infers. However, due to the synonymy and homonymy problems, it is impossible to guarantee the correctness of inferring correspondences in its full generality. The validity of each correspondence heuristic has to be considered:

- A correspondence heuristic would be *sound* if it never inferred that two things correspond when they do not (no false positives). The homonymy problem militates against soundness.

- A correspondence heuristic would be *complete* if it always inferred that two things correspond when they do (no false negatives). The synonymy problem militates against completeness.

The CCUS has adopted the approach of inferring all correspondences automatically, with the aid of thesauri to enable it to infer correspondences between named objects with synonymous names, and between frames of different frame-classes whose correspondence is sanctioned by a *frame-class-thesaurus* (Section 4.2.4). Thus the CCUS's inference of correspondences is fully automated, but with some human guidance (in the form of thesauri) provided at the start of a CCUS run.

An alternative to the full automation of the current CCUS would be for it to interact with a human. When using certain correspondence heuristics, it could ask the user to sanction or veto a particular correspondence. This could enhance soundness and completeness, were it not that to err is human. In the case of name equivalence, the user could have the option of adding the names of the corresponding objects to the thesaurus so that the system would not have to prompt the user again – thus the user would have to spend exactly as much effort as necessary on constructing the thesaurus. There are systems that interact with the user to determine correspondences, such as:

- Paul Pun's system for deducing mappings [Pun91] (Section 2.2.2) interacts with the user to choose between multiple mappings.

- Sari Hakkarainen's work on schema comparison [Hak98] (Section 2.2.4) uses correspondence heuristics that produce a set of suggestions for possible correspondences, which the user sanctions or vetoes. Each correspondence assertion is tagged with a numeric *correspondence degree*, which indicates how promising the correspondence is.

Before correspondence heuristics are discussed, the next subsection considers how the CCUS represents the correspondences it infers.

4.1.2 How the CCUS Represents Correspondences

A correspondence is an assertion that something in one code corresponds to something in the other code being compared. In fact, although one-to-one correspondences are the norm, it is possible to have one-to-many and many-to-many correspondences. However, such correspondences would tend to indicate redundant duplication of constraint semantics in at least one of the models. For example, consider comparing the following entity with itself:

```
ENTITY person;
  name : STRING;
  age : INTEGER;
WHERE
  (age >= 0) AND (age < 130);
  {0 <= age < 130};
END_ENTITY;
```

- `(age >= 0) AND (age < 130)` in the first model corresponds to `(age >= 0) AND (age < 130)` in the second model;

- `(age >= 0) AND (age < 130)` in the first model corresponds to `{0 <= age < 130}` in the second model;

- `{0 <= age < 130}` in the first model corresponds to `(age >= 0) AND (age < 130)` in the second model;

- `{0 <= age < 130}` in the first model corresponds to `{0 <= age < 130}` in the second model.

Whenever the CCUS infers a correspondence, it reports it to the user (in a form similar to the above), and it records the correspondence in the frame-representation of the EXPRESS models (cf. Sections 3.2 and 3.3). The point of recording correspondences is so that they would be available to applications which used the CCUS; it also enables the CCUS to avoid repeating the comparison of a given pair of frames, since it first checks whether they already correspond, or have already been compared but failed to correspond.[1]

The CCUS records each correspondence in one of two ways. The simpler way of recording correspondences is to fill the distinguished slot `corresponds_to` in each frame with a reference to the other frame. More precisely, the `corresponds_to` slot of each frame is filled with the set of frames to which it corresponds (ordinarily a singleton set, since almost all correspondences are one-to-one). The above example could be represented as follows:

```
INSTANCE domain_rule#1; (* (age >= 0) AND (age < 130) in first model *)
  corresponds_to = {domain_rule#3, domain_rule#4};
  ...
END_INSTANCE;

INSTANCE domain_rule#2; (* {0 <= age < 130} in first model *)
  corresponds_to = {domain_rule#3, domain_rule#4};
  ...
END_INSTANCE;

INSTANCE domain_rule#3; (* (age >= 0) AND (age < 130) in second model *)
  corresponds_to = {domain_rule#1, domain_rule#2};
  ...
END_INSTANCE;

INSTANCE domain_rule#4; (* {0 <= age < 130} in second model *)
  corresponds_to = {domain_rule#1, domain_rule#2};
  ...
END_INSTANCE;
```

The second way that the CCUS can represent a correspondence is as a frame of the distinguished frame-class `correspondence` – the two corresponding frames both have their `corresponds_to` slots filled with the `correspondence` frame,

[1]The CCUS frame-comparison algorithm maintains a hash-table for this purpose: each key is a frame from the first model, and its associated value is the set of frames from the second model with which it has been compared but failed to correspond.

rather than filled with each other. This enables the correspondence to be tagged with extra information – *reified* as it were, like the way that relationships can themselves have attributes in entity-relationship modelling (Chapter 3 of [EN94]).

Of course all correspondences could be represented by `correspondence` frames, but it is simpler (saving space and the user's browsing time) to fill the corresponding frames' `corresponds_to` directly with each other when it is not necessary for the correspondences to be tagged with extra information.

It so happens that the current CCUS represents all *correspondences between expressions* using `correspondence` frames, because these are tagged with extra information: namely whether the expressions are equivalent, or which expression is the stronger (Section 4.2.5).

In fact, only the correspondences between unnamed objects are represented using `correspondence` frames. So the above example would actually be represented using `correspondence` frames (rather than setting the `corresponds_to` slots of corresponding frames to each other, as shown above for the purpose of illustration), because domain rules are not inferred to correspond using name equivalence – they correspond if and only if their expressions correspond. This representation of the correspondence between two domain rules is illustrated below:

```
INSTANCE domain_rule#1;
  corresponds_to = {correspondence#1, correspondence#2};
  ...
END_INSTANCE;

INSTANCE correspondence#1;
  frame1 = domain_rule#1;
  frame2 = domain_rule#3;
  semantic_equivalence = semantic_equivalence(equivalent);
END_INSTANCE;

INSTANCE domain_rule#3;
  corresponds_to = {correspondence#1, correspondence#3};
  ...
END_INSTANCE;

...
```

Another piece of information with which correspondences could be tagged is the degree to which they correspond. For example, [PSU98] tag each of their correspondence assertions with a *plausibility factor* between 0.0 and 1.0, representing the degree of belief in that correspondence assertion. However, the CCUS takes the view that two things either correspond or they do not, and thus can be compared or not. One possible use of plausibility factors would be to choose between multiple candidate correspondences, but the CCUS would simply handle these as one-to-many (or many-to-many) correspondences.

A third way to represent correspondences would be to have a table of correspondences, separate from the frame-representation of the codes. This would be essentially like having `correspondence` frames but no `corresponds_to` slots, so correspondences would have to be looked up in the correspondence table. This approach is weakly justified by saving space, and not having `corresponds_to` slots cluttering up the frames, but the CCUS frame-comparison algorithm is simply programmed to ignore `corresponds_to` slots (by marking them with `--IGNORE` in the frame-class model – Appendix D) except for their special purpose.

EXPRESS-M (cf. Section 4.1.1) can also be thought of as a representation for correspondences, but it does not represent them explicitly, declaratively and bidirectionally, and the CCUS would only have to convert an EXPRESS-M specification into an internal form such as that described in this subsection, to facilitate looking up which objects a given object corresponds to. The current CCUS does not represent conditional mappings in full generality (but see Section 6.3.3), nor computational mappings and other sophisticated correspondences (Section 4.2.6.5), but the `correspondence` frames could be extended to do so.

Representing correspondences is not a particularly major issue – essentially, it just states that two objects correspond. Inferring correspondences, striving towards soundness and completeness, is the major challenge addressed in this chapter.

4.2 Correspondence Heuristics

Recall that the CCUS frame-comparison algorithm (Section 3.3) infers that two frames correspond before it proceeds to compare them slot by slot. This section explains in detail how correspondences are inferred.

The CCUS uses various *correspondence heuristics* to infer correspondences – different correspondence heuristics are appropriate for the different kinds of objects that could correspond. They are *heuristic* in the sense that they are not guaranteed to infer correspondences soundly and completely, as explained in Section 4.1.1.

Section 4.2.1 discusses the matter of selecting an appropriate correspondence heuristic for deciding whether a given pair of objects correspond, and then some particular correspondence heuristics are examined in detail in Sections 4.2.2 to 4.2.6.

Pseudocode algorithms for inferring correspondences are given in Appendix C, Section C.3.

4.2.1 Selecting a Correspondence Heuristic

The role of a correspondence heuristic is to sanction (allow) or veto (forbid) the correspondence of a given pair of objects. Figure 4.5 shows the CCUS's process of selecting and applying an appropriate correspondence heuristic for a given pair of frames, in order to decide whether they correspond. The correspondence heuristics are depicted as light grey rounded boxes, and the verdict (depicted as a dark grey box) is one of the following:

- `CORRESPONDENCE`: The frames correspond – proceed with comparison;
- `PLAUSIBLE_CORRESPONDENCE`: The frames may or may not correspond – proceed with comparison, which may have to retract the correspondence if the comparison fails;
- `NO_CORRESPONDENCE`: The frames do not correspond – do not compare them.

When the CCUS first considers a pair of frames for comparison, it starts by applying a *frame-class equivalence* correspondence heuristic, which **vetoes** inferring correspondence between those frames if both of the following are true:

- they are not of the same frame-class (Section 4.2.2);
- they are not of two frame-classes which have been declared to be comparable in a *frame-class-thesaurus* (Section 4.2.4).

Thus frame-class equivalence is a quick test to reject, immediately, many pairs of frames that could not correspond.

If two frame-class-comparable frames are named objects, then the CCUS uses a *name equivalence* correspondence heuristic to sanction or veto inferring correspondence between them.

- The CCUS considers a *named object* to be a frame whose frame-class has the distinguished slot `name` – the frame-class model (Appendix D) is written such that all and only frame-classes representing named objects have that slot.

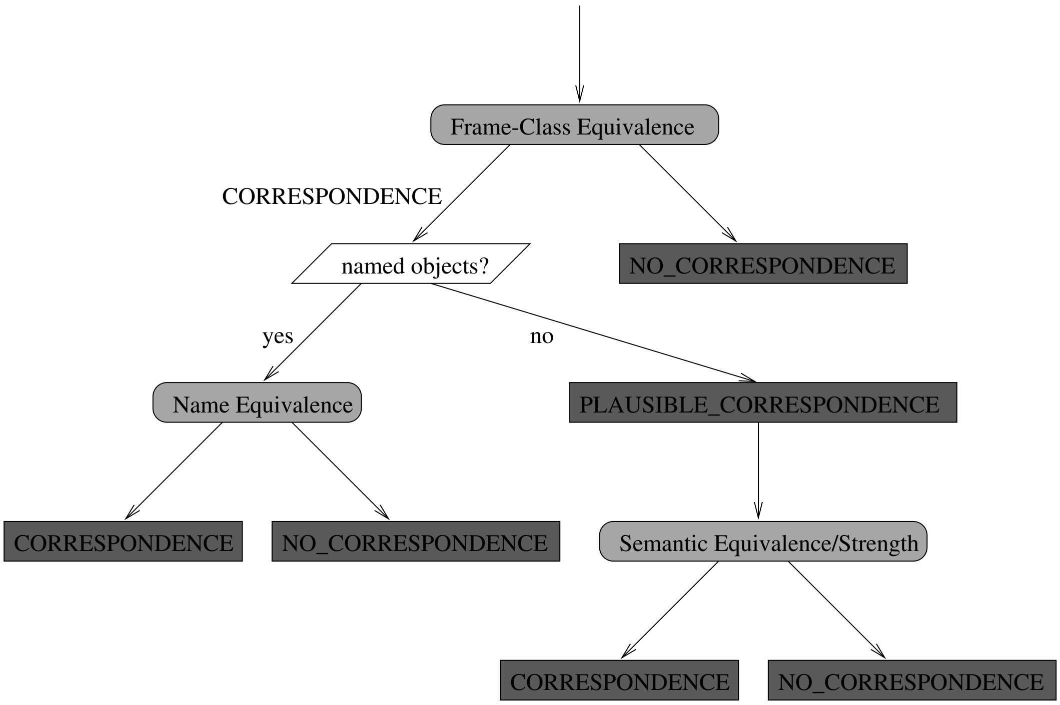


Figure 4.5: Flowchart for the selection and application of correspondence heuristics.

- Correspondence between two named objects is sanctioned if they have identical names (i.e. if their fillers of the slot `name` are string-equal) (Section 4.2.3), or, when using a *name-thesaurus*, if their names are declared as synonyms (Section 4.2.4). Otherwise, the correspondence is vetoed.

If two frame-class-comparable frames are unnamed objects, then the CCUS uses *semantic equivalence* as a correspondence heuristic (Section 4.2.5). It infers a *plausible correspondence* between those frames (a correspondence which may have to be retracted later), and proceeds to compare them slot by slot using the standard CCUS frame-comparison algorithm (Section 3.3), but with special procedures to *resolve* any differences between them by attempting to assess whether they are semantically equivalent, or whether one is semantically stronger than the other.

There are thus two distinct approaches to inferring correspondences:

- inferring whether two frames correspond, and only proceeding to compare them if they do (as in the frame-class equivalence (Section 4.2.2) and name equivalence (Sections 4.2.3 and 4.2.4) correspondence heuristics);

- having to compare two frames in order to infer whether they correspond – a hypothesise-and-test approach (as in the semantic equivalence correspondence heuristic, where it is not possible to decide whether they correspond before comparing them).

Although this framework for inferring correspondences is specific to the comparative code understanding of EXPRESS models, it should generalise to other kinds of code:

- Any code could be represented using frames, so frame-class equivalence would always be an appropriate first test.

- Name equivalence makes sense for other kinds of code – for example, in program code there are named objects such as classes, functions, and so on. It may be reasonable to assume that name equivalence is a valid correspondence heuristic – especially when comparing different revisions of the same code rather than two different codes – although it could certainly be fooled by the synonymy and homonymy problems, in which case it may be appropriate to consider semantic equivalence, or the structure of function calls. Name equivalence may also be more appropriate for a library of independent functions than for a program with nested function calls, because the external functions of a library relate much more directly to real-world concepts than the internal functions of a program.

- Semantic equivalence is relevant to all kinds of code – not just equivalence of expressions, but also *behavioural equivalence* (whether two functions produce the same output, or perform the same external function), and it may even be possible to apply similar techniques to assessing other kinds of equivalence, such as whether two algorithms have the same time and space complexities.

4.2.2 Frame-Class Equivalence

Frame-class equivalence is a quick initial test to veto correspondence between frames which are not of the same frame-class – assumed to be a necessary but not sufficient condition for two frames to correspond. It thus rejects many inappropriate correspondences immediately (for example, it would be inappropriate to compare an attribute `age : INTEGER` in one model with a domain rule `{0`

`<= age < 130`} in the other model – although they are related, they do not correspond). Frame-class equivalence has the additional advantage that the CCUS does not have to worry later on about corresponding frames not being of the same frame-class – for example, this simplifies name equivalence, because it only has to test whether the frame-class of the frames has the distinguished slot `name`, and then compare their names.

Frame-class equivalence suffers from the synonymy problem in that sometimes it *is* appropriate to compare frames of different frame-classes (such as `set_type` and `bag_type`), and it would be a shame to shoehorn them into a common frame-class just for the sake of sanctioning correspondence. Section 4.2.4 explains how the CCUS can get around this problem by means of a frame-class-thesaurus.

It could be argued that frame-class equivalence should respect *subsumption*, i.e. that correspondence should be sanctioned between a frame of one frame-class and a frame of a subtype or supertype (comparison is bidirectional) of that frame-class. However, this would incur two frame-class subsumption tests rather than one frame-class identicality test (i.e. are the frames' frame-classes the same `FrameClass` object?). Moreover, it may not always be appropriate to infer correspondence between two such frames. So if a case occurs where it is necessary to infer correspondence between two such frames, it would be more sensible to declare their frame-classes as synonyms in the frame-class-thesaurus (Section 4.2.4).

The homonymy problem is not a relevant issue for frame-class equivalence because this correspondence heuristic can only *veto* correspondences – it takes other correspondence heuristics, applied after frame-class equivalence, to *sanction* correspondences.

4.2.3 Name Identicality

Name identicality is the simplest *name equivalence* correspondence heuristic, which sanctions correspondence between two named objects if they have identical names (according to standard string equality), and vetoes correspondence if the names differ at all.

The aim of a name-based correspondence heuristic is to infer correspondence between two objects that represent the same concept. While this approach admittedly suffers from being blinded by names, with many kinds of object (such as entities and attributes in EXPRESS models) their names are by far the strongest indicator of the concepts they represent (whereas for other kinds of objects, other correspondence heuristics are appropriate, such as semantic equivalence for expressions).

The CCUS selects a name-based correspondence heuristic for a pair of frames if and only if they have the distinguished slot `name` – the frame-class model (Appendix D) is written such that exactly those frame-classes for which a name-based correspondence heuristic is appropriate have the slot `name`.

Name identicality is a quick and easy way to sanction and veto correspondences (without even having to try other correspondence heuristics), which should work for the majority of named objects. However:

- it will not infer correspondences between some objects that truly correspond (e.g. “woman” and “lady” – the synonymy problem);
- it may infer correspondences between objects that truly *do not* correspond (e.g. “SELECT” in relational algebra selects tuples (Section 6.5.1 of [EN94]), but “SELECT” in SQL selects attributes (Section 7.2 of [EN94]) – the homonymy problem);
- also recall the ‘vegetarian’ example from Section 4.1.1.

The next subsection discusses a way to overcome the synonymy and homonymy problems: by using thesauri.

4.2.4 Name and Frame-Class Equivalence with Thesauri

The synonymy problem associated with frame-class equivalence (Section 4.2.2) and name identicality (Section 4.2.3) can be combatted using thesauri. A *thesaurus* is essentially a separate data-structure that declares sets of frame-classes, or sets of names, to be synonymous. Following *WordNet*[2] [Cog], these sets of synonyms are called *synsets*. A thesaurus is thus a set of synsets.

The *frame-class equivalence* correspondence heuristic can be augmented with a *frame-class-thesaurus*, which specifies sets of frame-classes that may be compared. The CCUS may proceed to compare two frames if they are of the same frame-class, or if they are of two frame-classes which are declared in the frame-class-thesaurus to be comparable (i.e. if the names of the frame-classes are in the same synset). Thus every pair of frames that the CCUS considers which are not

[2] *WordNet* is a registered trademark of Princeton University.

of the same frame-class incurs the cost of a thesaurus-lookup. Therefore, the use of the frame-class-thesaurus is either enabled or disabled for a particular run of the CCUS.

Similarly, the name identicality correspondence heuristic can be replaced by *name equivalence with a thesaurus*. The *name-thesaurus* specifies sets of synonymous names. The CCUS sanctions correspondence between two named objects if their names are identical, or if their names are declared in the name-thesaurus to be synonyms (i.e. if the names are in the same synset), otherwise it vetoes the correspondence. Again, every pair of non-identical names considered by this correspondence heuristic incurs the cost of a thesaurus-lookup. Therefore, the use of a name-thesaurus is either enabled or disabled for a particular run of the CCUS.

Although there is only one frame-class-thesaurus, a particular run of the CCUS can use one of a number of name-thesauri. A thesaurus needs to be specific to the particular pair of models being compared (or at least domain-specific) – it is not appropriate to use one big thesaurus for all models, such as a standard English thesaurus. That would be both unnecessary (a standard English thesaurus is enormous, and thus very expensive in space and time) and insufficient, as the named objects which correspond in EXPRESS models (and code in general) do not tend to be synonyms in a standard English thesaurus, but rather, idiosyncratic technical terms. For example, [WWK94] asserts the following entity correspondences between the EDIF version 3 0 0 and CFI DR 1.3 EXPRESS models respectively:

- `instance` corresponds to `instance`;
- `signalGroup` corresponds to `netBundle`;
- `signal` corresponds to `netScalar`;
- `port` corresponds to `port`;
- `busses` weakly corresponds to `netBundle`;
- `busses` weakly corresponds to `netBus`.

The idea of having to use model-specific thesauri is supported by [PSU98]. Their aim is *schema integration* (Section 2.2.4): integrating a number of federated database schemata into one global schema. Given an initial set of *properties* (synonyms, inclusions[3] and homonyms), their system derives new properties, which it uses to integrate the schemata to be integrated. As noted in Section 4.1.2, these properties have plausibility factors between 0.0 and 1.0 (they do not conform to probability theory, though – an existing property can be rederived with a different plausibility factor).

Dearnley and Smith [DS99] present a method for discovering mappings between the global schema of a federated database system and the local schemata of the component databases, which will be discussed in Section 4.2.6.5. They too use a thesaurus for name equivalence.

The thesauri for the CCUS are implemented using *WordNet* [Cog], an on-line lexical database which can also be downloaded for local use. The *Java WordNet Library (JWNL)* was downloaded for the CCUS to use, an application programming interface (API) for accessing the *WordNet* database. The standard *WordNet* database is an enormous semantic network with extensive coverage of the English language such as can be found in a conventional dictionary or thesaurus, but linked by relationships such as synonymy, *antonymy* (opposites), *hyponymy* (subclasses), *meronymy* (part-of), and morphological relations.

The basic unit of organisation for a *WordNet* thesaurus is the synset. Words are grouped into synsets, each of which can be thought of as representing a concept. There are also *lexical relations* between words (such as antonymy) and *semantic relations* between synsets (such as hyponymy and meronymy), although the current CCUS is only concerned with synonyms. Homonyms are encoded implicitly by the fact that they occur in multiple synsets.

For example, a *WordNet* thesaurus for the example in Figure 4.2 consists of the following two synsets:[4]

```
{vegetarian, demivegetarian,}
{vegan, vegetarian,}
```

As explained above, the CCUS needs model-specific thesauri rather than a standard English thesaurus. Therefore, instead of using the standard *WordNet* database, the CCUS uses its own thesauri, which have to be written as *WordNet* lexicographers' source files (as in the example above), and compiled using the *Grinder* into *WordNet* database files which JWNL can read.

---

[3] An *inclusion property* is a (subclass, superclass) pair of words, e.g. (vegan, vegetarian) – like a synonym, but unidirectional.

[4] Lexicographers' source files use commas (`,`) as terminators rather than as separators.

JWNL provides two ways of accessing the *WordNet* database:

- accessing the database on disk for each lookup;
- loading the whole database into memory at the start, and storing it in a hash-table to support fast lookups.

The latter was chosen for the CCUS because the CCUS performs many thesaurus-lookups (every time frame-class or name identicality fails and thesauri are enabled), so it is far more important to optimise the lookup time than the initial set-up time, especially as the CCUS uses small, model-specific thesauri rather than the standard *WordNet* database.

Since *WordNet* is intended as a lexical database for English, and is rigidly partitioned into four parts-of-speech (nouns, verbs, adjectives and adverbs), and a running program can only have one instance of *WordNet*, it was necessary to abuse the infrastructure of *WordNet* in order to meet the needs of the CCUS. Thus it was decided to use the noun category for the name-thesaurus, and the adverb category for the frame-class-thesaurus (it is conceivable that the name-thesaurus of a more sophisticated CCUS could be partitioned into, say, nouns for entities, adjectives for attributes, and verbs for functions, while adverbs are least likely to occur in code). Technically, the frame-class-thesaurus has to be replicated for every name-thesaurus when building its *WordNet* database (using the *Grinder*).

For a particular run of the CCUS, either a *WordNet* database is specified on the command line, or else no thesauri are used (as in Sections 4.2.2 and 4.2.3). The use of either the frame-class-thesaurus or of a name-thesaurus can be disabled by giving it a *WordNet* database with an empty adverb category or an empty noun category respectively – the CCUS detects this at the start, and can thus avoid the cost of thesaurus-lookups whenever the appropriate identicality correspondence heuristic fails.

Because *WordNet* uses underscores ('_') for collocations (multi-word terms such as `fountain_pen`), this causes a problem for the frame-class-thesaurus, because the names of frame-classes tend to contain underscores (e.g. `entity_type`), but *WordNet* indexes the words between the underscores separately, such that looking up `entity_type` fails. The solution adopted for the CCUS is to write the frame-class-thesaurus without underscores (e.g. {`entitytype, enumerationsymbol,`}), and hence to strip underscores from the names of frame-classes when performing thesaurus-lookups. Stripping underscores also applies, of course, to name-thesauri.

One alternative thesaurus system to *WordNet* that was considered was the *Virtual HyperGlossary* (VHG)[5] [VHG]. VHG uses an XML[6] representation, and could represent the 'vegetarian' synonyms as follows:

```
<termEntry id="vegetarian">
  <term>vegetarian</term>
  <synonym>demivegetarian</synonym>
</termEntry>

<termEntry id="vegan">
  <term>vegan</term>
  <synonym>vegetarian</synonym>
</termEntry>
```

The advantages of VHG include:

- It is a standard.
- It is sufficient for many tasks.
- It is extensible to new relationship types whereas *WordNet* has a rich but fixed set of relationship types.

The disadvantages of VHG include:

- The above representation of synonyms is obviously more long-winded than the corresponding *WordNet* representation, and because it uses terms rather than synsets as the primary unit of organisation, the synonymy relationship is asymmetric. This problem could be obviated by using a human front-end for XML, however.
- No appropriate existing VHG thesauri could be found.

Another alternative would have been to implement the thesaurus system from scratch, writing the thesauri in FIL (Section 3.2.1). The advantages of an FIL-based thesaurus system are:

- It fits in with the 'house style' of the CCUS.

[5] *VHG* is a trademark of VHG.
[6] *XML (eXtensible Markup Language)* is a trademark of the World Wide Web Consortium.

- Any thesaurus that could be written in *WordNet* or VHG could be written in FIL. It is not difficult to imagine a frame-representation of the above thesauri – they are just another example of code, after all, and all code can be represented as frames (Section 3.2.3).
- An FIL thesaurus too could be extended to new relationship types – by editing the frame-class model.
- Frames have inverse slots, which are filled automatically, for both symmetric and asymmetric relationships.
- It would have avoided the difficulties of having to learn and incorporate an existing thesaurus system (i.e. a parser/API for *WordNet* or VHG) into the CCUS (which can already handle FIL).

The disadvantages of an FIL-based thesaurus system are:

- It would be a case of 'reinventing the wheel' when *WordNet* and VHG already exist.
- FIL is not a standard language, so existing thesauri could not be imported directly. They could certainly be converted to FIL, but the effort in writing a conversion program subsumes the effort of learning *WordNet* or VHG anyway.
- The FIL representation of a thesaurus would be just as long-winded as the VHG representation – certainly not as compact as *WordNet*'s synsets.

The advantage of the use of thesauri for frame-class equivalence and name equivalence is that it is more flexible – the CCUS can infer correspondences that cannot be inferred using the identicality correspondence heuristics alone. Thus it improves the completeness of inferring correspondences.

A given thesaurus could never be guaranteed to be complete for all possible models, although a complete thesaurus could, in principle, be written for any given pair of models. This might be tedious for the person who has to write such a thesaurus, of course, especially for large models; it is also very difficult for models which differ so radically that it is difficult for a human to see the correspondences in order to write the thesaurus.

Using a thesaurus could also make inferring correspondences less sound, since synonyms open new avenues for inferring false correspondences – especially if a generic thesaurus is used rather than a model-specific thesaurus.

The homonymy problem for name equivalence could be tackled using a thesaurus, too. If named objects with a given name are not to be inferred to correspond, then that name could be declared to be a homonym, and correspondence between any two objects with that name would be vetoed. However, this anti-correspondence heuristic has not been implemented due to the lack of homonyms found in real models, and due to the following drawbacks:

- Vetoing homonyms militates against completeness, since it could veto perfectly good correspondences – it cannot tell whether the name refers to the same concept or different concepts. Hence vetoing correspondences between homonymous objects would only be any use with model-specific homonyms.
- *WordNet* represents homonyms implicitly by the fact that a word is in multiple synsets, which sacrifices the control that would be necessary to overcome the above drawback.
- Vetoing homonyms would incur a thesaurus-lookup every time two names were identical.

The bottom line of this subsection is that the only validity of name equivalence is a human oracle, and there is no guarantee that the human would not make mistakes. Providing some guidance in the form of a thesaurus can certainly improve soundness and completeness beyond name identicality, though.

4.2.5 Semantic Equivalence and Semantic Strength

Correspondence between unnamed objects is sanctioned or vetoed by using *semantic equivalence and semantic strength* as a correspondence heuristic (e.g. `age >= 0` is semantically equivalent to `0 <= age`, and `age >= 21` is semantically stronger (more constrained) than `age >= 18`). Correspondence is sanctioned if they are semantically equivalent, or if one is semantically stronger than the other; otherwise, they are *semantically incomparable* and correspondence is vetoed. Semantic equivalence and semantic strength are discussed in detail in Chapter 5 – the purpose of this section is to set them in their place as a correspondence heuristic.

Name equivalence can sanction or veto correspondence between named frames immediately, and only compare the frames slot by slot if they do correspond. The semantic equivalence/strength correspondence heuristic, on the other hand, has to proceed with comparing the frames, failing the comparison and vetoing the correspondence if it encounters a difference (i.e. if the frames being compared have different fillers for the same slot) that cannot be resolved. The correspondence is implicitly sanctioned if it is not vetoed in this way.

Thus, when the CCUS encounters two unnamed objects (which have just passed the frame-class-equivalence test, as in Section 4.2.2), it sets up a *plausible correspondence* between them – i.e. a correspondence which may have to be retracted later if the comparison fails. The CCUS frame-comparison algorithm (Section 3.3) proceeds to compare the frames slot by slot in plausible-correspondence mode, which means that whenever it encounters a difference, it attempts to resolve it (to infer that the difference is acceptable, using semantic knowledge), and if it fails to do so, it immediately fails the frame-comparison.

The justification for using semantic equivalence/strength as a correspondence heuristic for all unnamed objects is that in the absence of any semantic knowledge, it reduces to structural equivalence (i.e. it proceeds to compare them slot by slot, failing the comparison if any difference is encountered – any situation in which this is unacceptable calls for the addition of semantic knowledge).

In plausible-correspondence mode, the CCUS attempts to resolve a difference in two ways:

- Certain frame-classes (of the frames currently being compared) have *semantic-equivalence/strength (SE/S) cases* (Section 5.3), which specify:
  - *necessary slots*: the slots that are necessary for semantic equivalence (if the difference is in one of these slots, it fails the comparison immediately);
  - *semantic-strength slots*: the slots that affect semantic strength (e.g. `age >= 21` is stronger than `age >= 18` because $21 > 18$).

  Differences in other slots can be safely ignored – if two frames differ only in slots which are not specified in an SE/S case (if there is one)[7] for their frame-class, then they are semantically equivalent (e.g. it does not matter

[7] Any given pair of frames has at most one applicable SE/S case.

whether a comparator is written $a > b$ or $b < a$). Only a difference in a necessary slot (not a semantic-strength slot), or the lack of an SE/S case that the frames match, cause the CCUS to infer the frames to be semantically incomparable, and thus to fail the comparison and retract the plausible correspondence. SE/S cases are discussed in detail in Section 5.3.

- If the comparison of the frames using an SE/S case fails, the HLC CCUS (if enabled) attempts to extract Higher-Level Constraints (HLCs) from the vicinity of the difference in both models (Section 6.3.1). The HLC CCUS compares any HLCs thus extracted, and if the HLCs correspond, it infers a correspondence between the frames being compared at the current abstraction level – called an *abstract correspondence*, since they correspond at a higher-level of abstraction. For example, the HLC CCUS infers an abstract correspondence between `(age >= 0) AND (age < 130)` and {`0 <= age < 130`} by extracting a `range` HLC from both expressions and inferring that these HLCs correspond. The extraction and comparison of HLCs is discussed in Chapter 6.

Because the CCUS frame-comparison algorithm (Section 3.3) is recursive, transitive retraction of plausible correspondences falls naturally into place. For example, when comparing two domain rules, the CCUS infers a plausible correspondence between them. It proceeds to compare the `domain_rule` frames slot by slot, and hence compares the top-level expressions of the domain rules. It infers a plausible correspondence between the top-level expressions, and proceeds to compare *them* slot by slot. If the comparison of the top-level expressions fails (on account of an unresolvable slot difference), this constitutes an unresolvable difference in the `expression` slot of the `domain_rule` frames, and so the plausible correspondence between the domain rules is retracted, as well as the plausible correspondence between the top-level expressions (and so on). The recursive structure of the CCUS frame-comparison algorithm means that the inference of correspondences propagates inwards, and the retraction of correspondences propagates back outwards.

For example, consider comparing the domain rules `age >= 0` and `age <= 0`. First, the CCUS establishes a plausible correspondence between the `domain_rule` frames:

```
INSTANCE domain_rule#1;
  corresponds_to = {correspondence#1};
```

```
  expression = comparator#1;
END_INSTANCE;

INSTANCE correspondence#1;
  frame1 = domain_rule#1;
  frame2 = domain_rule#2;
END_INSTANCE;

INSTANCE domain_rule#2;
  corresponds_to = {correspondence#1};
  expression = comparator#2;
END_INSTANCE;
```

Then, it compares `domain_rule#1` and `domain_rule#2` along the slot `expression`; thus it establishes a plausible correspondence between `comparator#1` and `comparator#2`:

```
INSTANCE comparator#1;
  corresponds_to = {correspondence#2};
  description = "age >= 0";
  tipe = simple_type(logical_type);
  greater = explicit_attribute#1; (* age *)
  lesser = integer_literal#1; (* 0 *)
  inclusive = inclusive_or_exclusive(inclusive);
  discrete = discrete_or_continuous(discrete);
  which_way_round = way_round(greater_first);
END_INSTANCE;

INSTANCE correspondence#2;
  frame1 = comparator#1;
  frame2 = comparator#2;
END_INSTANCE;

INSTANCE comparator#2;
  corresponds_to = {correspondence#2};
  description = "age <= 0";
  tipe = simple_type(logical_type);
  greater = integer_literal#2; (* 0 *)
  lesser = explicit_attribute#2; (* age *)
  inclusive = inclusive_or_exclusive(inclusive);
  discrete = discrete_or_continuous(discrete);
  which_way_round = way_round(lesser_first);
END_INSTANCE;
```

In comparing `comparator#1` and `comparator#2` slot by slot, the CCUS encounters a difference in the slot `greater` – their fillers of that slot fail the frame-class equivalence test (slot-fillers which are frames are considered different if and only if those frames do not correspond). This difference cannot be resolved, so the correspondence between `comparator#1` and `comparator#2` is retracted (i.e. `correspondence#2` is deleted from their `corresponds_to` slots).

This, in turn, means that `domain_rule#1` and `domain_rule#2` differ along the slot `expression`; this difference cannot be resolved, so the correspondence between `domain_rule#1` and `domain_rule#2` is retracted, too.

The retraction of plausible correspondences is an example of *dependency-directed backtracking* (Chapter 14 of [Win92]; Chapter 6 of [FdK93]), which means that when an inference is retracted, any other inferences that depend (transitively) on that inference are also retracted. Contrast this with *chronological backtracking*, where all inferences that were made since the inference to be retracted was hypothesised are also retracted, even if they do not depend on that inference (for example, in the process of comparing `age >= 0` and `age < 130`, the CCUS may infer, using name identicality, that the attributes `age` and `age` correspond – if it happens to compare the domain rules of a pair of entities before it compares their attributes – but it would not retract the attribute-correspondence when it retracts the domain-rule-correspondence, because correspondence between attributes is not plausible).

Other examples of dependency-directed backtracking include:

- *Truth-maintenance systems* (Chapter 6 of [FdK93]; Chapter 6 of [Pra94]) keep track of the reasons for each of an agent's beliefs by maintaining a dependence graph in which each node represents either a *belief* itself (labelled as currently believed or not) or a *justification* for one belief in terms of other beliefs. This network of beliefs is extended as reasoning progresses, and if one belief is retracted, any other beliefs that depend on it are retracted. An inference engine (such as a rule-based system) can be augmented with a truth-maintenance system (Chapter 8 of [FdK93]) such that it sets up belief nodes for the assertions it is given and infers, and it sets up a justification node every time it uses a rule to infer a new belief.

- In *propagation via constraints* (Chapter 15 of [FdK93]), information flows through a constraint network whenever a new value can be computed from existing values (see Section 5.2.3.2). If a value is retracted, all values which depend on that value must also be retracted. Related to this are *constraint-satisfaction* problems (Section 3.5 of [RK91]) such as cryptarithmetic, which are solved by a search process that involves guessing values for unknown variables, propagating the values in accordance with the constraints of the problem, and reguessing if the values tried do not work.

- In *planning* (Chapter 13 of [RK91]), dependency-directed backtracking can be used to retract only those parts of a plan that need to be retracted. For example, when planning a journey, one might have to abandon catching a train if one fails to buy a ticket at the railway station, but not if one fails to buy a newspaper (such plan-repair may take place either at planning time or at execution time).

- In *natural-language parsing*, words in a sentence are assigned parts-of-speech (verbs, nouns, adjectives, and so on), but the wrong part-of-speech may be assigned to a homonym, and this might only be detected later in the sentence when the assignments so far make the sentence ungrammatical (Section 15.2.1 of [RK91]). For example, in "The cotton clothes are made of grows in Mississippi", the reader might interpret "cotton" as an adjective at first, but have to change it to a noun upon encountering the word "grows", without necessarily having to retract the other previously-inferred parts-of-speech. Of course, some sentences have multiple grammatically-correct readings anyway (cf. "Time flies like an arrow", "Fruit flies like a banana").

The CCUS's assessment of semantic equivalence and strength is:

- *sound* (it never infers that two things are equivalent, or that one is stronger than the other, when they are in fact incomparable), but
- *incomplete* (it may infer that two things are incomparable when they are not).

In the context of transferring constraints for schema-to-schema mapping (Section 1.3), this means that unnecessary constraint-checks may be performed, but necessary constraint-checks would never be omitted.

It also means that inferring correspondences using semantic equivalence/strength as a correspondence heuristic is in turn *sound* (it never infers that two things correspond when they do not) but *incomplete* (it does not always infer that two things correspond when they do).

However, the CCUS is only sound to the extent of the assumption that syntactic identicality implies semantic equivalence (Sections 3.3 and 4.1.1), and to the extent that the SE/S cases and HLC-extraction cases are correct (they are human-written, after all).

Completeness could only be achieved with knowledge of **all** possible ways in which semantically equivalent constraints could be written; the CCUS can be made more complete by adding SE/S cases and HLC-extraction cases.

To summarise Section 4.2 so far, the CCUS uses the following correspondence heuristics. Whenever it considers two frames for comparison, it uses *frame-class equivalence* as a quick test to veto correspondence between them if they are not of the same (or compatible) frame-classes. Then, if they pass the frame-class-equivalence test:

- If they are named objects, it uses a *name-equivalence* correspondence heuristic to sanction or veto their correspondence, and proceeds to compare them if and only if they do correspond.
- If they are unnamed objects, it uses *semantic equivalence/strength* as a correspondence heuristic: it infers a plausible correspondence between the objects and proceeds to compare them, retracting the plausible correspondence if it encounters a difference that it cannot resolve – either by an SE/S case (Section 5.3), or by extracting and comparing HLCs (Chapter 6).

4.2.6 Other Correspondence Heuristics Not Implemented

There are, of course, correspondence heuristics that could be implemented other than the ones presented in the earlier subsections.

4.2.6.1 Structural Similarity

All the correspondence heuristics implemented for the CCUS are based on the idea that two objects either correspond or they do not, and that this can be decided by considering the two objects in isolation.

An alternative idea would be to infer correspondences between the most similar pairs of objects, selecting the best of several candidate correspondences and rejecting the others, rather than considering each correspondence in isolation and inferring it if and only if it is sanctioned.

Consider Figure 4.2 again, for example. An alternative correspondence heuristic to name equivalence would be *isomorphism*: the two models are isomorphic because they are identical up to identifier-renaming. The correspondence (`vegetarian`, `demi_vegetarian`) is preferred to the correspondence (`vegetarian`, `vegetarian`) because the former pair of entities are structurally isomorphic; similarly, the correspondence (`vegan`, `vegetarian`) is preferred to the correspondence (`vegan`, `demi_vegetarian`). Isomorphism is used in Paul Pun's work on deducing mappings between knowledge bases and databases [Pun91] (Section 2.2.2).

Note that the isomorphism correspondence heuristic would not necessarily be based on exact isomorphism, because that would mean that corresponding objects could not have **any** structural differences (such as an entity having an extra attribute in one of the models), which is grossly inflexible. Instead, it could infer correspondences between the most structurally similar objects using *nearest-neighbour matching* (Section 9.3.1 of [Kol93]), rejecting other candidate correspondences (on the assumption that correspondences are one-to-one, as they so often are in comparative code understanding – cf. Section 4.1.2).

A variation of nearest-neighbour matching would be to consider only certain kinds of structural relationships between objects, such as inheritance or association (an entity a is associated with another entity b by having an attribute of type b). For program code, this could entail comparing call graphs, control/data-flow graphs, subsystem hierarchies, or even cliché hierarchies (Section 2.1) – which would be an example of combining comparison and extraction (cf. Chapter 6). It is not clear what the criteria for selecting such correspondence heuristics would be – perhaps several correspondence heuristics could be applied on a 'try it and see' basis, inferring the best correspondences (based on some scoring function, or precedence of correspondence heuristics) that they suggest.

The nearest-neighbour correspondence heuristic might well be an appropriate replacement – or better still enhancement – for name equivalence in certain situations. The purpose of any correspondence heuristic is to sanction correspondence between objects which represent the same real-world concept – name equivalence can always be fooled, so it would be more intelligent to take other factors into account in the quest for sound and complete inference of correspondences.

Nearest-neighbour matching would *not* be an appropriate replacement for the semantic equivalence/strength correspondence heuristic (Section 4.2.5). Semantic equivalence/strength is *sound* to the extent of the assumption that syntactically identical expressions are semantically equivalent (Sections 3.3 and 4.1.1), and syntactically identical expressions would obviously have the highest similarity score anyway. The only sound way to improve the *completeness* of semantic equivalence/strength is to add SE/S cases and HLCs (and other techniques for rewriting expressions or rearranging equations) which capture further mathematical equivalences. The expressions `(a * b) + (c * d)` and `(a + b) * (c + d)` may be structurally similar, but it would be ridiculous to infer a correspondence between them.

Correspondence heuristics which are based on relative similarity rather than absolute correspondence tend to be used for *retrieval* problems, such as:

- retrieving a case in *case-based reasoning* to solve a new problem by analogy with an old one [Kol93] (Section 2.3.3) – in conventional CBR, the case which the new problem best matches is selected;
- *information retrieval* [FO96], where documents are ranked according to how well they match the search query;
- retrieving reusable components from a software library to meet a given specification [FFD96].

In retrieval, there is a clear distinction in status between the retrieval prompt and the objects retrieved – the latter must match the former but not necessarily vice versa – whereas in comparative code understanding, the two codes are 'equal in status'. Nevertheless, retrieval can be thought of as inferring correspondences between the retrieval prompts and the retrieved objects, and comparative code understanding can be thought of as 'retrieving' an object in one code that corresponds to an object in the other. Thus similarity-based correspondence heuristics, while not as appropriate for comparative code understanding as for retrieval, may be appropriate in many circumstances.

4.2.6.2 Statistical Analysis

Another idea would be to use some form of *statistical analysis*: if the CCUS could accept feedback on which of the correspondences it infers are correct (and which correspondences it should have inferred, but did not), it could maintain a reliability factor for each correspondence heuristic (i.e. the percentage of correspondences it inferred which were correct). It would also be desirable if it could keep statistics on the specific circumstances in which correspondences were inferred. It could then sanction a correspondence only if its reliability (based on these statistics) is above a certain threshold (say 50%).

4.2.6.3 Advanced Name Equivalence

The current name-equivalence-with-a-thesaurus correspondence heuristic (Section 4.2.4) uses only synonyms, but it could make use of the other relations provided by *WordNet*:

- It could veto correspondence between named objects whose names are homonyms. However, this would only work properly with explicitly-declared homonyms, as opposed to *WordNet*'s implicit homonyms which could veto perfectly good correspondences.

- Hyponyms could aid in inferring correspondences between entities in inheritance hierarchies (e.g. an entity `person` could be inferred to correspond to an entity `human` based on the fact that they both have subclasses `man` and `woman`, if the names of these two subclasses are declared as hyponyms of `person`, rather than declaring `person` and `human` as synonyms).

- Meronyms could aid in inferring correspondences between entities in association (via attributes) with other entities (e.g. an entity `book` could be inferred to correspond to an entity `thesis` based on the fact that they both have attributes of type `chapter`, if `chapter` is declared as a meronym of `book`, rather than declaring `book` and `thesis` as synonyms).

- Antonyms could aid in inferring correspondences between binary choice sets (as in `ENUMERATION OF (male, female)` and `SELECT (man, woman)` – in the current implementation, this would be handled simply by declaring the synsets {`male, man,`} and {`female, woman,`}, which is ugly because nouns are not supposed to be synonymous with adjectives).

The name-equivalence correspondence heuristic could also be extended to handle decomposable names. This is related to *collocations* in *WordNet* (e.g. "fountain_pen"), and to *inflectional morphology* (reasoning about the meaningful substrings of words). This would be particularly beneficial for transformations such as flattening (Section 1.4.2.1), which often uses naming conventions such as the use of underscores ('_') to combine the names of entities (e.g. in Figure 1.6, the flatten-down algorithm would actually name the entities `person_man` and `person_woman`, rather than `man` and `woman` as shown there for the purpose of illustration).

Since name equivalence is based on exact string-matching, differences in spelling would cause a problem (e.g. "organisation", "organization"). This could be overcome by considering the *editing distance* between names (how many characters are changed from one name to another), and sanctioning correspondence if this is below a certain threshold. This would obviously militate against soundness in the case of words with similar spellings that mean totally different things (e.g. "complement", "compliment") – an extended version of the homonymy problem.

4.2.6.4 Pair by Order

If all else fails, it might even be an idea to pair objects in the order in which they appear in the codes (i.e. use an ordered-aggregation-matching policy for a pair of unordered aggregations – cf. Section 3.4). This would make sense for comparing a code with a revision (modification) of itself (especially if some objects have been renamed such that name equivalence fails), comparing a code with a transformed version of itself (if the transformation preserves the order of corresponding objects), or even comparing two models of the same thing such that the second model has been written based on the first, preserving the order.

4.2.6.5 Inferring More Sophisticated Correspondences

Dearnley and Smith [DS99] present a method for discovering mappings between the logical schema (considered as a set of entities and their attributes) of a federated database system, and the local schemata (considered as relational databases) of the component databases. They assume six types of mappings from logical entities to relational tables, and use a decision tree to recognise these mappings for a particular federated database system:

- *Direct-storage model mapping*: the obvious mapping from entities to relational tables.
- *Horizontal-partition mapping*: the instances of an entity are partitioned into two or more tables with the same attributes.[8]
- *Vertical-partition mapping*: the attributes of an entity are partitioned into two or more tables (e.g. an entity with attributes {`name`, `nationality`, `age`} could be mapped to two tables, with attributes {`name`, `nationality`} and {`name`, `age`}, where `name` is the primary key).
- *Master-detail mapping*: the attributes of an entity are partitioned into a *master table* and a *detail table* such that there is a one-to-many relationship from the instances in the master table to the instances in the detail

[8]Note that there is no meaningful analogy of this in EXPRESS/FIL, because FIL instances are not organised in tables – the set of instances of an entity can be thought of as either a single table for the entity, or as a set of singleton tables.

table (e.g. an entity representing tennis players' biographical details and match-results could be mapped to a master table for their biographical details and a detail table for their individual match-results, which eliminates redundancy).

- *Multi-column mapping*: an entity is mapped to a table with attributes corresponding to particular instances of the data (e.g. an entity representing tennis players' results in a tournament, with attributes {`name`, `round`, `result`}, could be mapped to a table with attributes {`round`, `monica_result`, `jelena_result`, `iva_result`, `karina_result`, `daniela_result`}).

- *Multiple-table mapping*: an entity is mapped to a set of tables corresponding to particular instances of the data (e.g. an entity representing tennis players' results in a tournament, with attributes {`name`, `round`, `result`}, could be mapped to a table for each player, each with attributes {`round`, `result`}).

The first four are mappings that implement partitionings, whereas the multi-column and multiple-table mappings handle data that are instance-data in one database but metadata in another. The CCUS, which currently only infers direct, one-to-one entity mappings, could be extended to infer more complex mappings such as the last four listed above. However, the method of Dearnley and Smith examines not just the schemata of the databases, but also the instance-data – it would be a major paradigm shift for the CCUS to take instance-data into account.

It would be desirable to extend the CCUS to infer more complex correspondences than direct mappings between one entity and another. For instance, it would be desirable to be able to infer *conditional mappings* (Section 1.2.2 of [Has96]), where an entity in one model is mapped to one of several entities in the other model, depending on some condition. See the models in Figure 4.4 for example. As presented in Section 6.3.3, the CCUS can extract a `choice_set` HLC from each of these models, and in the process of inferring correspondence between these HLCs, it infers abstract correspondences between `man` and `male`, and between `woman` and `female` (sanctioned by a name-thesaurus). The current CCUS cannot handle conditional mapping in its full generality, however – for example, if `SCHEMA a` in Figure 4.4 were flattened down, it would not be able to infer correspondences with `person` in `SCHEMA b` (essentially a flattened-up transformation of `SCHEMA a`). A possible solution for this example would be to use the fact that `man` and `woman` are hyponyms of `person` to infer a conditional correspondence.

<pre>ENTITY record;
  title : STRING;
  artist : STRING;
END_ENTITY;</pre>
	<pre>ENTITY record;
	  title : STRING;
	  artist : artist;
	END_ENTITY;
	
	ENTITY artist;
	  name : STRING;
	UNIQUE
	  name;
	INVERSE
	  records : SET [1 : ?] OF record FOR artist;
	END_ENTITY;</pre>

Figure 4.6: Mapping to an encapsulated entity – create or reference?

It would also be desirable for the CCUS to handle *encapsulated entities* properly (Section 1.2.5 of [Has96]), where an entity in one model does not have a corresponding entity in the other model, but is represented as an attribute of another entity. Consider mapping from the first model in Figure 4.6 to the second, for example. The current CCUS merely reports that the attributes `artist` correspond, and that the entity `artist` does not correspond to anything in the first model. It is beyond the current CCUS to infer the correspondence between the attribute `artist` in the first model and the attribute `name` in the second – a common-sense heuristic might suggest this correspondence based on the fact that `name` is the only attribute of type `STRING` in the entity `artist`, and it does not correspond to anything else in the first model.

Another issue, when mapping from an attribute to an entity in another model, is whether to create a new instance of the target entity or to refer to an existing instance (Section 1.2.3 of [Has96]). Consider mapping from the first model in Figure 4.6 to the second, for example. An instance of the entity `artist` should be created only if an artist of the name in question does not already exist, otherwise a reference to an existing artist should be used as the value of the attribute `artist`. The subtleties of this correspondence are lost on the current CCUS. A naïve schema-to-mapping mapping system based on correspondences inferred by the CCUS would probably create an instance of `artist` for every instance of `record`, although it could use the `UNIQUE` clause to decide to refer to existing instances of `artist`. Alternatively, a human could tell it whether to create or reference.

4.2.6.6 Meta Correspondence Heuristics

In the current implementation, the selection of a correspondence heuristic (Section 4.2.1) is hard-wired. An alternative would be to use *meta correspondence heuristics* which select an appropriate correspondence heuristic in a data-driven manner. This could add flexibility and power to inferring correspondences, but it would be computationally more expensive, and it is not clear that it would be significantly more effective.

Section 4.2 has discussed the selection of a correspondence heuristic to sanction or veto correspondence between a given pair of frames (Section 4.2.1), presented the implemented correspondence heuristics (Sections 4.2.2 to Section 4.2.5), and ideas for further correspondence heuristics (Section 4.2.6).

The next section presents a mini-contribution related to inferring correspondences: how to infer correspondence when an expression in one code corresponds to a subexpression of an expression in the corresponding position in the other code.

4.3 Aligning Expressions with Subexpressions

Consider the pair of models and the frame-representation of their domain rules shown in Figure 4.7. The CCUS vetoes correspondence between `domain_rule#1` and `domain_rule#2` because their fillers of the `expression` slot, `comparator#1` and `conjunction#2`, fail the frame-class equivalence test. However, it would be a shame to miss the correspondence between `comparator#1` and `comparator#2`.

To add some flexibility to the CCUS in this matter, *subexpression cases* have been implemented, which enable the CCUS to align an expression in one model (call it the *primary expression* – `comparator#1` in the above example) with subexpressions of an expression in the corresponding position in another model (`conjunction#2` in the above example).

Two subexpression cases have been implemented: one which tells the CCUS to compare each of the conjuncts of a conjunction with the expression in the other model (Figure 4.8), as needed in this example, and another subexpression case which does the same thing for the disjuncts of a disjunction (Appendix E, Section E.4).

The subexpression case in Figure 4.8 is written in the case-representation language FIL-C (see Appendix B, Section B.1):

```
ENTITY person;
  name : STRING;
  age : INTEGER;
WHERE
  age >= 0;
END_ENTITY;
```

```
ENTITY person;
  name : STRING;
  age : INTEGER;
WHERE
  (age >= 0) AND (age < 130);
END_ENTITY;
```

```
INSTANCE domain_rule#1;
  expression = comparator#1;
END_INSTANCE;

INSTANCE comparator#1;
  description = "age >= 0";
  ...
END_INSTANCE;
```

```
INSTANCE domain_rule#2;
  expression = conjunction#2;
END_INSTANCE;

INSTANCE conjunction#2;
  conjuncts = {comparator#2, comparator#3};
END_INSTANCE;

INSTANCE comparator#2;
  description = "age >= 0";
  ...
END_INSTANCE;

INSTANCE comparator#3;
  description = "age < 130";
  ...
END_INSTANCE;
```

Figure 4.7: Two EXPRESS models with subexpressions that need aligning.

- This case is retrieved when `conjunction#2` in Figure 4.7 is submitted as a retrieval *prompt*, since a case is indexed by the frame-class of the top-level frame in its problem part (`conjunction#1` in Figure 4.8).

- Then `conjunction#2` is matched to `conjunction#1`,[9] and the variable `?c` is bound to {`comparator#2, comparator#3`}.

- Since `conjunction#2` matches the problem part of the case, the frame in the solution part (`submit#1`) is instantiated with `frames =` {`comparator#2, comparator#3`}, which tells the CCUS to compare `comparator#1` (the primary expression) with each frame listed in the `submit` frame.

Thus `comparator#1` is compared with `comparator#2` and inferred to correspond (by semantic equivalence – Section 4.2.5); it is also compared unsuccessfully with `comparator#3` (`age >= 0` is semantically incomparable to `age < 130`).

The procedure for using subexpression cases is called both ways round – with each of the two frames (`comparator#1` and `conjunction#2` in the example) as the

[9]Note that `conjunction#2` has an *internal instance-number* whereas `conjunction#1` has an *external instance-number* (cf. Section 3.2.1) – `conjunction#2` could have been called `conjunction#1` and it would still be a distinct frame.

```
(****************************************************)
(* subexpression-conjunction.filc   2000:04:07 16:03 *)
(****************************************************)
(* When matching a frame f to a conjunction frame c, *)
(* match f to all the conjuncts of c.                *)
(****************************************************)

INSTANCE subexpression_case#1;
  language = express;
  description = "A case for walking the conjuncts of a conjunction";
  problem = conjunction#1;
  solution = submit#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE conjunction#1;
  conjuncts = ?c;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE submit#1;
  frames = {?c};
END_INSTANCE;
```

Figure 4.8: The subexpression case for conjunctions.

primary expression – whenever frame-class equivalence fails (which still vetoes the correspondence between the two frames, regardless of whether the subexpression-case procedure infers any correspondences). Thus every failure of frame-class equivalence incurs at least the cost of two attempted case-retrievals (a subexpression case is only retrieved if the prompt frame is a conjunction or a disjunction).

The subexpression-case procedure can thus be thought of as a correspondence heuristic that sanctions correspondence between an expression in one model and a subexpression in another model. It can also be thought of as a 'semi-aggregation-matching policy' (cf. Section 3.4), since it compares a frame in one model with each of the conjuncts/disjuncts of a conjunction/disjunction in the other model.

The current implementation of aligning expressions with subexpressions has its limitations. For example, the CCUS cannot infer that `(age >= 0) AND (age < 130)` is stronger than `age >= 0`. Two alternative ideas for achieving this are:

- Treat `age >= 0` as an implicit singleton conjunction: extract a `conjunction` HLC from `comparator#1`, and compare it with `conjunction#2` using the SE/S case for conjunctions (Appendix E, Section E.3.3). Implicit singleton conjunctions would only be extracted when triggered by the frame-class of the frame in the other model (`conjunction#2`), otherwise it could end up extracting an excessive number of extraneous surrounding constructs (implicit singleton disjunctions, additions, multiplications, and so on).

- Treat subexpression cases as stylised SE/S cases, which cause the CCUS to infer, in this example, that `comparator#1` corresponds to `conjunction#2` with `comparator#1` weaker (or stronger, in the case of disjunctions) than `conjunction#2`.

Related to this is the way that the constraints specified for an entity are semantically in conjunction. For example, the second model in Figure 4.7 is equivalent to a model which, instead of the domain rule `(age >= 0) AND (age < 130)`, has two domain rules `age >= 0` and `age < 130`. The current CCUS would infer that `(age >= 0)` corresponds to `age >= 0` (semantically equivalent), and that `(age < 130)` corresponds to `age < 130` (semantically equivalent), but it would not infer a one-to-two correspondence: that `(age >= 0) AND (age < 130)` is semantically equivalent to `age >= 0` and `age < 130` collectively. One rather ad hoc solution is to extract a `conjunction` HLC representing the implicit conjunction of an entity's constraints.

The technique of aligning expressions with subexpressions also needs to be extended to subexpressions of arbitrary depth, if it is to handle more complicated examples such as `((age >= 0) AND (age <= 18)) OR (age >= 35)`. The alignment of subexpressions needs to be propagated, which could be achieved by chaining together the applications of subexpression cases.

It could also be extended to walk down the subexpressions of both top-level expressions (other than when they match anyway, which is just the standard CCUS frame-comparison algorithm of Section 3.3), although it would be inefficient to walk down both expressions unselectively for all expressions.

Though the top-level expressions would tend to be semantically incomparable in such cases, it would be interesting to align corresponding subexpressions. For example, they may be an indication that two models have a different structure yet have the same constraints (as in flattening – Section 1.4.2.1).

4.4 Conclusion

This chapter has focused on inferring correspondences, which is one of the four subcontributions of this thesis on comparative code understanding. It has argued for inferring correspondences automatically over providing them as input (Section 4.1.1), and discussed the representation of correspondences (Section 4.1.2). It has discussed the selection of a correspondence heuristic for a given pair of objects (Section 4.2.1), and presented the particular correspondence heuristics used by the CCUS, namely:

- frame-class equivalence as an initial filter (Section 4.2.2);
- name equivalence for named objects (Section 4.2.3);
- the use of thesauri to augment frame-class equivalence and name equivalence (Section 4.2.4);
- semantic equivalence/strength for unnamed objects, which entails the propagation and possible retraction of plausible correspondences (Section 4.2.5);
- other correspondence heuristics not implemented (Section 4.2.6).

The contribution is bringing these ideas together.

Section 4.3 presented a mini-contribution: the use of subexpression cases to align an expression in one code with a corresponding subexpression in the other code, enabling the CCUS to infer appropriate correspondences between expressions such as `age >= 0` and `(age >= 0) AND (age < 130)`.

The use of semantic equivalence/strength *as a correspondence heuristic* is believed to be a novel contribution. The topic of semantic equivalence and semantic strength, including a presentation of how the CCUS assesses them, is discussed in the next chapter, as it constitutes the second subcontribution of this thesis.

Chapter 5

Semantic Equivalence and Semantic Strength

This chapter defines semantic equivalence and semantic strength, discussing techniques that may be used for assessing them, and areas where they may be applied. This chapter focuses mainly on the semantic equivalence/strength of expressions, and at the code level. Semantic equivalence of structure belongs to Chapters 3 and 4, and semantic equivalence/strength at higher levels of abstraction belongs to Chapter 6.

This chapter has a strong link with Section 4.2.5 – using semantic equivalence/strength as a correspondence heuristic. Before this correspondence heuristic can decide whether two frames correspond, the Comparative Code Understanding System (CCUS) must compare them. If it encounters a difference, it attempts to resolve it using semantic-equivalence/strength cases (Section 5.3) to assess whether the two frames are semantically equivalent, whether one is semantically stronger than the other, or whether they are semantically incomparable – in which case the comparison fails and they do not correspond.

5.1 Introduction

Recall the definitions from Section 1.1:

- A *constraint* is a condition in a model that must not be violated by instances of the model's entities.
- A constraint is *satisfied* if the condition evaluates to true – or even to

`ENTITY person;`
`  age : INTEGER;`
`WHERE`
`  age >= 0;`
`END_ENTITY;`
	`ENTITY person;`
	`  age : INTEGER;`
	`WHERE`
	`  0 <= age;`
	`END_ENTITY;`

Figure 5.1: Two EXPRESS models with semantically equivalent domain rules.

'unknown' – and *violated* if it evaluates to false. So for a given instance of an entity, each constraint is either satisfied or violated.

- An instance is *valid* if and only if it satisfies all of the constraints specified for the entity of which it is an instance, otherwise it is *invalid*.
- A data repository *conforms* to a model if and only if every instance in the repository is valid, and no *inter-instance constraints* specified for the model are violated (constraints which must be satisfied by sets of instances collectively, such as global rules – Appendix A, Section A.6 – and uniqueness rules – Section A.7).

5.1.1 Semantic Equivalence

Consider the two EXPRESS models in Figure 5.1. The two domain rules in Figure 5.1 (`age >= 0` and `0 <= age`) are semantically equivalent (even though they are syntactically different) because one is satisfied just when the other is satisfied.[1]

In general, two constraints are *semantically equivalent* if they are both satisfied in exactly the same circumstances – if the set of possible instances that satisfy one constraint is equal to the set of possible instances that satisfy the other.

In the context of schema-to-schema mapping (Section 1.1), a constraint in one schema is semantically equivalent to a constraint in the other schema if any instance that does not violate the first constraint can be mapped to the other schema without violating the second constraint, and vice versa.

This chapter will ignore the problem of *computational mapping*, where value-conversions (e.g. degrees Celsius to degrees Fahrenheit) have to be taken into

[1]This relies on the assumption that `person` corresponds to `person` and `age` corresponds to `age` – these correspondences can be inferred using the *name identicality* correspondence heuristic (Section 4.2.2).

```
ENTITY person;
  age : INTEGER;
WHERE
  age >= 0;
END_ENTITY;
```

```
ENTITY person;
  age : INTEGER;
WHERE
  age >= 18;
END_ENTITY;
```

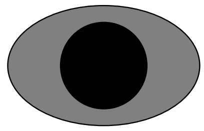


Figure 5.2: The second domain rule is stronger than the first.

account before semantic equivalence can be assessed correctly. See Sections 1.3 and 4.1.1 for a discussion of computational mapping.

5.1.2 Semantic Strength

Now consider the two domain rules in Figure 5.2. Mapping an instance that satisfies `age >= 0` would violate `age >= 18` in the case that `age` < 18, but an instance that satisfies `age >= 18` would never violate `age >= 0`. Thus `age >= 18` is said to be *semantically stronger* (more constrained) than `age >= 0`, and `age >= 0` is said to be *semantically weaker* (less constrained) than `age >= 18`.

In general, one constraint is *semantically stronger* than another if whenever the first constraint is satisfied then the second constraint is satisfied, but it is possible to satisfy the second constraint yet violate the first constraint. To put it another way, the set of instances that satisfy the stronger constraint is a proper subset of the set of instances that satisfy the weaker constraint, as depicted by the Venn diagram in Figure 5.2.

Saying that a constraint c_1 is stronger than another constraint c_2 is the same as saying that c_1 *implies* c_2 (i.e. if c_1 is satisfied then c_2 is satisfied), except that "c_1 implies c_2" actually means "c_1 is stronger than or equivalent to c_2". Thus c_1 is equivalent to c_2 when c_1 implies c_2 and c_2 implies c_1.

Semantic strength of constraints has been called *constraint subsumption*, for example by [GSUW94]: "c_2 subsumes c_1" means "c_1 implies c_2".

There is a duality between constraints and database queries, in that a query returns a subset of the instances in a database, and a constraint admits the subset of all possible instances which are valid.

Semantic equivalence and strength of queries are known in the database field as *query equivalence* and *query containment* [CV94]. If a query q_2 contains another query q_1, then the set of instances returned by q_1 is a subset of the set of instances returned by q_2 – in the terminology of this thesis, q_1 is stronger than or equivalent

NAME	NAT	DOB
Monica Seles	USA	19731202
Jelena Dokić	YUG	19830412
Iva Majoli	CRO	19770812
Karina Habšudová	SVK	19730802

Table 5.1: The relational database PLAYER, to which the queries in Figure 5.3 pertain.

```
SELECT NAME
FROM   PLAYER
WHERE  DOB < 19751028
```

```
SELECT NAME
FROM   PLAYER
WHERE  DOB < 19780420
```

Figure 5.3: Two SQL queries for the database in Table 5.1, the second of which contains the first.

to q_2. Figure 5.3 gives an example of two SQL[2] queries, the second of which contains the first.

Note that query equivalence and containment are *intensional* rather than *extensional* (i.e. they apply to all possible tuples (instances) which conform to the database schema, rather than the actual tuples in the database) – that the first query returns {Monica Seles, Karina Habšudová} while the second query returns {Monica Seles, Iva Majoli, Karina Habšudová} is a necessary but insufficient condition for the fact that the second query contains the first. A query with "`NAT = 'USA' OR NAT = 'SVK'`" as the `WHERE` clause also returns {Monica Seles, Karina Habšudová}, but is neither equivalent to the first query nor contained by the second (to see this, consider adding the tuple (Daniela Hantuchová, SVK, 19830423) to the database in Table 5.1).

In the area of *program equivalence*, there are notions of *operational equivalence* and *operational approximation* [Tal98] – or *bisimulation* and *simulation* [TW96] – which correspond to the notions of semantic equivalence and semantic strength. These will be discussed in Section 5.2.3.4.

[2]Structured Query Language (Chapter 7 of [EN94]).

```
ENTITY person;
  age : INTEGER;
WHERE
  age >= 0;
END_ENTITY;
```

```
ENTITY person;
  age : INTEGER;
WHERE
  age < 18;
END_ENTITY;
```

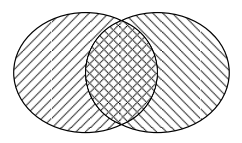


Figure 5.4: Two EXPRESS models with semantically incomparable, overlapping domain rules.

```
ENTITY person;
  age : INTEGER;
WHERE
  age >= 18;
END_ENTITY;
```

```
ENTITY person;
  age : INTEGER;
WHERE
  age < 18;
END_ENTITY;
```

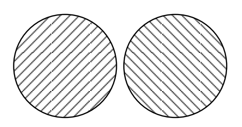


Figure 5.5: Two EXPRESS models with semantically incomparable, disjoint domain rules.

5.1.3 Semantic Incomparability

If it is neither the case that two given constraints are equivalent, nor that one is stronger than the other, then they are said to be *semantically incomparable*. For example, `age >= 0` is incomparable to `age < 18` (Figure 5.4), because satisfaction of neither one implies satisfaction of the other.

Usually, semantically incomparable constraints have some overlap, as depicted in Figure 5.4, but they can be *disjoint*, as in Figure 5.5. This means that it is impossible, in all cases (as opposed to some cases), to map a valid instance from one model to a valid instance in the other. Semantically incomparable constraints that are not disjoint are called *overlapping* (Section 3.3.4 of [Tür99]).

It would be useful to be able to detect disjoint constraints (both in the same model and in another model), but this is beyond the scope of the current CCUS.

5.1.4 Propagation of Semantic Equivalence/Strength

Semantic equivalence and semantic strength propagate from subexpressions to superexpressions as follows:

- Two superexpressions are *equivalent* if and only if all corresponding subexpressions are equivalent. For example, `(age >= 0) AND (age < 130)` is equivalent to `(130 > age) AND (0 <= age)` because `AND` is commutative, `(age >= 0)` is equivalent to `(0 <= age)`, and `(age < 130)` is equivalent to `(130 > age)`.

- One superexpression is *stronger* than another if and only if at least one subexpression is stronger, and no subexpressions are weaker or incomparable. For example, `(age >= 18) AND (age < 65)` is stronger than each of the following expressions:
  - `(age >= 0) AND (age < 65)`;
  - `(age >= 18) AND (age < 130)`;
  - `(age >= 0) AND (age < 130)`.
- One superexpression is *weaker* than another if and only if at least one subexpression is weaker, and no subexpressions are stronger or incomparable. For example, `(age >= 0) AND (age < 130)` is weaker than each of the following expressions:
  - `(age >= 18) AND (age < 130)`;
  - `(age >= 0) AND (age < 65)`;
  - `(age >= 18) AND (age < 65)`.
- Otherwise – if there is a mixture of stronger and weaker subexpressions, or if at least one pair of corresponding subexpressions is incomparable[3] – the superexpressions are *incomparable*. For example, `(age >= 18) AND (age < 130)` and `(age >= 0) AND (age < 65)` are incomparable because `(age >= 18)` is stronger than `(age >= 0)`, but `(age < 130)` is weaker than `(age < 65)`.

To summarise Section 5.1, two constraints are either semantically equivalent, or they are semantically different in one of two ways: either one is semantically stronger than the other, or they are semantically incomparable.

[3]Of course, incomparable subexpressions cannot correspond in commutative expressions, because the semantic equivalence/strength correspondence heuristic (used by the unordered aggregation-matching policy – Section 3.4.1) requires them to be equivalent, or one to be stronger than the other – the clause "corresponding subexpressions are incomparable" applies to subexpressions in fixed positions, e.g. `f(age >= 0, age < 130)` and `f(age < 130, age >= 0)` are incomparable (where `f` is a function whose signature is `(BOOLEAN, BOOLEAN) : BOOLEAN`).

5.2 Comparing Expressions

This section discusses techniques for comparing expressions, and assessing their semantic equivalence and relative[4] semantic strength. This section is divided according to three classes of expressions: *Boolean expressions* (Section 5.2.1), *Boolean-valued expressions* (Section 5.2.2) and *general expressions* (Section 5.2.3).

5.2.1 Boolean Expressions

A *Boolean expression* is an expression that takes only Boolean values (true or false) as its inputs, and evaluates to a Boolean value. Thus, the only operators in a Boolean expression are the *Boolean connectives* (`AND` ($\wedge$), `OR` ($\vee$), `NOT` ($\neg$), and others such as `NAND`, `NOR`, `XOR` and `IMPLIES` ($\rightarrow$) which are defined in terms of those three primitives).

Expressions which contain Boolean connectives, but take non-Boolean inputs, can be considered as Boolean expressions by treating the non-Boolean subexpressions as atomic variables. For example, `(age >= 0) AND (age < 130)` (which takes as input a numeric value for `age`) can be considered as a Boolean expression x `AND` y, where x and y are the truth-values to which `(age >= 0)` and `(age < 130)` evaluate.

5.2.1.1 Truth-Tables

One method of assessing the semantic equivalence and strength of Boolean expressions is by comparing truth-tables.[5] For example, $x \wedge y$ is equivalent to $\neg(\neg x \vee \neg y)$ because they have the same truth-value as each other for every row in the truth-table, as shown in Table 5.2.

Truth-tables can also be used to show that one Boolean expression is semantically stronger than another: for every row in which the stronger expression is true, the weaker expression is true – but there are rows in which the weaker expression is true but the stronger expression is false. The stronger expression is more constrained, so it is true in fewer situations.

[4] The word 'relative' here is to emphasise that semantic strength is a notion of one expression being stronger than another expression – there is no notion of a single expression having an absolute semantic strength.

[5] Also known as *proof by perfect induction* – cf. Section 5.2.3.3.

x	y	$x \wedge y$	$\neg(\neg x \vee \neg y)$
false	false	false	false
false	true	false	false
true	false	false	false
true	true	true	true

Table 5.2: Truth-table to show the semantic equivalence of two Boolean expressions.

x	y	z	$x \wedge y \wedge z$	$x \wedge y$
false	false	false	false	false
false	false	true	false	false
false	true	false	false	false
false	true	true	false	false
true	false	false	false	false
true	false	true	false	false
true	true	false	false	true
true	true	true	true	true

Table 5.3: Truth-table to show that one Boolean expression is semantically stronger than another.

- For example, Table 5.3 shows that $x \wedge y \wedge z$ is stronger than $x \wedge y$ – a conjunction is *strengthened* by adding conjuncts. (The weakest of all Boolean expressions is *true*, which can be thought of as a conjunction with no conjuncts.)

- Conversely, a disjunction is *weakened* by adding disjuncts: Table 5.4 shows that $x \vee y \vee z$ is weaker than $x \vee y$. (The strongest of all Boolean expressions is *false*, which can be thought of as a disjunction with no disjuncts.)

The implemented CCUS can assess the relative semantic strength of conjunctions and of disjunctions, but it does not use truth-tables – Section 5.3 will explain how.

Two Boolean expressions are semantically incomparable if one is sometimes true when the other is false, and sometimes false when the other is true. For example, x and $\neg x$ are semantically incomparable, and in fact disjoint.

The major drawback of using truth-tables to assess semantic equivalence/strength is combinatorial explosion – a truth-table for a Boolean expression with n variables has 2^n rows.

x	y	z	$x \vee y \vee z$	$x \vee y$
false	false	false	false	false
false	false	true	true	false
false	true	false	true	true
false	true	true	true	true
true	false	false	true	true
true	false	true	true	true
true	true	false	true	true
true	true	true	true	true

Table 5.4: Truth-table to show that one Boolean expression is semantically weaker than another.

5.2.1.2 Boolean Algebra

An alternative to truth-tables is to use *Boolean algebra* (see, for example, Section 2.4 of [Cle91]). This entails rewriting Boolean expressions using *axioms*, which are predefined equivalences such as $x \wedge y \equiv y \wedge x$.[6] Once two expressions have been proved to be equivalent using axioms, this equivalence is a *theorem*, which may in turn be used to prove the equivalence of more complex expressions. In fact, making previously-proved theorems available for future reasoning is an application of *case-based reasoning* [Kol93] (Section 2.3.3).

e.g. To prove that $x \vee (x \wedge y) \equiv x$, Boolean algebra can be used to rewrite one into the other, using the axioms $x \wedge true \equiv x$, $x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$, and $true \vee x \equiv true$:

$$\begin{aligned} x \vee (x \wedge y) &\equiv (x \wedge true) \vee (x \wedge y) \\ &\equiv x \wedge (true \vee y) \\ &\equiv x \wedge true \\ &\equiv x \end{aligned}$$

Boolean algebra is used either to prove the equivalence of two Boolean expressions, or to rewrite a Boolean expression into another form, which is semantically equivalent by construction (since the axioms are equivalence-preserving transformations). In particular, Boolean algebra is used to *simplify* expressions. One application of this is in *digital logic circuits*. Such a circuit can be represented by

[6] These axioms could always be proved using truth-tables, which is tractable since each axiom involves at most three variables.

a Boolean expression, which can be simplified in order to minimise the number of logic gates for speed and economy (Section 2.3 of [Cle91]). It is also common to rewrite the Boolean expression such that the circuit uses only NAND gates or only NOR gates,[7] because it is usually faster and cheaper to use only one type of gate (Section 2.4.3 of [Cle91]).

Because axioms and theorems can be applied to subexpressions as well as top-level expressions, Boolean algebra does not suffer the combinatorial explosion problem of truth-tables, which would have to be applied to whole expressions at once, were it not for Boolean algebra.

However, Boolean algebra does have the drawback of having to choose which axiom or theorem to apply at each step, and that it is not always obvious (to a human) that an expression is in its simplest form. Simplifying an expression does not merely entail simplifying subexpressions – as the above example illustrates, subexpressions may actually have to be rewritten as *more complex* expressions before the superexpression can be simplified. So the search space of rewrite sequences branches out, threatening combinatorial explosion – a threat which could be alleviated by a good library of theorems, and by heuristics about when it is appropriate to rewrite a subexpression to be more complex. These issues will be discussed further in Section 6.1.

5.2.1.3 Semantic Strength

Boolean algebra could be extended with axioms for semantic strength as well as equivalence, but it would not be as simple as replacing equals by equals. The axioms would not be bidirectional equivalences, and their application would have to respect the ways that semantic strength can propagate from subexpressions to superexpressions as explained in Section 5.1.4.

Semantic strength of Boolean expressions appears in various guises throughout computer science. The most basic example is the *implication operator*, $x \rightarrow y$, which is logically equivalent to $\neg x \vee y$. This is a fundamental building block of *rule-based systems* (which use inference rules of the form `IF x THEN y`) (Chapter 7 of [Win92] or Chapter 6 of [RK91]), and of logic programming languages such as Prolog [CM94] (which use *Horn clauses* of the form `y :- x`).

A related area is *logical arguments* (Section A.2.2 of [Pra94]), where a *conclusion* ψ is inferred from a set of *premises* $\{\phi_1 \ldots \phi_n\}$ (where the premises and the

[7] Any Boolean expression can be operationalised using only NAND gates or only NOR gates.

conclusion are all Boolean expressions). An argument is *valid* if and only if the conclusion is true in all circumstances which make all of the premises true.

- $\{\phi_1 \ldots \phi_n\} \vdash \psi$ means that the argument can be shown to be valid using a given proof strategy;
- $\{\phi_1 \ldots \phi_n\} \models \psi$ means that it really is valid.

One such proof strategy is *resolution* (Section A.2.3 of [Pra94]), which is known to be:

- *sound*: if $\{\phi_1 \ldots \phi_n\} \vdash \psi$ then $\{\phi_1 \ldots \phi_n\} \models \psi$;
- *complete*: if $\{\phi_1 \ldots \phi_n\} \models \psi$ then $\{\phi_1 \ldots \phi_n\} \vdash \psi$.

Clearly, then, one could test whether any Boolean expression ϕ is semantically stronger than another Boolean expression ψ by asking a (resolution) theorem-prover whether $\{\phi\} \vdash \psi$, and one could also test for equivalence ($\{\phi\} \vdash \psi$ and $\{\psi\} \vdash \phi$).

Section 9.3.1 of [Pra94] defines *logical strength* as follows: "if ϕ and ψ are formulae such that $\phi \vdash \psi$, but not $\psi \vdash \phi$, we say that ϕ *is logically stronger than* ψ, and that ψ *is logically weaker than* ϕ." The application here is *learning by induction* (learning general rules from examples, where the *examples* are effectively incomplete truth-tables, and the *rules* are Boolean expressions that cover the given examples and are supposed to generalise to unseen examples).

This subsection has considered Boolean expressions, which take Boolean values as inputs and produce a Boolean value as output. The next subsection considers *Boolean-valued expressions*, where the inputs may be of any types.

5.2.2 Boolean-Valued Expressions

A *Boolean-valued expression* is an expression that evaluates to a Boolean value (as does a Boolean expression), but the types of the inputs do not have to be Boolean. Examples of Boolean-valued expressions are *relational expressions* (e.g. in `age < 130`, the input `age` is of a numeric type), and – to generalise – all *predicates* (any function that takes n input values, and returns a Boolean value, is an n-ary predicate).

The relationship of Boolean-valued expressions to Boolean expressions is somewhat analogous to the relationship of predicate calculus to propositional calculus.

```
SCHEMA test;
  ENTITY everything;
    man : BOOLEAN;
    mortal : BOOLEAN;
    married : BOOLEAN;
  END_ENTITY;

  RULE all_men_are_mortal FOR (everything);
  WHERE
    QUERY (x <* everything | NOT x.man OR x.mortal) = everything;
  END_RULE;

  RULE some_men_are_married FOR (everything);
  WHERE
    SIZEOF (QUERY (x <* everything | x.man AND x.married)) >= 1;
  END_RULE;
END_SCHEMA;
```

Figure 5.6: Universal and existential quantification in EXPRESS.

- *Propositional calculus* (Section A.2 of [Pra94]) has a *syntax* consisting of Boolean values and Boolean connectives, a *semantics* (Boolean algebra), and a *proof strategy* (as mentioned in Section 5.2.1.3).

- *Predicate calculus* (Section A.3 of [Pra94]) extends propositional calculus with the *universal quantifier* ($\forall$) and the *existential quantifier* ($\exists$). For example, in $(\forall x.man(x) \rightarrow mortal(x))$ – "all men are mortal" – and $(\exists x.man(x) \wedge married(x))$ – "some men are married" – $man(x)$, $mortal(x)$ and $married(x)$ are predicates, whose inputs are values from the universe with respect to which these two Boolean-valued expressions (which can be thought of as taking as input a set representing said universe) are interpreted. Figure 5.6 shows how these two examples can be written in EXPRESS (Appendix A, Section A.6).

Two Boolean-valued expressions are *equivalent* if and only if they are true in exactly the same set of situations. One Boolean-valued expression (σ) is stronger than another (ω) if and only if ω is true in all situations that make σ true but not vice versa (i.e. the set of situations that make σ true is a proper subset of the set of situations that make ω true).

The use of truth-tables to assess equivalence and strength does *not* generalise to Boolean-valued expressions, because the set of possible input values can be infinite (whereas each variable in a Boolean expression has only two possible values, so it is possible to enumerate the output values for all possible situations).

Semantic equivalence of Boolean-valued expressions can be reasoned about using mathematical and logical laws, of course, and the resolution theorem-proving method mentioned in Section 5.2.1.3 does generalise to the predicate calculus (Section A.3.3 of [Pra94]). Theorem-proving is used by, for example, [GKD97] (Section 2.4.3), [Tal98] (Section 5.2.3.4) and [ZW97] (Section 5.2.2.4).

5.2.2.1 Semantic-Strength Cases

As will be described in Section 5.3, the CCUS uses *semantic-strength cases* to assess the relative semantic strength of:

- comparators (cf. Section 5.1), e.g.
  - `(a > 5)` is equivalent to `(5 < a)`,
  - `(a > 10)` is stronger than `(a > 5)`,
  - `(a < 10)` is weaker than `(a < 5)`,
  - `(a > 5)` is incomparable to `(a < 5)`;
- the bounds of aggregation types, e.g.
  - `SET [1 : ?]` is stronger than `SET [0 : ?]` (a set that must have one-or-more elements can always be mapped to a set that must have zero-or-more elements, but an empty set cannot be mapped to a set that must have one-or-more elements),
  - `SET [0 : 9]` is stronger than `SET [0 : 10]` (a set that can have at most nine elements can always be mapped to a set that can have at most ten elements, but not necessarily vice versa),
  - `SET [1 : 9]` is stronger than `SET [0 : 10]` (both the lower bound and the upper bound of the first set type are stronger than the corresponding bounds in the second set type, so the conjunction of the bounds is stronger in the first set type),
  - `SET [0 : 9]` is incomparable to `SET [1 : 10]` (the first lower bound is the weaker, but the first upper bound is the stronger, so the conjunctions of the bounds in the two set types are incomparable);
- instances of the `choice_set` Higher-Level Constraint (HLC) (Appendix A, Section A.1), e.g.

- {`masculine, feminine`} is equivalent to {`feminine, masculine`} (the order of elements in a set does not matter),
- {`masculine, feminine`} is stronger than {`masculine, feminine, neuter`} (neuter objects cannot be mapped to a model that does not allow neuter, but all objects can be mapped from the first model to the second),
- `OPTIONAL` {`masculine, feminine`} is weaker than {`masculine, feminine`} (null cannot be mapped to a model that does not allow null, but non-null can always be mapped to a model that does),
- {`masculine, feminine`} is incomparable to {`masculine, neuter`}.

Of course aggregation bounds and HLCs are not expressions in the syntactic sense of the word, but all constraints can be thought of as Boolean-valued expressions, because they are either satisfied or violated.

Can Türker [Tür99] studies integrity constraints in the context of federated database schemata. A *federated database* consists of several local schemata and one global schema.

- If the constraints in the local schemata are not properly reflected in the global schema, then a global database operation may be rejected because it violates local integrity constraints, which are invisible to a global user (the problem of *global understandability*). The solution is a correct integration of local integrity constraints during schema integration (Section 2.2.4). This entails transferring constraints from the local schemata to the global schema, and if the local schemata have corresponding constraints (such as `salary > 5000` and `salary > 2000`), then the weakest constraint (`salary > 2000`) is transferred, to ensure that the integrity constraints of at least one local schema will be satisfied, without imposing unnecessarily strong constraints on the global schema.

- If the constraints in the global schema cannot be derived from the constraints in the local schemata, then a local database operation may be rejected because it violates the global integrity constraints, which are invisible to the local user (the problem of *local understandability*). The global schema has *extensional assertions* stating, for example, that two local classes are extensionally equivalent (a human judgement), in which case corresponding local integrity constraints may have to be strengthened or weakened to make them semantically equivalent (e.g. `salary > 2000` may be strengthened to make it equivalent to `salary > 5000`, a constraint in another local schema).

Hence Türker's work entails assessing the semantic equivalence and relative semantic strength of constraints. He presents algorithms for computing the relationship between two given constraints (the relationship between any two constraints is either equivalence, containment (strength), overlap or disjointness – the two subclasses of what this thesis calls semantic incomparability (Section 5.1.3)). He presents such constraint-relationship algorithms for various classes of restricted constraints (since the constraint-relationship problem is undecidable in its full generality), such as linear arithmetic constraints (cf. Appendix A, Section A.2), uniqueness constraints (cf. Section A.7), and functional dependences.

Türker's assessment of semantic strength appears to be more complete than that of the CCUS (i.e. to handle a wider range of constraints), but it is achieved using hard-wired algorithms rather than data-driven SE/S cases, and does not use any techniques of assessing semantic equivalence and strength at higher levels of abstraction (Chapter 6).

5.2.2.2 Behavioural Subtyping

One area which involves semantic equivalence and semantic strength is *behavioural subtyping* in *object-oriented programming* (Section 11.1.2 of [Mey88]). Objects have *methods*: functions that can be called by the methods of other objects (or by other methods within the object itself). Each method can be specified by a precondition and a postcondition:

- The *precondition* specifies what condition(s) must be true before the method can be called (constraints on the input). It is the client's responsibility to ensure that the precondition is true, and the method may assume it to be true (without testing).

- The *postcondition* specifies what condition(s) must be true when the method returns (constraints on the output). The method must ensure that the postcondition is true – given that the precondition is true.

Each object is an instance of a *class*, and one class can be a subtype (subclass) of another, which means that it inherits the attributes and methods of its supertype (superclass). The subtype can add new attributes and methods, and it can redefine the inherited ones.

Behavioural subtyping guarantees that an object of the subtype can always be used in place of an object of the supertype (without the calling code having to know about the subtype), by constraining the ways in which methods can be redefined:

- The *precondition* of a redefined method must be semantically *weaker* or equivalent to its precondition in the supertype. This means that if the precondition is satisfied for the supertype then it is satisfied for the subtype – the client does not have to ensure any extra preconditions for the subtype.
- The *postcondition* of a redefined method must be semantically *stronger* or equivalent to its postcondition in the supertype. This means that the client will get at least as much effect from calling the subtype method as from calling the supertype method.

Most object-oriented programming languages do not include preconditions and postconditions explicitly – they are either omitted, specified in the design but not the implementation, or checks can even be inserted by hand. Eiffel [Mey92], however, includes preconditions and postconditions in the language itself (`require` and `ensure` clauses, respectively). For example, consider a method to compute iteratively the square root of x to within a specified precision ϵ, which cannot handle values of ϵ smaller than 10^{-6}:

```
sqrt (x : REAL, epsilon : REAL) is
  require
    epsilon >= 10^(-6)
  do
    ...
  ensure
    |sqrt^2 - x| <= epsilon
end sqrt
```

Suppose a better algorithm is devised, which can now handle values of ϵ down to 10^{-20}, and furthermore guarantees a precision of $\epsilon/2$, and it is decided to redefine the above method in a subtype:

```
sqrt (x : REAL, epsilon : REAL) is
  require
    epsilon >= 10^(-20)
  do
    ...
  ensure
    |sqrt^2 - x| <= epsilon/2
end sqrt
```

The precondition has been *weakened* ($\epsilon \geq 10^{-20}$ is weaker than $\epsilon \geq 10^{-6}$), and the postcondition has been *strengthened* ($|sqrt^2 - x| \leq \epsilon/2$ is stronger than $|sqrt^2 - x| \leq \epsilon$).

A call to the supertype method would be handled either by the subtype method (if the object on which it is called is an instance of the subtype), or by the supertype method itself (if the object on which it is called is an instance of the supertype but not of the subtype). The calling code does not need to know whether it is an instance of the subtype, only that it is an instance of the supertype.

- A call which is handled by the subtype method would benefit from the stronger postcondition.
- To take advantage of the weaker precondition, the subtype method must be called directly on an instance of the subtype.

According to Section 11.1.3 of [Mey88], Eiffel does not check at compile-time for weaker preconditions and stronger postconditions – it merely checks preconditions and postconditions at run-time. That is to say, it cannot detect invalid subtypes – it can only detect when assertions are violated as a result. Such compile-time checking could be achieved using techniques for assessing semantic equivalence/strength such as those discussed in this chapter.

5.2.2.3 Semantics of Programming Languages

Another area to which semantic equivalence and semantic strength are related is the formal semantics of programming languages (Chapter 12 of [Lou93]). Three kinds of formal semantics are:

- *operational semantics* (which specify how programs are to be executed);
- *denotational semantics* (which use functions to specify semantics);
- *axiomatic semantics* (which specify semantics in terms of preconditions and postconditions).

In axiomatic semantics (Section 12.4 of [Lou93]), a language construct (or a particular statement, or a program) S can be specified with a *Hoare triple* $\{P\}\ S\ \{Q\}$, which means that if precondition P is true just before executing S, then postcondition Q will be true just after executing S.

However, such a specification may be 'loose', in that the precondition may be unnecessarily strong, or the postcondition unnecessarily weak (i.e. if it understates what S achieves).

To ensure that the specification is as tight as possible, a function wp can be used to derive the *weakest precondition* for S and a given postcondition Q, $wp(S, Q)$.[8] For example, $wp(x := x + 1, x > 1) = x > 0$.

5.2.2.4 Specification Matching

Moormann Zaremski and Wing's work on *specification matching* [ZW97] entails comparing two software components, based on descriptions of their behaviours (as pre-and-postcondition specifications in first-order logic – cf. Section 1.2).[9] The purpose of comparing specifications relates to *software reuse* (Section 2.1): to assess when one component can be substituted for (i.e. is a behavioural subtype of) another (Section 5.2.2.2), or adapted to meet other requirements.[10] Their method performs a *signature match* (comparing the input and output types of the functions) followed, if the signature match succeeds, by a *specification match* (comparing preconditions and postconditions).

There are various kinds of *relaxed match*, appropriate for different situations, and they are all defined in terms of logical equivalence ($\Leftrightarrow$) and implication ($\Rightarrow$), and performed using a theorem-prover:

- *Pre/post matches* (comparing preconditions with preconditions, and postconditions with postconditions, separately):
  - *Exact pre/post match*: $(Q_{pre} \Leftrightarrow S_{pre}) \wedge (S_{post} \Leftrightarrow Q_{post})$ – this is used when components Q and S are required to be behaviourally equivalent and thus completely interchangeable.
  - *Plug-in match*: $(Q_{pre} \Rightarrow S_{pre}) \wedge (S_{post} \Rightarrow Q_{post})$ – S can be substituted for (is a behavioural subtype of) Q, but not necessarily vice versa.
  - *Plug-in postmatch*: $(S_{post} \Rightarrow Q_{post})$ – only concerned with the effects of functions.

[8] Equally, a function to derive the *strongest postcondition* for S and a given precondition P could be used, but it is nicer to have to provide a postcondition than a precondition.

[9] This relates to the notion of *observational equivalence* [Tal98].

[10] Adaptability implies *comparative code generation* (generating code to bridge the semantic gap revealed by comparison – Section 1.3).

  - *Guarded plug-in match*: $(Q_{pre} \Rightarrow S_{pre}) \wedge ((S_{pre} \wedge S_{post}) \Rightarrow Q_{post})$ – for when the postcondition relation only holds when the precondition does.
  - *Guarded postmatch*: $((S_{pre} \wedge S_{post}) \Rightarrow Q_{post})$ – a combination of plug-in postmatch and guarded plug-in match.

- *Predicate matches* (comparing preconditions and postconditions with preconditions and postconditions together) – let $S_{pred} = S_{pre} \Rightarrow S_{post}$:
  - *Exact predicate match*: $(S_{pred} \Leftrightarrow Q_{pred})$ – this is weaker than exact pre/post match, because the preconditions and postconditions can interact.
  - *Generalized match* (sic): $(S_{pred} \Rightarrow Q_{pred})$ – this is useful for retrieving specifications from a library: it allows for simple queries which retrieve stronger (more generic) specifications.
  - *Specialized match* (sic): $(Q_{pred} \Rightarrow S_{pred})$ – this allows for queries which retrieve weaker (more specific) specifications.

As well as *function matching*, Moormann Zaremski and Wing also support *module matching* for comparing collections of functions. This could be done by inferring correspondences between the functions in two modules using the function matches given above, thus using semantic equivalence/strength as a correspondence heuristic like the CCUS (Section 4.2.5), but instead they require the user to provide the correspondences between functions.

5.2.2.5 Database Queries: Equivalence and Containment

As mentioned in Section 5.1, there is a duality between constraints and database queries. In fact, a query can be thought of as a Boolean-valued expression, which evaluates to true for just those instances which are included in the set of results. Hence query equivalence and containment belong to this subsection.

Chaudhuri and Vardi [CV94] give some complexity results for the equivalence and containment of various classes of restricted Datalog[11] programs. Their underlying motivation is query optimisation by *query rewriting*, i.e. "to transform a query into an equivalent one that can be evaluated more efficiently. Therefore,

[11] Datalog is a logic-programming language like Prolog, without function symbols.

determining equivalence of queries is one of the most fundamental optimisation problems."

The main theme of [CV94] is equivalence between recursive and non-recursive Datalog programs. Query ψ contains query θ (i.e. θ is semantically stronger than or equivalent to ψ) if and only if a containment mapping from ψ to θ can be constructed. A *containment mapping* is a renaming of the variables of ψ where, for *conjunctive queries*, every literal in ψ must be among the literals of θ (as noted in Section 5.2.1.1, a conjunction is strengthened by adding conjuncts, so θ has all the conjuncts of ψ and possibly more).

5.2.3 General Expressions

This section would not be complete without a discussion of general expressions (of which Boolean-valued expressions are a subclass – another important subclass is arithmetic expressions), where the inputs and outputs can be of any types.

The notion of semantic equivalence generalises to all expressions – two expressions are *semantically equivalent* if and only if they always evaluate to the same value as each other for the same input values, e.g.

- $\log(x \times y)$ is equivalent to $\log x + \log y$;
- `length(x @ y)` is equivalent to `length x + length y` (i.e. the `length` function respects concatenation).

Assessing the semantic equivalence of general expressions is in fact relevant to Boolean-valued expressions, which can have general expressions as subexpressions. In particular, constraints in EXPRESS are always either satisfied or violated, but although the top-level expression of a domain rule has to evaluate to a logical value (true, false or unknown), the subexpressions can be any expressions that are expressible in EXPRESS, which includes a wide range of arithmetic functions, aggregation functions, and so on.

The notion of semantic strength does *not* generalise to all expressions, although it is not entirely true to say that it only makes sense for Boolean-valued expressions (where ϕ is stronger than ψ if ψ is true whenever ϕ is true but not vice versa). There *can* be related notions for other classes of expressions. For example, with arithmetic expressions it is sometimes possible to infer whether one expression is always greater than another, which in turn can be used to assess the relative semantic strength of two Boolean-valued superexpressions – a generalisation of inferring that `(age >= 21)` is stronger than `(age >= 18)` because 21 is (always) greater than 18.

5.2.3.1 Standard Forms

Mathematica[12] [Wol91] is an extensive mathematics program which can manipulate algebraic expressions. Of particular interest is the way that it reduces expressions to *standard forms* (also known as *normal forms* or *canonical forms*), such that some pairs of semantically equivalent expressions are reduced to syntactically identical expressions. For example:

- *Mathematica* reduces associative operations (such as addition and multiplication) to flat lists, as does the CCUS – e.g. $(x + y) + z$ is equivalent to $x + (y + z)$, so they can both be written as $x + y + z$ with no semantic ambiguity (the order in which the additions are performed does not affect the result).

- *Mathematica* reduces commutative operations (such as addition and multiplication) to orderless bags, as does the CCUS – e.g. $x + y$ is equivalent to $y + x$, so they can both be represented by the bag $\{x, y\}$.

- *Mathematica* reduces polynomials to sums of terms – e.g. $x(x+2)$ is reduced to $x^2 + 2x$.

- *Mathematica* removes identity operations such as adding zero or multiplying by one.

Other standard forms include:

- *conjunctive normal form* and *disjunctive normal form* for Boolean expressions;

- *first normal form*, *second normal form*, *third normal form* and *Boyce-Codd normal form* for relational databases (Chapter 12 of [EN94]).

An obvious decision procedure for the semantic equivalence of two expressions is thus to reduce the two expressions to a standard form and compare them syntactically. Such a decision procedure is sound (two expressions are semantically

[12] *Mathematica* is a registered trademark of Wolfram Research, Inc.

equivalent if they are syntactically identical), but could never be complete: it is not the case that *any* pair of expressions could be reduced to a standard form such that they are semantically equivalent if and only if they are syntactically identical (Section 2.5.2 of [Wol91]). Only certain restricted classes of expressions – such as those listed above – can be reduced to a standard form.

Reducing expressions to standard forms to show that they are equivalent is like extracting higher-level knowledge from two code-fragments to show that they are equivalent at an abstract level (Chapter 6). Higher-level knowledge can be extracted from structure as well as from expressions, however.

5.2.3.2 Propagation via Constraints

An area related to standard forms is *propagation via constraints* (Chapter 15 of [FdK93]). Here, arithmetic constraints are connected together to form *constraint networks* (Figure 5.7), which are made up of *cells* (values), *primitive constraints* (adders and multipliers) and *defined constraints* (existing constraint networks). The idea is that unknown values can be derived when enough of the surrounding values are known, and can thus propagate through the constraint network in any direction.

These constraints are not like the integrity constraints considered in this thesis, because they are so tight that they can only be satisfied by a single value – tight enough for unknown values to be derived from known values.

Constraint networks are like standard forms in the way that they abstract away from which way round equations are written (e.g. $x + y = z$, $x = z - y$ and $y = z - x$ are all equivalent, and are all represented as an adder with addends x, y and sum z). Inspired by constraint networks, the HLC CCUS (Chapter 6) can actually extract `adder` and `multiplier` HLCs to show that such equations are equivalent (Appendix A, Section A.3).

5.2.3.3 Proof by Induction

Proof by induction (Section 0.6 of [FB94]) is an important technique for proving the equivalence of two expressions, which is applicable only to discrete-typed expressions (such as natural numbers, integers, lists and trees – but not continuous types such as real numbers). It works by showing that the equivalence holds for a particular value (the *base case*), and then proving that if it holds for one value then it holds for the next value (the *inductive case*).

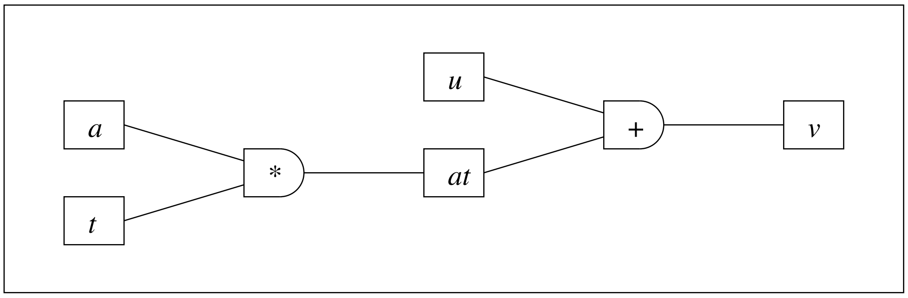


Figure 5.7: A constraint network, representing the equation $v = u + at$.

For example, it can be proved that $\sum_{0 \le i \le n} i \equiv \frac{1}{2}n(n+1)$:

- **Base case** ($n = 0$):

$$\sum_{0 \le i \le 0} i = 0$$

$$\tfrac{1}{2}0(0+1) = 0$$

- **Inductive case** (if it holds for n then it holds for $n+1$):

$$\begin{aligned}\sum_{0 \le i \le n+1} i &= \sum_{0 \le i \le n} i + (n+1) \\ &= \tfrac{1}{2}n(n+1) + (n+1) \\ &= \tfrac{1}{2}n^2 + \tfrac{1}{2}n + (n+1) \\ &= \tfrac{1}{2}n^2 + 1\tfrac{1}{2}n + 1\end{aligned}$$

$$\begin{aligned}\tfrac{1}{2}(n+1)((n+1)+1) &= (\tfrac{1}{2}n + \tfrac{1}{2})(n+2) \\ &= \tfrac{1}{2}n^2 + 1\tfrac{1}{2}n + 1\end{aligned}$$

Note that this proof strategy uses as a building block the fundamental technique of reducing two expressions to two syntactically identical expressions.

5.2.3.4 Operational Equivalence

Carolyn Talcott [Tal98] defines the terms *operational equivalence* and *operational approximation* as follows:

- "Intuitively, two expressions are operationally equivalent (also called contextually equivalent in the literature) if no program context can distinguish them." This means that they will always evaluate to the same value as each other for a given program state (i.e. an assignment of values to variables). This corresponds to the notion of semantic equivalence.

- "One expression operationally approximates another expression if when placed in any closing program context either the resulting first program is undefined or both resulting programs are defined" – i.e. if the first program is defined then the second program is defined. Two expressions are operationally equivalent ($\phi \cong \psi$) if and only if they operationally approximate each other ($\phi \sqsubseteq \psi \wedge \psi \sqsubseteq \phi$). This roughly corresponds to the notion of semantic strength – 'ϕ approximates ψ' ($\phi \sqsubseteq \psi$) is like saying that ϕ is semantically stronger or equivalent to ψ. However, operational approximation is not limited to Boolean-valued expressions, but applies to general expressions.

Talcott presents an operational semantics (Section 5.2.2.3) for a λ-language (a functional programming language consisting of λ-calculus expressions augmented with primitive operations). This entails defining a set of equivalence laws (such as `fst(pr(x,y))` $\cong$ `x` and `snd(pr(x,y))` $\cong$ `y`), which can even handle functions with effects (e.g. `set(x,v); get(x)` $\cong$ `set(x,v); v`). This enables reasoning about program equivalence by replacing equals by equals, and implies a theorem-proving implementation.

This section has discussed techniques used to assess semantic equivalence and semantic strength (of Boolean expressions, Boolean-valued expressions and general expressions), and has reviewed literature in related areas. The next section explains in detail how the CCUS uses semantic-equivalence/strength cases to assess semantic equivalence and semantic strength.

5.3 Semantic-Equivalence/Strength Cases

As mentioned in Section 3.3, when the CCUS encounters a difference (when comparing in plausible correspondence mode), it attempts to resolve it at the current abstraction level using *semantic-equivalence/strength (SE/S) cases*, which are the subject of this section – if these fail to sanction correspondence between the two frames being compared, then the CCUS attempts to infer that they correspond at a more abstract level, by extracting and comparing HLCs (Chapter 6).

When thus using semantic equivalence/strength as a correspondence heuristic (Section 4.2.5), the CCUS infers that two frames correspond if they are semantically equivalent, or if either one is stronger than the other. Only if they are semantically incomparable does it veto the correspondence.

This policy for inferring correspondences makes sense in the context of schema-to-schema mapping (Section 1.1) because:

- Instances can be mapped in at least one direction without violating constraints, when the constraints on the target schema are weaker than or equivalent to those on the source schema.

- Also, the constraints may be partially satisfied in the other direction, meaning that constraints which are stronger on the target schema may only need to be transferred *partially* to the source schema (Section 1.3). For example, when mapping from a source schema with `age >= 0` to a target schema with {`0 <= age < 130`}, only `age < 130` needs to be transferred to the source schema – the source instances do not need to be checked, prior to mapping, for `age >= 0` because that constraint already holds on the source schema.

The CCUS uses SE/S cases to attempt to infer that at least one of the constraints implies the other, and if it fails to do so, it assumes that they are incomparable (the CCUS is sound but incomplete in its assessment of semantic equivalence/strength).

The CCUS retrieves at most one SE/S case[13] for each pair of frames for which it encounters a difference. An SE/S case specifies two sets of slots, whose fillers in the frames being compared affect the semantic equivalence of those frames:

- *necessary slots*: these slots *must* match – if the two frames have different fillers for these slots, then the CCUS fails the comparison of these frames and retracts the plausible correspondence between them (see Section 4.2.5);

- *semantic-strength slots*: these are the slots that determine which expression is semantically stronger.

[13] All the examples considered require no more than one SE/S case. It is conceivable that examples which require multiple SE/S cases may arise in the future, but they are beyond the scope of the current implementation.

Note that SE/S cases also sanction correspondences to be inferred between frames that would otherwise fail to correspond, because they specify by omission *unnecessary slots* (such as which way round a comparator is written), differences in which are permitted.

Semantic-strength slots can be numeric, Boolean or aggregate (or any type with an order relation, in principle), and can affect semantic strength either *positively* (the stronger frame has a greater value for this slot) or *negatively* (the stronger frame has a lesser value for this slot).

For example, `(age >= 21)` is stronger than `(age >= 18)` because $21 > 18$. To generalise, let x be a variable (such as an attribute), and let c_1 and c_2 be constants. Then the SE/S cases for *comparators* are as follows:

- Case 1: $x > c_1$ is stronger than $x > c_2$ if and only if $c_1 > c_2$;
- Case 2: $x \geq c_1$ is stronger than $x \geq c_2$ if and only if $c_1 > c_2$;
- Case 3: $x < c_1$ is stronger than $x < c_2$ if and only if $c_1 < c_2$ (e.g. `(age < 60)` is stronger than `(age < 65)`);
- Case 4: $x \leq c_1$ is stronger than $x \leq c_2$ if and only if $c_1 < c_2$.

In Cases 1 and 2, the relative values of the constants affect semantic strength positively; in Cases 3 and 4, they affect semantic strength negatively.

The reason that $>$ and $\geq$ have to be treated as separate cases is that $x > c$ is stronger than $x \geq c$ if it is a real-number comparison, and $x > c$ is equivalent to $x \geq c + 1$ if and only if it is an integer comparison. Similarly, $x < c$ is stronger than $x \leq c$ if it is a real-number comparison, and $x < c$ is equivalent to $x \leq c - 1$ if and only if it is an integer comparison. These complications are beyond the scope of the implemented CCUS, which requires the comparator operators to be the same, otherwise they are inferred to be semantically incomparable.

There are, of course, twelve further cases based on the fact that $a > b$ is always equivalent to $b < a$, and $a \geq b$ to $b \leq a$ (regardless of whether each of a and b is a variable or a constant): Cases 1 to 4 are replicated for writing the comparator in the first model the other way round, for writing the comparator in the second model the other way round, and for writing the comparators in both models the other way round.

The EXPRESS frame-class model (Appendix D) facilitates the assessment of semantic equivalence and strength in the way that it represents comparators:

```
ENTITY expression ABSTRACT SUPERTYPE
                  SUBTYPE OF (frame);
  tipe : abstract_type; -- "type" is a reserved word in EXPRESS
  is_variable : BOOLEAN;
END_ENTITY;

ENTITY comparator SUBTYPE OF (expression);
  greater : expression;
  lesser : expression;
  inclusive : inclusive_or_exclusive;
  discrete : discrete_or_continuous;
  which_way_round : way_round;
END_ENTITY;
```

This is a semantics-oriented representation (cf. Section 3.2.2, and the discussion of standard forms in Section 5.2.3.1), which is much more flexible in enabling semantic equivalence to be seen than the obvious syntax-oriented representation. It allows the sixteen SE/S cases for comparators to be encoded as just two FIL-C cases (see Appendix B, Section B.1 for an explanation of the FIL-C case representation and case matching):

- one for which the variable is greater than the constant (see Figure 5.8);
- one for which the variable is less than the constant (the same as Figure 5.8 except for the words in bold face: `true` becomes `false`, `false` becomes `true`, `greater` becomes `lesser`, `lesser` becomes `greater`, and `positive` becomes `negative`).

The slot `is_variable` represents whether each expression is constant or variable (i.e. different for different instances). For example, literals are regarded as constants, and attributes are regarded as variables.

Figure 5.8 encodes Case 1 and Case 2 as a single FIL-C case by specifying `inclusive` as a necessary slot – the comparators being compared must be either both inclusive or both exclusive for the match to succeed. It does not matter which way round the comparator is written, as $a > b$ and $b < a$ are represented by `comparator` frames which are identical (except for the unnecessary slot `which_way_round`, which records which way round they are written).

Given a pair of plausibly-corresponding frames for which a difference has been found, the CCUS retrieves at most one SE/S case whose problem part (and any descendants specified in the problem part – in this case, the two subexpressions `expression#1` and `expression#2`) is matched by the first frame. It then checks whether the SE/S case is matched by the second frame. If both frames match the SE/S case, then it compares the two frames along the slots specified in the

```
(***********************************************************************)
(* semantic-comparator-positive.filc                   2001:04:26 10:47 *)
(***********************************************************************)
(* A case for assessing the semantic equivalence of two comparators for   *)
(* which the value of the limit positively affects the semantic strength. *)
(***********************************************************************)

INSTANCE semantic_case#1;
  language = express;
  description =
  "A case for assessing the semantic equivalence of two comparators";
  problem = comparator#1;
  solution = semantic_match_prescription#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE comparator#1;
  greater = expression#1;
  lesser = expression#2;
END_INSTANCE;

INSTANCE expression#1;
  is_variable = true;
END_INSTANCE;

INSTANCE expression#2;
  is_variable = false;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE semantic_match_prescription#1;
  necessary_slots = {"greater", "inclusive"};
  semantic_strength_slots = {semantic_strength_slot#1};
END_INSTANCE;

INSTANCE semantic_strength_slot#1;
  slot_name = "lesser";
  influence = semantic_strength_influence(positive);
END_INSTANCE;
```

Figure 5.8: A semantic-equivalence/strength case for comparators.

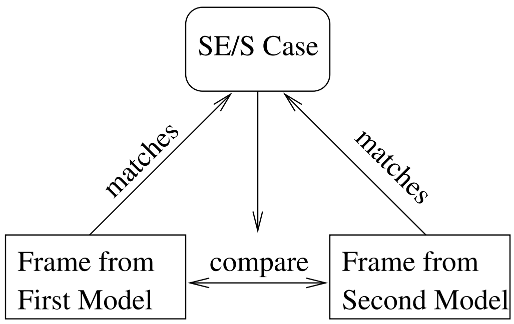


Figure 5.9: SE/S case matching and CCUS frame comparison.

solution part of the case. Please note the two orthogonal types of comparison here: matching the frame-representation of the actual code to an SE/S case to see if it can be applied, and applying a matching SE/S case to compare two frame-networks (Figure 5.9).

If any necessary slots fail to match in the two frames, or if no SE/S case is matched by both frames, then the comparison of the two frames fails, they are deemed to be semantically incomparable, and the correspondence between them is retracted (Section 4.2.5).

The two frames are compared along each semantic-strength slot. First, the fillers of that slot are compared in a type-specific manner to see whether the first filler is 'equal to', 'greater than', 'less than', or incomparable to, the second filler:

- Numbers are compared using the standard comparators $>$ and $<$;
- Logical values are compared using true $>$ unknown $>$ false;[14]
- Unordered aggregations are compared as follows: a 'equals' b if they are equal as sets, a is 'greater than' b if a is a proper superset of b, and they are incomparable if neither contains the other.
- These are the only types currently supported.

If the fillers are 'equal' then the frames are inferred to be semantically equivalent. If one filler is 'greater' than the other, then the frame with the 'greater'

[14]Although true is logically weaker than false (Section 5.2.1.1), in this context true and false are not expressions which are being compared for semantic strength directly, but fillers of semantic-strength slots, which can be positive or negative.

filler is inferred to be semantically stronger than the other frame if the semantic-strength slot has a positive influence, weaker if it has a negative influence. Otherwise, the frames are inferred to be semantically incomparable – the match fails, and the correspondence between the frames is retracted.

Pseudocode algorithms for using SE/S cases are given in Appendix C, Section C.4.

The other SE/S cases implemented are (see Appendix E, Section E.3 for the FIL-C representation of all implemented SE/S cases):

- adding conjuncts strengthens a conjunction (Section 5.2.1.1);
- adding disjuncts weakens a disjunction;
- the lower bound ϕ of one aggregation type is stronger than the lower bound ψ of another aggregation type if and only if $\phi > \psi$ (Section 5.2.2.1);
- the upper bound ϕ of one aggregation type is stronger than the upper bound ψ of another aggregation type if and only if $\phi < \psi$;
- adding attributes weakens a uniqueness rule;
- one `choice_set` HLC, with `choice_set` ϕ, is stronger than another `choice_set` HLC, with `choice_set` ψ, if $\phi \subset \psi$ (Section 5.2.2.1);
- one `choice_set` HLC π is stronger than another `choice_set` HLC ρ if π is mandatory and ρ is optional.

The reason the last two cases say "if" rather than "if and only if" is that they interact with each other according to the way that semantic strength propagates from subexpressions to superexpressions (Section 5.1.4).

The two SE/S cases for `choice_set` HLCs listed above are actually implemented as a single SE/S case with multiple semantic-strength slots (Figure 5.10). This means that it is possible for a pair of frames to receive different assessments of semantic equivalence/strength. If this happens, then strength takes precedence over equivalence, because strength is what counts when all else is equal (Section 5.1.4). If one of the frames is inferred by one semantic-strength slot to be stronger than the other, and by another semantic-strength slot to be weaker than the other, then the frames are inferred to be incomparable because the first frame is stronger in one aspect and weaker in another.

```
(*********************************************************************)
(* semantic-choice_set.filc                           2002:05:08 18:50 *)
(*********************************************************************)
(* A case for assessing the semantic equivalence of two choice_set HLCs. *)
(*********************************************************************)

INSTANCE semantic_case#1;
  language = express;
  description =
  "A case for assessing the semantic equivalence of two choice_sets";
  problem = choice_set#1;
  solution = semantic_match_prescription#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE choice_set#1;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE semantic_match_prescription#1;
  necessary_slots = {"constrained_entity"};
  semantic_strength_slots = {semantic_strength_slot#1,
                             semantic_strength_slot#2};
END_INSTANCE;

INSTANCE semantic_strength_slot#1;
  slot_name = "is_optional";
  influence = semantic_strength_influence(negative);
END_INSTANCE;

INSTANCE semantic_strength_slot#2;
  slot_name = "choice_set";
  influence = semantic_strength_influence(negative);
END_INSTANCE;
```

Figure 5.10: A semantic-equivalence/strength case for `choice_set` HLCs.

SE/S cases are a bit like equivalence laws, but they are unorthodox in that instead of specifying two frame-networks to compare, they specify a single frame-network which both expressions being compared must match (in Figure 5.8, a comparator and two subexpressions), and prescribe which slots to compare (necessary slots and semantic-strength slots) in order to assess semantic equivalence/strength. An SE/S case can thus be thought of as a meta-level prescription (i.e. specified in terms of frame-classes and slots, rather than frames and slot-fillers) of how to compare two particular frame-networks. This representation of SE/S cases makes them much smaller and simpler than they would have to be if they specified two frame-networks to compare.

SE/S cases may seem inflexible in that they require two frame-networks from the frame-representation of the code to match the frame-network in the problem part of the case, but it is not the purpose of SE/S cases to compare non-matching frame-networks. That is the purpose of extracting HLCs to compare (Chapter 6), so that things which are written differently at the code level (other than in the simple ways prescribed by SE/S cases) may be seen to be equivalent – and SE/S cases apply just as well to HLCs as at the code level, as Figure 5.10 demonstrates.

5.4 Conclusion

This chapter has discussed semantic equivalence and semantic strength in depth, surveying related areas and techniques for assessing semantic equivalence/strength (Section 5.2), including the CCUS's own approach using SE/S cases (Section 5.3).

The basic CCUS frame-comparison algorithm (Section 3.3) walks in parallel two frame-networks representing the EXPRESS models, comparing frames slot by slot. When it encounters a difference, the CCUS uses an SE/S case to assess whether the current pair of frames are semantically equivalent (using necessary slots), or whether one frame is stronger than the other (using semantic-strength slots).

Although by no means the only system to assess semantic equivalence/strength, the CCUS is distinguished from most other systems by the way that it uses semantic equivalence/strength as a correspondence heuristic (cf. Section 4.2.5) (another example of this is in the module matching of [ZW97], as mentioned in Section 5.2.2.4). The CCUS uses assessment of semantic equivalence/strength to compare the semantics of the constraints in EXPRESS models – something which is often ignored in schema comparison and schema integration [Tür99] (Section 2.2.4).

The CCUS is also distinguished by the way that it assesses semantic equivalence/strength not just at the code level, but also at higher levels of abstraction, which is the subject of the next chapter.

Chapter 6

Combining Comparison with Extraction

This chapter discusses how the comparison of two codes (specifically EXPRESS models) is combined with the extraction of higher-level knowledge (*higher-level constraints* (HLCs)) from code, so that semantically equivalent constraints which are written differently can be assessed to be equivalent (Chapter 5) and inferred to correspond (Chapter 4).

An HLC is a small chunk of higher-level knowledge which characterises the semantics of a constraint at a higher level of abstraction, so that semantically equivalent constraints which are written differently can be seen to be equivalent (e.g. `(age >= 0) AND (age < 130)` can be seen to be equivalent to {`0 <= age < 130`} by extracting an HLC of type `range` from both of these expressions, and comparing the HLCs).

The purpose of extracting HLCs is not to cover the complete semantics of the input codes (unlike conventional cliché-based code understanding – Section 2.1.1), but to facilitate the assessment of semantic equivalence when it cannot be assessed at the code level. Extracting HLCs does not add semantics – rather, it makes implicit semantics explicit.

As explained in Section 3.1, the *HLC-extracting Comparative Code-Understanding System* (HLC CCUS) combines the code-level CCUS, which compares two EXPRESS models (Chapter 3), with the *Code-Understanding System* (CUS), which extracts HLCs from an EXPRESS model (Appendix B).

In the HLC CCUS, the extraction of HLCs is triggered when the CCUS encounters a difference that it cannot resolve at the current abstraction level.[1] The assumption that justifies extraction being triggered **only** by differences is that if two corresponding code-fragments are syntactically identical, then they are semantically equivalent (see Sections 3.3 and 4.1.1 for a discussion of this assumption).

As will be explained in Section 6.3.1, the HLC CCUS submits frames from the vicinity of each difference in (the frame-representation of) both models as extraction prompts to the CUS, which extracts from each model a small set of HLCs. The HLCs extracted from one model are then compared with the HLCs extracted from the other model (Section 6.3.2), and for each pair of HLCs found to be equivalent, correspondence is inferred between them, and between the lower-level frames whose semantics they cover (so-called *abstract correspondences* – Section 6.3.3).

Section 6.4 presents a mini-contribution: how to propagate constraints from defined types to entities.

This chapter begins with a justification of comparative extraction over code-level equational reasoning (Section 6.1), followed by a justification of the method of comparative extraction adopted here, with a discussion of possible alternatives (Section 6.2).

6.1 Introduction

The first issue to be addressed in this chapter is how to assess the semantic equivalence of two code-fragments when there are differences that cannot be resolved at the code level (using the techniques discussed in previous chapters). There are fundamentally two ways to do this:

- *Equational reasoning*: Rewrite one or both code-fragments[2] so that they are syntactically identical and thus semantically equivalent, using some predefined equivalences (e.g. to see that `(age >= 0) AND (age < 130)` and `{0 <= age < 130}` are equivalent, use a predefined equivalence such

[1] The term *current abstraction level* refers to the level at which the CCUS is currently operating: either the code level, or a level of HLCs.

[2] Code-fragments are often mathematical expressions, so it may help to read “expression” for “code-fragment”.

as $(x \geq l) \wedge (x < u) \equiv \{l \leq x < u\}$). This is the orthodox mathematical approach.

- *Extract-and-compare*: Extract from each code, and compare, higher-level knowledge that expresses the semantics of code-fragments using common abstractions (e.g. to see that `(age >= 0) AND (age < 130)` and {`0 <= age < 130`} are equivalent, extract an HLC of type `range` from each expression, and compare these HLCs to see that both expressions express a range-constraint with a lower bound of 0 (inclusive) and an upper bound of 130 (exclusive)). This is the approach chosen for the HLC CCUS.

Equational reasoning and extract-and-compare have overlapping capabilities – they are both used to assess equivalence. For the rest of this section, the following terms will be used:

- *Equivalence laws* are used in equational reasoning to see that two code-fragments are equivalent, for example $(x > l) \wedge (x < u) \equiv \{l < x < u\}$ is an equivalence law.
- *Extraction cases* are used in extract-and-compare to extract higher-level knowledge from a code-fragment, for example:
  - "An expression of the form $(x > l) \wedge (x < u)$ is a range-constraint on x with an exclusive lower bound of l and an exclusive upper bound of u."
  - "An expression of the form $\{l < x < u\}$ is a range-constraint on x with an exclusive lower bound of l and an exclusive upper bound of u."

The use of equational reasoning to infer equivalence is already well established – for example, Talcott [Tal98] emphasises the use of equational reasoning in program equivalence (cf. Section 5.2.3.4), giving several equivalence laws for functional programs.

There appears to be very little use of extract-and-compare in computer science; however, it does appear to be present in various guises, such as:

- In *case-based reasoning* (CBR) [Kol93] (Section 2.3.3), a problem is solved by remembering a similar problem that was solved in the past, and adapting its solution to solve the current problem. This entails matching the current problem to the problem part of a case and, if they match, extracting the solution to the new problem from the solution part of the case (with appropriate adaptations). However, instead of extracting knowledge to facilitate comparison (triggered by differences at the code level), CBR performs comparison to decide whether it is appropriate to extract (thus extraction is triggered by similarities rather than by differences).

- In CBR, the problem situation may be *elaborated* to conform to the indexing vocabulary of cases – a process known as *situation assessment* (Section 10.1 of [Kol93]) – which is a form of extraction to facilitate comparison (case matching).

- In schema comparison (Section 2.2.4), the schemata may be semantically enriched in order to resolve semantic heterogeneity [Hak98].

  - "*Semantic enrichment* involves the usage of other knowledge sources than the component schemas. A semantically enriched schema expresses more of the semantics of a UoD[3] than an original schema by introducing additional, generic information... Characteristic for a semantic enrichment method is that it provides additional semantics to the component schemas not covered by the used modelling language... a semantic enrichment method virtually imposes a global view to the schema structures in the local schemas." Semantic enrichment techniques include concept hierarchies, shared ontologies, linguistic knowledge, fuzzy terminologies, semantic proximity, and domain abstractions.

  - Semantic enrichment entails tasks such as transforming relational database schemata to object-oriented database schemata [HP96], but it is not clear that semantic enrichment adds any semantics that would not be present in EXPRESS models anyway (EXPRESS already supports subtype hierarchies, for example).

- In *schema integration* (merging the local schemata of a federated database system into a global schema), the local schemata may first be transformed into a common data model and filtered (Section 3.2.4 of [Tür99]). This is

[3]Universe of Discourse.

another example of extraction to support comparison (schema integration includes schema comparison as a subprocess – Section 2.2.4).

- In schema integration, the global schema may be so large that it is necessary to simplify it – "to produce, through abstraction, a semantic summary" [PSU98]. This entails "interleaving integration and abstraction steps." The global schema may also be very generic (as in data warehousing – Section 2.3.2) and have to be specialised.
- The *Resource Description Framework* (RDF) [Mil98] provides a data model for describing resources such as Web pages using structured metadata. RDF represents the semantics of resource descriptions unambiguously. For example, "The author of Document 1 is Andrew Broad" and "Andrew Broad is the author of Document 1" are syntactically different strings, but RDF represents both these statements as a resource `Document 1`', which has a property `Author` with value `''Andrew Broad''`.

Extract-and-compare can be more space-efficient than equational reasoning in that $O(n^2)$ equivalence laws are needed to simulate a set of n extraction cases. To infer that `(age >= 0) AND (age < 130)` is equivalent to `{0 <= age < 130}`, either two extraction cases or a single equivalence law are needed, but consider the three EXPRESS models in Figure 6.1. They express the constraint that a person must have one of two genders (encoded by the `choice_set` HLC (Appendix A, Section A.1), which can be extracted from each model) in three different ways. These three models can be considered semantically equivalent, in the sense that mapping from one model to one of the others cannot violate any constraints (Section 1.2).

- Three extraction cases are needed: one to extract a `choice_set` from an enumeration type, one to extract a `choice_set` from a select type, and one to extract a `choice_set` from a supertype expression.
- Alternatively, three equivalence laws are needed: to state the equivalences between enumeration and select, between enumeration and supertype, and between select and supertype. Admittedly one of these equivalence laws could be dropped, but then it would sometimes be necessary to apply two equivalence laws to show one equivalence – saving space at the expense of

```
ENTITY person;
  name : STRING;
  gender : gender_type;
END_ENTITY;

TYPE gender_type =
  ENUMERATION OF (male, female);
END_TYPE;
```

```
ENTITY person;
  name : STRING;
  gender : gender_type;
END_ENTITY;

TYPE gender_type =
  SELECT (male, female);
END_TYPE;

ENTITY male;
END_ENTITY;

ENTITY female;
END_ENTITY;
```

```
ENTITY person ABSTRACT SUPERTYPE
              OF (ONEOF (man,woman));
  name : STRING;
END_ENTITY;

ENTITY man SUBTYPE OF (person);
END_ENTITY;

ENTITY woman SUBTYPE OF (person);
END_ENTITY;
```

Figure 6.1: Three semantically equivalent EXPRESS models.

n	$\frac{1}{2}n(n-1)$
2	1
3	3
4	6
5	10
6	15
7	21

Table 6.1: The number of equivalence laws needed when there are n ways to write something.

time and elegance (why discriminate between three equally valid equivalence laws?).

Similarly, if there are four ways to write something then either four extraction cases or six equivalence laws would be needed $\{a = b,\ a = c,\ a = d,\ b = c,\ b = d,\ c = d\}$. Table 6.1 shows the number of equivalence laws needed when there are n ways to write something.

To generalise, if there are n different ways to write something, then either n extraction cases are needed, or a total of $\frac{1}{2}n(n-1)$ equivalence laws (or $n-1$ equivalence laws if storing unredundantly, every one of which would have to be applied in the extreme case – cf. Figure 6.2). Thus extraction cases combine space-efficiency (only n of them) with time-efficiency (two extractions and one comparison in every case), whereas for equivalence laws there is a trade-off between time-efficiency and space-efficiency.

Extract-and-compare requires fewer extraction cases than equational reasoning requires equivalence laws because in extract-and-compare, the comparison is 'dynamic' rather than hard-wired into equivalence laws. That is to say, it extracts higher-level knowledge and does the comparison using a generic algorithm, rather than trying to capture all primitive comparisons as equivalence laws.

The benefits of extract-and-compare over equational reasoning become more pronounced not only with the more different ways there are to write something, but also with the complexity of the code-fragments to be compared. For example, equational reasoning may be all very well as far as mathematical expressions are concerned, but when comparing programs that implement different sorting algorithms, it makes much more sense to extract this fact (e.g. as a tree of clichés where the top-level cliché is 'sort' – cf. Section 2.1.1) and just compare the top-level clichés, than to write pairs of program templates[4] (as the equivalence laws) for all envisaged pairs of implementations and rewrite one program into the other using these equivalence laws.

Extract-and-compare is more time-efficient than equational reasoning in that it is more focused. Given two code-fragments a and b, extract-and-compare simply extracts higher-level knowledge from a and from b, and compares the higher-level knowledge thus extracted. Equational reasoning on the other hand, unless it has an equivalence law that can infer directly that a and b are equivalent, entails search – the trying of possibilities for rewriting one code-fragment into another.

At worst, this search may be random and combinatorially explosive (cf. Section 5.2.1.2). Thus it helps to have heuristics (metarules, as it were) for trying to prove the equivalence of two expressions. For example, one heuristic is to reduce both expressions to a standard form such that they are semantically equivalent if and only if they are syntactically identical (Section 5.2.3.1). Here, the equivalence laws can be used as *reduction rules* that only work in one direction – towards a solution. However, since not all expressions can be reduced to a standard form (Section 2.5.2 of [Wol91]), some equivalence problems inherently entail antireduction as well as reduction. The proof in Section 5.2.1.2 that $x \vee (x \wedge y) \equiv x$ uses antireduction, although this could have been avoided by comparing truth-tables instead, or by accepting this equivalence as an axiom rather than proving it from first principles.

Extract-and-compare is like reducing to a standard form (Section 5.2.3.1) using one-way reduction rules (the extraction cases). The differences are that:

- The 'standard form' is at an abstract level rather than the code level. The semantics are expressed explicitly, whereas code-level standard forms tend

[4]With gaps to be filled – cf. FIL-C cases (Appendix B, Section B.1).

to be less human-readable than the original expressions, let alone higher-level knowledge representing the semantics of those expressions.

- Extract-and-compare applies to code-fragments in general, not just expressions – although equational reasoning could be extended to cover structural constructs as well as expressions.

Another heuristic approach to equational reasoning would be to use CBR. That is, when attempting to prove the equivalence of two expressions, retrieve a case whose problem part is a similar pair of expressions, and whose solution part is a proof that these expressions are equivalent. The old proof can be used to guide the new proof, suggesting what steps to try, thus framing the solution rather than trying proof steps at random with much backtracking. For example, the proof that $x \vee (x \wedge y) \equiv x$ in Section 5.2.1.2 can be adapted to be a proof that $x \wedge (x \vee y) \equiv x$ by replacing $\vee$ with $\wedge$, $\wedge$ with $\vee$, and $true$ with $false$ throughout the proof. The new proof only has to be verified, rather than searched for in the space of possible sequences of rewrites.

- Melis and Schairer [MS98] present an application of CBR to the reuse of proofs for formal program verification;

- Walther and Kolbe [WK00] present a method of theorem-proving which reuses previously-computed proofs – not only retrieving and adapting them, but *preparing* proofs to make them more reusable by extracting the essence of a proof and generalising it, so that a wider range of theorems to be proved will match the *proof shell* thus prepared.

Of course, when a pair of equivalent expressions is retained as a case, it can simply be used as an equivalence law (a theorem) itself, as mentioned in Section 5.2.1.2. The use of theorems as well as axioms (primitive equivalence laws) reduces the number of steps in proofs where the theorems are applicable. However, due to the costs of space and retrieval time, the system should not store all possible theorems (of which there are infinitely many), or even all the theorems it encounters – it should just store the axioms and a subset of theorems that are frequently useful.

In summary, extract-and-compare has the advantages of an abstract representation, which could handle complexity better than equational reasoning (recall the example of sorting algorithms from earlier in this section). Moreover, the abstract representation makes implicit semantics explicit, which is more understandable to humans, as well as useful for showing at an abstract level that two code-fragments are semantically equivalent. Furthermore, extract-and-compare can be more efficient in space and time, avoiding the quadratic/combinatorial explosion problems that can afflict equational reasoning.

Of course, equational reasoning and extract-and-compare could complement each other, rather than having to choose one to the exclusion of the other. It would be desirable for a comparative code-understanding system to be able to use equational reasoning as well as extract-and-compare. Equational reasoning could add flexibility in that it can perform arbitrary manipulations of expressions at the code level – it would certainly be desirable to have standard mathematical equivalence laws at the system's disposal. The arguments of this section suggest that equational reasoning may be better for expressions, and extract-and-compare for more complicated structural constructions (such as programs). This calls for experiments which are beyond the scope of the current implementation.

The current HLC CCUS uses extract-and-compare without standard equational reasoning, and thus serves as an early test-bed for comparative code-understanding which could be enhanced in the future by implementing equational reasoning. Section 6.2 discusses various possible approaches to comparative extraction as a prelude to Section 6.3, which presents the currently implemented HLC CCUS.

6.2 Methods of Comparative Extraction

There are a number of possible methods for combining the comparison of code with the extraction of higher-level knowledge (*comparative extraction*), which are discussed in this section.

The method of comparative extraction adopted in the HLC CCUS is to walk the frame-networks representing the two codes (i.e. the EXPRESS models) in parallel, extracting higher-level knowledge (i.e. HLCs) locally whenever a difference is encountered (Section 6.3), comparing the HLCs thus extracted (Section 6.3.2), and inferring *abstract correspondences* back at the current abstraction level when a pair of HLCs are found to correspond (Section 6.3.3).

Walking the two codes in parallel and trying to understand them together is a natural method for comparative code understanding, which humans tend to adopt when comparing two codes by hand. Because higher-level knowledge is only extracted when a difference is encountered, selective extraction (Chapter 7) is achieved as a by-product of this method. So the code only needs to be analysed to the depth necessary for the purpose at hand (comparing the semantics of the constraints in two EXPRESS models): the codes are compared at the code level where that suffices, and at an abstract level only where necessary. In the extreme case, no higher-level knowledge would be extracted from two identical codes.[5]

6.2.1 Extraction-Driven Comparison or Comparison-Driven Extraction

An alternative method would be to consider each code in isolation, extracting all the higher-level knowledge (HLCs) it is capable of extracting, unselectively, and then to compare just the HLCs. It could extract all the HLCs *bottom-up* from one model, and then try to match them *top-down* to the other model, i.e. hypothesise that an HLC could be extracted from the second model (by reinstantiating[6] an HLC from the first model) and try to match it to the code, possibly hypothesising other, lower-level HLCs along the way.[7]

This approach could be termed *extraction-driven comparison*, because it extracts first and then compares, whereas the approach adopted is *comparison-driven extraction*, as it compares first at the code level, with differences triggering the extraction and comparison of higher-level knowledge.

It was actually envisaged (Section G.3 of [Bro99]) that the extraction-driven comparison approach would be the chosen one, but the approach proposed there proved to be naïve because it assumed that all comparison could be done at the HLC level, with no code-level comparison. However, extracting HLCs unselectively and then comparing just the HLCs is both unnecessary and insufficient:

- It is unnecessary when the code could be compared perfectly well at the code level, as is often the case.

[5] This is not strictly true for the HLC CCUS, because the aggregation-matching policy for unordered vectors (Section 3.4.1) demands that each element in one vector be compared with each element in the other, which can, in fact, lead to differences being encountered and thus HLCs being extracted.

[6] To *reinstantiate* a frame (or a network of frames) means to copy it, slot for slot, with appropriate substitutions.

[7] These intermediate HLCs would have to be generated rather than matched, so in a sense, it would be doing code generation.

- It is insufficient because the HLCs cover the semantics of the code only partially – their primary aim is not to cover the semantics of the code (the frame-representation of the code does that completely), but to facilitate comparing the code at an abstract level, which entails making some of the implicit semantics explicit.
- It also seemed much more natural and methodical to start by implementing the code-level CCUS, and then to progress to the HLC CCUS.

6.2.1.1 Comparing Three or More Models – Persistent Extraction

A reason for extracting all HLCs unselectively from each model (other than for their own sake – HLCs can be a useful aid to human understanding in their own right) could be to facilitate comparing three or more models. The work presented in this thesis is tuned to comparing just two models, but there are often more than two models that could be compared together.

One possible approach to comparing three or more models would entail *persistent extraction*, i.e. preprocessing each model by extracting all HLCs unselectively, and saving the frame-network representing the model and its HLCs. The models could then be compared with the HLCs already attached to the code-level frames for when differences are encountered.

The issue is whether the benefit of not having to parse the models, build the internal frame-representation and extract any HLCs when running a comparison, would outweigh the filestore costs and I/O time-overhead of saving and loading the frame-networks.[8]

This approach is more likely to be advantageous the more pairs of models are to be compared for the number of models under consideration, and the more different the models are from each other (i.e. the more HLCs have to be extracted to understand them comparatively) (Section 1.4.2):

- models which are different revisions of each other would tend to be compared in a chain (i.e. $n-1$ comparisons for n models) and to have few and minor differences;
- different models of the same thing can be radically different, and up to

[8] The frame-networks can be large, and could lead to some extremely large binary files as saved by Java's serialisation mechanism.

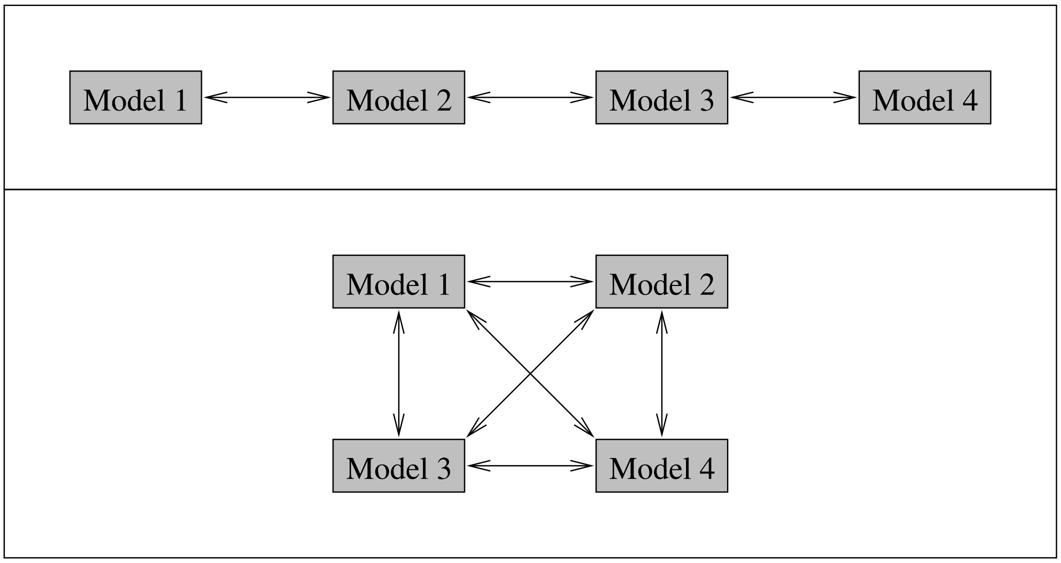


Figure 6.2: Comparing n models: $n-1$ or $\frac{1}{2}n(n-1)$ comparisons.

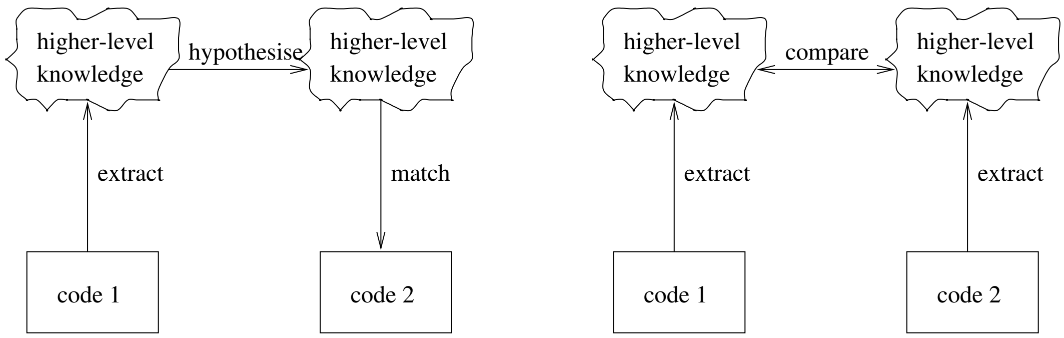
$\frac{1}{2}n(n-1)$ different comparisons may be called for (see Figure 6.2) – persistent extraction would be much more beneficial in this case.

6.2.2 Bottom-up/Top-down or Bottom-up/Bottom-up

Just as non-comparative code understanding can be either bottom-up or top-down (Section 2.1.1), there are similar notions for comparative code understanding, as illustrated in Figure 6.3.

- *Bottom-up* from one code, *top-down* from the other (BUTD). The comparative code-understander extracts higher-level knowledge from one code, then hypothesises that it holds on the other code and attempts to match it top-down to the second code. The extraction-driven comparison approach discussed in Section 6.2.1 is an example of BUTD.

- *Bottom-up* from both codes (BUBU). The comparative code-understander extracts higher-level knowledge from both codes and compares it. The comparison-driven extraction approach adopted in the HLC CCUS is an example of BUBU, but it extracts HLCs in localised bouts (triggered by differences), rather than extracting all HLCs unselectively from one whole model and then the other.



(a) Bottom-up/Top-down

(b) Bottom-up/Bottom-up

Figure 6.3: Two basic models of comparative code understanding.

- Hybrid approaches between BUTD and BUBU would also be possible, using artificial intelligence techniques such as meta-level reasoning (cf. Section 7.1) to decide whether to work BUTD or BUBU in the specific circumstances.

- The obvious top-down-first approaches (TDTD and TDBU) make little sense, because there is nothing on which to base the initial hypothesis for top-down extraction (unless the comparative code-understander was told what to look for).

BUTD comparative extraction relates to *heuristic classification* [Cla85], a problem-solving method used in knowledge-based systems for classification tasks such as diagnosis. In the case of medical diagnosis, for example, heuristic classification works by *abstracting* patient data to a class of symptoms, performing a *heuristic match* to associate the class of symptoms with a class of diseases, and then *refining* from the class of diseases to a specific disease. The inference structure of heuristic classification is isomorphic to Figure 6.3(a); the key differences are that:

- the corresponding pieces of higher-level knowledge are of the same kind in comparative extraction, whereas heuristic classification relates abstractions of input data to abstractions of solutions;

- the diagnosis of a specific disease is generated, as opposed to matching top-down to the second code, which is given as input.

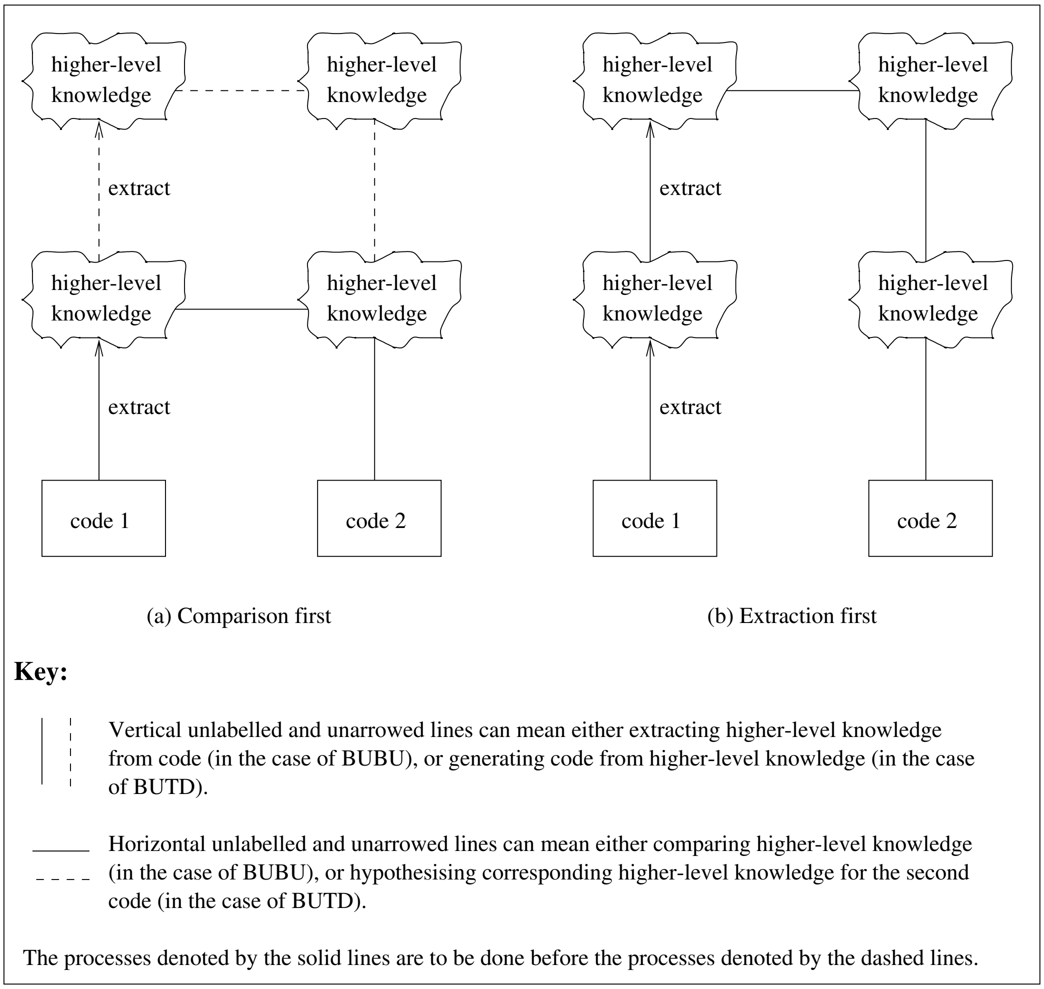


Figure 6.4: Comparison-first and extraction-first comparative code understanding.

6.2.3 Comparison First or Extraction First

Another issue, which is orthogonal to whether comparative extraction is BUTD or BUBU, is whether to give priority to extraction or to comparison, i.e. whether to compare the HLCs extracted (or attempt the top-down match) immediately (*comparison first*), or to prefer extracting to an even higher level over matching what has already been extracted (*extraction first*). Figure 6.4 illustrates these two approaches.

- The comparison-first approach would be better for highly similar codes (e.g. different revisions of the same code), or codes which can be seen to be equivalent at a shallow level of abstraction (e.g. the same algorithm written in different languages), as failure to match could be expensive if it has to go up through many levels of abstraction before they match.

- The extraction-first approach would be better for radically different codes which have only abstract similarities (e.g. two different sorting algorithms, or two different models of the same thing), if intermediate levels of abstraction are to be discarded for comparison (which saves effort but misses correspondences).

- Again, hybrid approaches should be possible, to decide dynamically whether to compare at the current level or move up a level of abstraction (quick heuristics should be more efficient than detailed reasoning).

This distinction makes little difference for the comparative code understanding of EXPRESS models anyway, as the semantics of constraints are highly localised. Whereas program understanding aims to build one big, connected tree of clichés for a program, code understanding for information models tends to produce a forest of numerous, unconnected little trees of HLCs, each only one or two levels deep (Section 2.1.1). The HLC CCUS works extraction-first once it starts one of its localised bouts of extraction (Section 6.2.2), and does not discard intermediate HLCs for comparison.

To summarise Section 6.2, the HLC CCUS:

- uses *comparison-driven extraction*: the extraction of HLCs is triggered by the differences found by the comparison process (Section 6.2.1);

- extracts HLCs bottom-up from both models (in localised bouts of extraction from the vicinity of each difference) (Section 6.2.2);

- works extraction-first for each bout of extraction (Section 6.2.3).

The details of how the HLC CCUS works are presented in the next section.

6.3 The HLC CCUS

This section explains how the HLC CCUS interacts with the CUS to extract HLCs from the vicinity of a difference in both models (Section 6.3.1). Informally, the *vicinity* of a difference is the code-fragment, in each model, that is

`ENTITY person;`
`  name : STRING;`
`  age : INTEGER;`
`WHERE`
`  (age >= 0) AND (age < 130);`
`END_ENTITY;`
	`ENTITY person;`
	`  name : STRING;`
	`  age : INTEGER;`
	`WHERE`
	`  {0 <= age < 130};`
	`END_ENTITY;`

Figure 6.5: A pair of simple EXPRESS models for the HLC CCUS to understand comparatively.

'far enough out' to extract the HLCs needed to resolve the difference, and yet 'far enough in' to minimise the extraction of extraneous HLCs. Section 6.3.2 discusses comparing the HLCs thus extracted, and Section 6.3.3 discusses how abstract correspondences are inferred back at the code level when the HLCs thus compared correspond.

Consider, for example, the pair of models in Figure 6.5. The domain rules `(age >= 0) AND (age < 130)` and {`0 <= age < 130`} are syntactically different but semantically equivalent.[9] To assess their semantic equivalence, the HLC CCUS extracts an HLC of type `range` from each of the domain rules:

- a discrete range on `age` in the first model, with an inclusive lower bound of 0 and an exclusive upper bound of 130;
- a discrete range on `age` in the second model, with an inclusive lower bound of 0 and an exclusive upper bound of 130.

It then compares these two HLCs (Section 6.3.2) and, upon inferring that they correspond, infers an *abstract correspondence* between the two domain rules (Section 6.3.3) – so called because they correspond at an abstract level, but their correspondence cannot be inferred at the code level.

Pseudocode algorithms for the HLC CCUS are given in Appendix C, Section C.5.

[9]Semantic-equivalence/strength cases (Section 5.3) do not suffice to assess this semantic equivalence – they only allow simple differences to be resolved by specifying, for a given frame-class of frames to be compared, unnecessary slots (by omission) and semantic-strength slots.

```
INSTANCE domain_rule#1;
  expression = conjunction#1;
END_INSTANCE;

INSTANCE conjunction#1;
  conjuncts = {comparator#1, comparator#2};
  covered_by = {range#1};
END_INSTANCE;

INSTANCE comparator#1;
  greater = explicit_attribute#2; (* age *)
  lesser = integer_literal#1; (* 0 *)
  inclusive = inclusive_or_exclusive(inclusive);
  discrete = discrete_or_continuous(discrete);
  hlcs_extracted = {bound#1};
  covered_by = {bound#1};
END_INSTANCE;

INSTANCE comparator#2;
  greater = integer_literal#1; (* 130 *)
  lesser = explicit_attribute#2; (* age *)
  inclusive = inclusive_or_exclusive(exclusive);
  discrete = discrete_or_continuous(discrete);
  hlcs_extracted = {bound#2};
  covered_by = {bound#2};
END_INSTANCE;
```

```
INSTANCE domain_rule#2;
  expression = interval#1;
END_INSTANCE;

INSTANCE interval#1;
  lower_bound = integer_literal#3; (* 0 *)
  lower_inclusive = inclusive_or_exclusive(inclusive);
  variable = explicit_attribute#4; (* age *)
  discrete = discrete_or_continuous(discrete);
  upper_inclusive = inclusive_or_exclusive(exclusive);
  upper_bound = integer_literal#4; (* 130 *)
  hlcs_extracted = {range#2};
  covered_by = {range#2}
END_INSTANCE;
```

```
INSTANCE bound#1;
  covers = comparator#1;
  extracted_from = {comparator#1};
  kind = lower;
  inclusive = inclusive_or_exclusive(inclusive);
  discrete = discrete_or_continuous(discrete);
  variable = explicit_attribute#2; (* age *)
  limit = integer_literal#1; (* 0 *)
  hlcs_extracted = {range#1};
END_INSTANCE;

INSTANCE bound#2;
  covers = comparator#2;
  extracted_from = {comparator#2};
  kind = upper;
  inclusive = inclusive_or_exclusive(exclusive);
  discrete = discrete_or_continuous(discrete);
  variable = explicit_attribute#2;
  limit = integer_literal#2; (* 130 *)
  hlcs_extracted = {range#1};
END_INSTANCE;

INSTANCE range#1;
  covers = conjunction#1;
  extracted_from = {bound#1, bound#2};
  lower_bound = bound#1;
  upper_bound = bound#2;
  variable = explicit_attribute#2; (* age *)
  discrete = discrete_or_continuous(discrete);
END_INSTANCE;
```

```
INSTANCE range#2;
  covers = interval#1;
  extracted_from = {interval#1};
  lower_bound = bound#3;
  upper_bound = bound#4;
  variable = explicit_attribute#4; (* age *)
  discrete = discrete_or_continuous(discrete);
END_INSTANCE;

INSTANCE bound#3;
  kind = lower;
  inclusive = inclusive_or_exclusive(inclusive);
  discrete = discrete_or_continuous(discrete);
  variable = explicit_attribute#4; (* age *)
  limit = integer_literal#3; (* 0 *)
END_INSTANCE;

INSTANCE bound#4;
  kind = upper;
  inclusive = inclusive_or_exclusive(exclusive);
  discrete = discrete_or_continuous(discrete);
  variable = explicit_attribute#4;
  limit = integer_literal#4; (* 130 *)
END_INSTANCE;
```

Figure 6.6: Part of the frame-representation of the models in Figure 6.5.

6.3.1 Extracting HLCs

In order to appreciate precisely what it means for the HLC CCUS to extract HLCs from the vicinity of a difference in both models, it is necessary to consider the above example at the frame level. Figure 6.6 shows part of the frame-representation of the two domain rules in Figure 6.5 and of the HLCs extracted therefrom.[10] Note the special slots `covers`, `covered_by`, `extracted_from` and `hlcs_extracted`[11] – their roles will be explained in Section 6.3.3.

[10] Some slots and frames have been omitted for brevity.

[11] These slots are ignored by the frame-comparison algorithm, since they are marked with `--IGNORE` in the frame-class model (Appendix D).

When the CCUS frame-comparison algorithm (Section 3.3) comes to compare `domain_rule#1` and `domain_rule#2`, it attempts to resolve the difference in the `expression` slot. Since `conjunction#1` and `interval#1` are not even of the same frame-class, their correspondence is vetoed immediately (Section 4.2.2).[12] So the HLC CCUS tries to resolve the difference by extracting and comparing HLCs.

On behalf of the HLC CCUS, the CUS performs a *bout of extraction* from the vicinity of the difference in both models. To achieve this, the HLC CCUS submits a frame from each of the models as an *extraction prompt* to the CUS, and the CUS extracts whatever HLCs it can from each frame given as an extraction prompt (in fact it works inwards, extracting HLCs from the subnetwork of frames whose top-level frame is the extraction prompt). The HLC CCUS then compares each of the HLCs extracted from the first model with each of the HLCs extracted from the second model, as will be described in Section 6.3.2.

The CUS's method of extracting HLCs is not a contribution of *this* thesis – it was first presented in [Bro99], and an overview is presented in Appendix B. In a nutshell, the CUS extracts HLCs from a model by retrieving *extraction cases*, to whose *problem parts* code-fragments in the model are matched. If a code-fragment successfully matches the problem part of an extraction case, the HLCs in the case's *solution part* are extracted by reinstantiating them (essentially substituting constants for variables) for that code-fragment.

Given a slot difference, the HLC CCUS has to decide which frames to submit as extraction prompts to the CUS – this is what is meant by the *vicinity* of the difference.

Since the CUS uses *navigation cases* to guide it inwards in its submission of extraction prompts (e.g. given a model, navigate its schemata; given a schema, navigate its entities; given an entity, navigate its attributes, local rules, supertype expression and subtypes), the given extraction prompt needs to be in the right place for optimum extraction of HLCs:

- If the frame used as an extraction prompt is 'too far in', the CUS might miss HLCs that it needs to extract in order for the HLC CCUS to infer the abstract correspondences that it should.

- If the extraction prompt is 'too far out', the CUS might extract *extraneous*

[12]If they did pass the frame-equivalence test, the CCUS would attempt to resolve the difference using a semantic-equivalence/strength case (Section 5.3) before attempting to resolve it by extracting and comparing HLCs.

HLCs that are not needed for the HLC CCUS to see abstract correspondences (extraneous HLCs will be discussed further in Chapter 7 – in short, they constitute a fairly major efficiency problem). The extreme case is to use a whole model as an extraction prompt, thus extracting all HLCs that can be extracted from that model (as the stand-alone CUS does).

To decide which frames to submit as extraction prompts, the HLC CCUS considers the frames f_1 and f_2 currently being compared (i.e. `domain_rule#1` and `domain_rule#2`), which contain the slot s for which there is a difference (`expression` in the current example), and, of course, the fillers $f_1.s$ and $f_2.s$ of that slot in both models (`conjunction#1` and `interval#1`). For each of the two models (for the ith model):

- If the slot-filler $f_i.s$ is null, no extraction prompts are submitted.
- If the slot-filler $f_i.s$ is a frame, then the slot-filler $f_i.s$ is submitted as an extraction prompt (thus in the current example, `conjunction#1` and `interval#1` are submitted as extraction prompts).
- If the slot-filler $f_i.s$ is a vector (aggregation) of frames, then each frame in the vector is submitted as an extraction prompt.[13]
- If the slot-filler $f_i.s$ is neither a frame nor a vector, then the surrounding frame f_i is submitted as an extraction prompt, f_i thus being the innermost frame in the vicinity of the difference.

The above assumes that extraction prompts are submitted *separately*, but alternatively they can be submitted *together* (at the choice of the user for a particular run of the CCUS). Submitting extraction prompts together causes more frames to be submitted as extraction prompts, so it can cause more HLCs to be extracted and more correspondences to be inferred (hence it is more complete than submitting extraction prompts separately), but it also causes a greater number of extraneous HLCs to be extracted (so it is less efficient).

When submitting extraction prompts together (f_1 and f_2 are the frames being compared):

- If both $f_1.s$ and $f_2.s$ are null, no extraction prompts are submitted.

[13]The current HLC CCUS ignores any non-frame vector elements – it just gives a warning.

- If both $f_1.s$ and $f_2.s$ are frames, then $f_1.s$ and $f_2.s$ are submitted as extraction prompts.

- If both $f_1.s$ and $f_2.s$ are vectors of frames, then each frame in the vectors is submitted as an extraction prompt.

- Otherwise both surrounding frames, f_1 and f_2, are submitted as extraction prompts. This is where submitting extraction prompts separately would either submit an extraction prompt which is more inner than f_1, or submit an extraction prompt which is more inner than f_2.

Examples of where it makes a difference:

- If the type of an attribute is `INTEGER` (represented by the enumeration symbol `integer_type`) in one model, and a defined type (represented as a `defined_type` frame) in the other model, then both attributes would be submitted as extraction prompts when submitting extraction prompts together (the CUS finds the defined type as it works inwards from the second attribute), but when submitting extraction prompts separately, the attribute from the first model and the defined type from the second model (a more inner extraction prompt than the attribute from the second model) would be submitted as extraction prompts.

- If an entity in one model had a domain rule (represented as a vector containing a `domain_rule` frame), and the corresponding entity in the other model had no domain rules (represented as null), then both entities would be submitted as extraction prompts when submitting extraction prompts together, but when submitting extraction prompts separately, the domain rule from the first model and nothing from the second model would be submitted.

From now on, it is to be assumed that extraction prompts are submitted separately unless otherwise stated.

For the running example (Figure 6.6), the submission of extraction prompts to the CUS causes the following HLCs to be extracted in the following two bouts of extraction (one for each model):

- Given `conjunction#1` from the first model as an extraction prompt, the CUS internally submits `comparator#1` and `comparator#2` as further (more inner) extraction prompts (the navigation case for `conjunction` (Appendix E, Section E.2.6) tells it to submit each of the conjuncts as an internal extraction prompt), thus extracting `bound#1` and `bound#2` respectively (using the extraction cases for extracting bounds from comparators – Sections E.1.5 and E.1.6). It then uses `conjunction#1` itself as a prompt to retrieve the extraction case for extracting a range from a conjunction of comparators, from which conjunction a lower bound and an upper bound have been extracted (Section E.1.7), and thus extracts `range#1`.[14]

- Given `interval#1` from the second model as an extraction prompt, the CUS uses `interval#1` as a prompt to retrieve the extraction case for extracting a range and two bounds from an interval (Section E.1.8), and thus extracts `range#2`, `bound#3` and `bound#4`.[15]

Thus the following HLCs are returned to the HLC CCUS: {`bound#1`, `bound#2`, `range#1`} from the first model, and {`bound#3`, `bound#4`, `range#2`} from the second model. The HLC CCUS compares the HLCs extracted from the first model with the HLCs extracted from the second model (Section 6.3.2), and when it infers correspondences between HLCs, it can infer *abstract correspondences* back at the code level (Section 6.3.3).

6.3.2 Comparing the HLCs Extracted

Having extracted HLCs from the vicinity of a difference in two models (as described in the previous subsection), the HLC CCUS now has to compare the HLCs extracted from one model with the HLCs extracted from the other model. The actual comparison of HLCs is performed simply by calling the standard CCUS frame-comparison algorithm (Section 3.3) with pairs of HLCs to compare; the issue here is how to control which pairs of HLCs are to be compared.

In the running example from Section 6.3.1 (Figure 6.6), the CUS returns the HLCs {`bound#1`, `bound#2`, `range#1`} from the first model, and {`bound#3`, `bound#4`, `range#2`} from the second model.

The set of HLCs that the CUS returns, for each bout of extraction, is actually

[14] The CUS extraction algorithm submits a frame as a navigation prompt before it submits it as an extraction prompt – the correct order to extract HLCs for examples such as this to work.

[15] There are no navigation cases for `interval`.

represented as a hash-table, which partitions the HLCs according to their frame-classes. The keys of the hash-table are frame-classes, and the values of the hash-table are sets of HLCs such that two HLCs are in the same set if and only if they are of the same frame-class. Thus the first set of HLCs above is represented as the hash-table {`bound` → {`bound#1`, `bound#2`}, `range` → {`range#1`}}.

The sets {`bound#1`, `bound#2`} and {`bound#3`, `bound#4`} are compared using the unordered-aggregation-matching policy (Section 3.4.1). Thus:

- `bound#1` is compared with `bound#3`, and they are inferred to correspond (by semantic equivalence – Section 4.2.5);
- `bound#1` is compared with `bound#4`, but they do not correspond;
- `bound#2` is compared with `bound#3`, but they do not correspond;
- `bound#2` is compared with `bound#4`, and they are inferred to correspond.

Similarly, {`range#1`} and {`range#2`} are compared, and `range#1` and `range#2` are found to correspond.

The current example is simple because the corresponding HLCs happen to come from two corresponding bouts of extraction. This is not always the case in general, however, because differences can be *delocalised* such that corresponding HLCs do not come from corresponding bouts of extraction. Therefore, it is necessary to *remember* HLCs from previous bouts of extraction, so that they can be compared with HLCs from later bouts of extraction. One approach is to remember all HLCs unselectively, but since comparing a set of n HLCs with a set of m HLCs is an $O(nm)$ operation, HLCs should be remembered selectively – this is achieved through the use of *remembrance heuristics*, which tell the HLC CCUS which HLCs to remember and which to discard. The topic of remembering HLCs belongs to Chapter 7 as it is a matter of *selective comparison*, and will be described in detail in Section 7.2.

In summary, HLCs are compared by comparing sets of HLCs of the same frame-class (since sets of HLCs are represented as hash-tables, partitioned by frame-class). HLCs may be remembered from previous bouts of extraction, which entails merging hash-tables (i.e. unioning[16] together sets of HLCs of the same frame-class from different bouts of extraction). When HLCs are inferred to correspond, any lower-level frames whose semantics they cover are also inferred to

[16]Sets of HLCs from different bouts of extraction are, of course, disjoint.

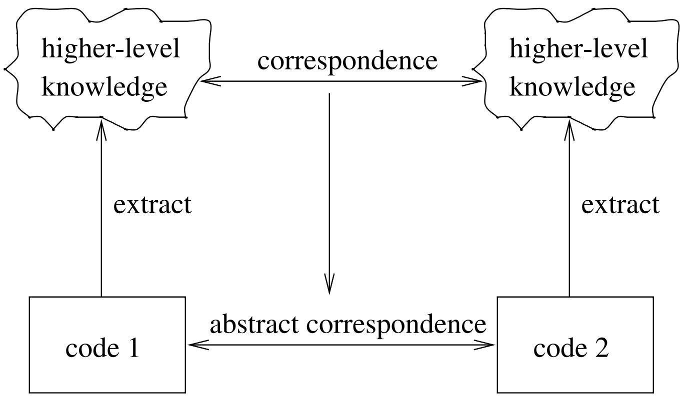


Figure 6.7: Abstract correspondence.

correspond – these *abstract correspondences* are the subject of the next subsection.

6.3.3 Abstract Correspondences

An *abstract correspondence* (Figure 6.7) is a correspondence at the code level – or at any level of abstraction – which cannot be inferred at that level, but can be inferred at a higher level of abstraction, through the extraction and comparison of corresponding HLCs.

Note the following distinguished slots of HLC frames:

- `extracted_from` and its inverse slot, `hlcs_extracted`: this is a structural relationship between an HLC and the frames from which it was extracted, and is for the benefit of humans browsing the frames;

- `covers` and its inverse slot, `covered_by`: this is a semantic relationship between an HLC and the frame whose semantics it covers, which the HLC CCUS uses to infer abstract correspondences when HLCs correspond.

As noted in Section 6.3.2, the HLC CCUS compares pairs of HLCs extracted from both models. When two HLCs are inferred to correspond, their `covers` slot tells the HLC CCUS what abstract correspondence (if any) to infer. It infers a correspondence between the fillers of the `covers` slot for the two HLCs that have just been inferred to correspond (provided that both slot-fillers of `covers` are non-null).

So in the running example (Figure 6.6), when `range#1` and `range#2` are inferred to correspond, the HLC CCUS infers an abstract correspondence between their fillers of the `covers` slot: `conjunction#1` and `interval#1`. Thus an abstract correspondence is inferred between `(age >= 0) AND (age < 130)` and {`0 <= age < 130`}:

```
INSTANCE correspondence#1;
  frame1 = range#1;
  frame2 = range#2;
  semantic_equivalence = semantic_equivalence(equivalent);
END_INSTANCE;

INSTANCE correspondence#2;
  frame1 = conjunction#1;
  frame2 = interval#1;
  semantic_equivalence = semantic_equivalence(equivalent);
END_INSTANCE;
```

The slot `semantic_equivalence` in `correspondence#2` (the abstract correspondence) inherits its filler from `correspondence#1` (the HLC-correspondence).

As a more complicated example, consider comparing the first and third models of Figure 6.1 (where a person's gender is represented by an enumeration type and by a subtype hierarchy, respectively). A partial frame-representation of these models is shown in Figure 6.8. The constraint, in both models, that a person must have exactly one of two genders is represented by instances of the `choice_set` HLC extracted from `defined_type#1` and from `oneof_supertype_expression#1`.[17]

Because the attribute `gender` in the 'enumeration' model does not correspond to any attributes in the 'supertype' model, this constitutes a difference in the slot `explicit_attributes` in `entity_type#1` and `entity_type#2`. So, by the rules given in Section 6.3.1 in the case that the slot-fillers are vectors, the HLC CCUS submits `explicit_attribute#1`, `explicit_attribute#2` and `explicit_attribute#3` to the CUS as extraction prompts. Thus `choice_set#1` and `choice_set#2` are extracted.

In the course of comparing `choice_set#1` and `choice_set#2` (as in Section 6.3.2), the HLC CCUS compares `enumeration_symbol#1` with `entity_type#3`, and compares `enumeration_symbol#2` with `entity_type#4`. Hence two correspondences are inferred – sanctioned by the frame-class-thesaurus and by a name-thesaurus (Section 4.2.4) which contain the following synsets respectively:

[17]Note that the `choice_set` HLCs do not fill the `covers` slot, because it is not the case that they cover the whole semantics of, and nothing but the semantics of, the frames from which they are extracted.

```
ENTITY person;
  name : STRING;
  gender : gender_type;
END_ENTITY;

TYPE gender_type =
  ENUMERATION OF (male, female);
END_TYPE;
```

```
ENTITY person ABSTRACT SUPERTYPE
              OF (ONEOF (man, woman));
  name : STRING;
END_ENTITY;

ENTITY man SUBTYPE OF (person);
END_ENTITY;

ENTITY woman SUBTYPE OF (person);
END_ENTITY;
```

```
INSTANCE entity_type#1;
  name = "person";
  is_abstract = false;
  explicit_attributes = {explicit_attribute#1,
                         explicit_attribute#2};
END_INSTANCE;

INSTANCE explicit_attribute#2;
  name = "gender";
  is_optional = false;
  tipe = defined_type#1;
END_INSTANCE;

INSTANCE defined_type#1;
  name = "gender_type";
  tipe = enumeration_type([enumeration_symbol#1,
                           enumeration_symbol#2]);
  hlcs_extracted = {choice_set#1};
END_INSTANCE;

INSTANCE enumeration_symbol#1;
  name = "male";
  the_type = defined_type#1;
END_INSTANCE;

INSTANCE enumeration_symbol#2;
  name = "female";
  the_type = defined_type#1;
END_INSTANCE;
```

```
INSTANCE entity_type#2;
  name = "person";
  is_abstract = true;
  supertype_of = oneof_supertype_expression#1;
  explicit_attributes = {explicit_attribute#3};
END_INSTANCE;

INSTANCE oneof_supertype_expression#1;
  expressions = {entity_type#3, entity_type#4};
  hlcs_extracted = {choice_set#2};
END_INSTANCE;

INSTANCE entity_type#3;
  name = "man";
  is_abstract = false;
  subtype_of = {entity_type#2};
  explicit_attributes = {};
END_INSTANCE;

INSTANCE entity_type#4;
  name = "woman";
  is_abstract = false;
  subtype_of = {entity_type#2};
  explicit_attributes = {};
END_INSTANCE;
```

```
INSTANCE choice_set#1;
  extracted_from = {defined_type#1};
  choice_set = {enumeration_symbol#1,
                enumeration_symbol#2};
  is_optional = false;
  constrained_entity = entity_type#1;
  attribute = explicit_attribute#2;
END_INSTANCE;
```

```
INSTANCE choice_set#2;
  extracted_from = {oneof_supertype_expression#1};
  choice_set = {entity_type#3, entity_type#4};
  is_optional = false;
  constrained_entity = entity_type#2;
END_INSTANCE;
```

Figure 6.8: Part of the frame-representation of the first and third models in Figure 6.1.

```
{entitytype, enumerationsymbol,}

{male, man,}
{female, woman,}
```

The correspondences between `enumeration_symbol#1` and `entity_type#3`, and between `enumeration_symbol#2` and `entity_type#4`, are abstract correspondences, since they are inferred through the comparison of HLCs (even though they are not inferred through the `covers` slot). Because these correspondences are sanctioned, `choice_set#1` and `choice_set#2` match along their `choice_set` slots: the comparison succeeds and they are inferred to correspond. Thus the HLC CCUS has, to some extent, inferred that `man` corresponds to `person` with a value of `male` for `gender`, and that `woman` corresponds to `person` with a value of `female` for `gender`.

The problem of representing how HLCs cover the semantics of code (or of lower-level HLCs) is a difficult one in general, so not all HLCs fill the `covers` slot. For an HLC to cover the semantics of a code-fragment means that the covered code could be disregarded (by a system that transfers constraints for schema-to-schema mapping (Section 1.3), for example), but an HLC does not necessarily cover the total semantics of the code-fragment from which it is extracted – it may just express some of the semantics at a higher level of abstraction. For example, a `non_null` HLC expresses the semantics that a mandatory attribute must have a value, not the total semantics of the attribute (e.g. that the value must be of a specified type).

The current implementation permits a many-to-one relationship between an HLC and the frames it covers:

- `covered_by` is a set of HLCs that cover a frame, so a mandatory attribute could, in principle, be covered by a `non_null` HLC and a `type_hlc`. However, there is no distinction between whether each HLC covers the semantics of the attribute partially (collectively, these HLCs cover the semantics of the attribute totally), or whether the HLCs each cover the semantics totally and are therefore mutually redundant.

- `covers` is filled by at most one frame, so an HLC cannot cover the semantics of more than one top-level frame (it does implicitly cover the semantics of the subnetwork of frames under the frame it explicitly covers), as a `choice_set` extracted from a subtype hierarchy should, for example. The

<pre>ENTITY person;
  age : INTEGER;
WHERE
  {0 <= age < 130};
END_ENTITY;</pre>
	<pre>ENTITY person;
	  age : age_type;
	END_ENTITY;
	
	TYPE age_type = INTEGER;
	WHERE
	  {0 <= SELF < 130};
	END_TYPE;</pre>

Figure 6.9: A pair of EXPRESS models to illustrate propagating domain rules from defined types to entities.

> difficulty in allowing `covers` to be a set of frames is the need to specify which aspect of the semantics of the HLC covers each frame, and hence to know between which frames the abstract correspondences should be inferred (that `covers` is filled by at most one frame means that the HLC CCUS simply infers an abstract correspondence between the two frames specified by the `covers` slot-fillers of two corresponding HLCs).

The `covers` slot of an HLC is filled by the extraction case used to extract it, so see Appendix E, Section E.1 to see which HLCs do fill the `covers` slot. The `covers` slot is filled conservatively, making the inference of abstract correspondences *sound* (never inferring an abstract correspondence when it should not) but *incomplete* (not always inferring an abstract correspondence when it should).

Section 6.3 has explained how the HLC CCUS extracts HLCs from the vicinity of each difference in both models (Section 6.3.1), comparing the HLCs thus extracted (Section 6.3.2), and inferring abstract correspondences back at the code level when two HLCs correspond (Section 6.3.3).

The next section presents a mini-contribution which uses the HLC CCUS: propagating constraints from domain rules to entities via attributes.

6.4 Propagation of Constraints

Consider the pair of models in Figure 6.9. To see that they are semantically equivalent entails propagating the domain rule {`0 <= SELF < 130`} from the defined type `age_type` to the entity `person` in the second model.[18]

[18] Please note that in propagation *of* constraints, it is the constraints themselves that are being propagated, rather than propagating values through constraint networks as in propagation *via* constraints (Section 5.2.3.2).

To see that the said domain rule is semantically equivalent to {`0 <= age < 130`} entails substituting the explicit attribute `age` for the constant `SELF`, since it is through `age` in the second model that {`0 <= SELF < 130`} is propagated. There are two ways that this substitution could be performed:

- *Extraction-time substitution*: Extract a domain-rule HLC from the domain rule {`0 <= SELF < 130`}, substituting `age` for `SELF`. The difficulties with this approach are that either the case-representation language FIL-C (Appendix B, Section B.1) would have to be extended with a construct for substituting for `SELF`, or substitution would have to be hard-wired. Furthermore, it would have to perform a deep copy of the domain rule (i.e. including all subexpressions, because they could contain `SELF`), which would require an extension to FIL-C for specifying that a frame is to be deeply copied rather than substituting a reference to that frame. Moreover, any implementation of deep copying in the frame system would face the problem of knowing where the boundaries of the subnetwork to copy are (a domain rule can contain, for example, references to attributes, which should be plugged in as frame-references rather than deeply copied).

- *Comparison-time substitution*: The CCUS keeps track of the value (if any) to substitute for `SELF`, which is propagated through the CCUS frame-comparison algorithm (Section 3.3, and Appendix C, Section C.1). Whenever the CCUS encounters `SELF`, it substitutes the value it is holding for `SELF` before it compares it with a value in the other model. This is the approach that has been implemented.

Figure 6.10 shows a partial frame-representation of the models in Figure 6.9.[19] The HLC `domain_rules_hlc#1` is extracted from `entity_type#1`, and `domain_rules_hlc#2` is extracted from `explicit_attribute#2` – the latter by using an extraction case for extracting `domain_rules_hlc` from an attribute a whose type is a defined type with domain rules (Appendix E, Section E.1.26); this extraction case fills the distinguished slot `substitute_for_self` with a.

The HLCs `domain_rules_hlc#1` and `domain_rules_hlc#2` are compared and inferred to correspond. In the course of comparing these two frames, the CCUS

[19] To focus on what is important for this example, some slots have been omitted, along with some extraneous HLCs (a `non_null` from `explicit_attribute#1`, a `range` and two `bounds` from `interval#1`, and a `range` and two `bounds` from `interval#2`).

```
INSTANCE entity_type#1;
  name = "person";
  explicit_attributes = {explicit_attribute#1};
  domain_rules = {domain_rule#1};
  hlcs_extracted = {domain_rules_hlc#1};
END_INSTANCE;

INSTANCE explicit_attribute#1;
  name = "age";
  tipe = simple_type(integer_type);
  hlcs_extracted = {type_hlc#1};
END_INSTANCE;

INSTANCE domain_rule#1;
  expression = interval#1;
END_INSTANCE;

INSTANCE interval#1;
  lower_bound = integer_literal#1; (* 0 *)
  variable = explicit_attribute#1; (* age *)
  upper_bound = integer_literal#2; (* 130 *)
END_INSTANCE;
------------------------------------------------
INSTANCE domain_rules_hlc#1;
  extracted_from = {entity_type#1};
  the_entity = entity_type#1; (* person *)
  domain_rules = {domain_rule#1}; (* {0 <= age < 130) *)
END_INSTANCE;

INSTANCE type_hlc#1;
  extracted_from = {explicit_attribute#1};
  attribute = explicit_attribute#1;
  tipe = simple_type(integer_type);
END_INSTANCE;
```

```
INSTANCE entity_type#2;
  name = "person";
  explicit_attributes = {explicit_attribute#2};
END_INSTANCE;

INSTANCE explicit_attribute#2;
  name = "age";
  tipe = defined_type#1;
  hlcs_extracted = {domain_rules_hlc#2, type_hlc#2};
END_INSTANCE;

INSTANCE defined_type#1;
  name = "age_type";
  tipe = simple_type(integer);
  domain_rules = domain_rule#2;
END_INSTANCE;

INSTANCE domain_rule#2;
  expression = interval#2;
END_INSTANCE;

INSTANCE interval#2;
  lower_bound = integer_literal#3; (* 0 *)
  variable = built_in_constant#1; (* SELF *)
  upper_bound = integer_literal#4; (* 130 *)
END_INSTANCE;
------------------------------------------------
INSTANCE domain_rules_hlc#2;
  extracted_from = {explicit_attribute#2};
  the_entity = entity_type#2; (* person *)
  domain_rules = {domain_rule#2}; (* {0 <= SELF < 130} *)
  substitute_for_self = explicit_attribute#2; (* age *)
END_INSTANCE;

INSTANCE type_hlc#2;
  extracted_from = {explicit_attribute#2};
  attribute = explicit_attribute#2;
  tipe = simple_type(integer_type);
END_INSTANCE;
```

Figure 6.10: Part of the frame-representation of the models in Figure 6.9.

compares `domain_rule#1` ({`0 <= age < 130`}) and `domain_rule#2` ({`0 <= SELF < 130`}) and infers an abstract correspondence between them. In the course of comparing the `domain_rule` frames, the CCUS compares `interval#1` and `interval#2` and infers an abstract correspondence between them – their `variable` slots match because it substitutes `explicit_attribute#2` for `SELF`, having recorded this substitution from `substitute_for_self` in `domain_rules_hlc#2`.

Similarly, but more simply, the constraint that `age : age_type` must be an integer can be seen by propagating the underlying type `INTEGER` from the defined type `age_type` to the attribute `age`. This is implemented by extracting `type_hlc#2` from `explicit_attribute#2`, so that it can be compared and inferred to correspond to `type_hlc#1` (extracted from `explicit_attribute#1`).

Propagation of constraints is not, in principle, limited to propagating from defined types to entities via attributes. For example, constraints could be propagated from entities to entities via attributes (Figure 6.11) or via supertype/subtype relationships. Handling the example in Figure 6.11 would require knowing that the value of `age` in the first model corresponds to the value of `a` of an instance of `age_type` in the second (cf. encapsulated entities – Section 4.2.6.5), which is constrained by a corresponding domain rule. This is an example of the sort of

`ENTITY person;`
`  name : STRING;`
`  age : INTEGER;`
`WHERE`
`  {0 <= age < 130};`
`END_ENTITY;`
	`ENTITY person;`
	`  name : STRING;`
	`  age : age_type;`
	`END_ENTITY;`
	
	`ENTITY age_type;`
	`  a : INTEGER;`
	`WHERE`
	`  {0 <= a < 130};`
	`END_ENTITY;`

Figure 6.11: A pair of EXPRESS models to illustrate propagating domain rules from entities to entities.

flexibility desirable in a more sophisticated CCUS.

6.5 Conclusion

This chapter has presented a novel method of comparative code understanding which combines comparison with the extraction of higher-level knowledge to facilitate the inference of abstract correspondences between code-fragments that cannot be seen to correspond at the code level.

There is very little evidence in the existing computer science literature that comparison and extraction have been combined in such a way before – the traditional approach to assessing semantic equivalence being equational reasoning, which suffers from a space/time trade-off, and lacks the benefit of an abstract representation which makes implicit semantics explicit (Section 6.1).

Possible methods of comparative extraction vary along three dimensions (Section 6.2):

- *extraction-driven comparison* or *comparison-driven extraction* (Section 6.2.1);
- *bottom-up/top-down* or *bottom-up/bottom-up* extraction (Section 6.2.2);
- *comparison first* or *extraction first* (Section 6.2.3).

The CCUS works bottom-up/bottom-up and uses comparison-driven extraction, performing localised bouts of extraction (extraction-first) from the vicinity, in both models, of each difference encountered. The HLC CCUS is described in Section 6.3.

Section 6.4 presented a mini-contribution which adds flexibility to the CCUS: how to propagate constraints from defined types to entities, substituting for `SELF`.

One property of the HLC CCUS is that it extracts higher-level knowledge *selectively*, only extracting that which is needed for the purpose at hand: assessing the semantic equivalence of two code-fragments at a higher level of abstraction than the code itself. The HLC CCUS is also selective in its comparison of HLCs (Section 6.3.2), comparing only subsets of the HLCs extracted from the two models. Selective extraction and comparison are the subjects of the next chapter.

Chapter 7

Selective Extraction and Comparison

This chapter follows on from Chapter 6, which discussed the combination of comparison with the extraction of higher-level knowledge, in order to assess the semantic equivalence of code at a higher level of abstraction when it cannot be assessed at the code level. These two chapters are heavily intertwined, but this one constitutes a separate contribution: selective extraction and comparison.

Unlike in most code-understanding systems, this extraction is *selective* – rather than extracting all the knowledge that could be extracted from the codes (EXPRESS models) being compared, only that which is required to resolve differences needs to be extracted (Section 7.1).

The higher-level constraints (HLCs) extracted from the models are compared in order to assess semantic equivalence at an abstract level (Section 6.3.2). Rather than comparing every HLC extracted from one model with every HLC extracted from the other model, HLC-comparison is selective too (Section 7.2). Corresponding HLCs might be extracted from parts of the models which do not correspond structurally, so it is necessary to remember HLCs extracted earlier so that they can be compared with HLCs extracted later. *Remembrance heuristics* control which HLCs are remembered.

7.1 Selective Extraction

Existing code-understanding systems (Section 2.1) tend to extract knowledge *unselectively* from code. That is, given a code, they extract all the knowledge from that code which they know how to extract.

Unselective extraction is all very well when understanding code for the sake of it, but the purpose of code understanding is typically to support reengineering or reuse tasks (Section 2.1). The task that extraction supports does not necessarily require all the knowledge that *could* be extracted.

Selective extraction means extracting knowledge selectively from code. This only makes sense if there is some criterion for selecting which knowledge is extracted and which knowledge is not extracted – namely, that the extraction is driven by the purpose which it is intended to support.

As an example of a selection criterion, extraction could be driven by queries from the user. These queries may call upon the code-understanding system to extract knowledge *bottom-up* from a specified fragment of the code (e.g. what are the HLCs on a given entity?), or *top-down* (e.g. does a given HLC hold on this model?) (Section 2.1.1) – the latter type of queries would arise in bottom-up/top-down comparative code understanding (Section 6.2.2). Indeed, *comparative* extraction could be driven by user queries (as in comparing two given entities, or, given a code-fragment, finding a corresponding code-fragment in the other code). User queries would make a useful test-bed for selective extraction, to test the types of queries that may be posed automatically by systems that use a (comparative) code-understanding system as a building block.

In this work, the selection criterion is that extraction is driven by the needs of comparative code understanding – instead of extracting all the higher-level knowledge that it knows how to extract from two codes, the higher-level Comparative Code-Understanding System (HLC CCUS) extracts higher-level knowledge in order to assess the semantic equivalence of two code-fragments at an abstract level, when it is not possible to do so at the code level. As described in Section 6.3.1, differences at the code level trigger the extraction of higher-level knowledge in localised bouts of extraction from the vicinity of the difference in both models.

Thus the HLC CCUS *passively* achieves selective extraction – it is an intrinsic property of the method of comparative extraction (Section 6.2), rather than achieved actively.

An *active* approach to selective extraction would be to use some form of rules (call them *meta-extraction cases*) that tell it whether to bother extracting each HLC or not. The current system just uses *extraction cases* to tell it what higher-level knowledge to extract from a given code-fragment, with no notion of whether it is appropriate to extract. The HLC CCUS also uses *navigation cases* (Section 6.3.1) which contribute to selective extraction in that they control extraction by telling the HLC CCUS how to walk the code (i.e. inwards) to extract higher-level knowledge.

The use of *meta-extraction cases* would be akin to *relevance reasoning*, "in which a system reasons about which parts of its knowledge are relevant (or irrelevant) to a specific query" with the aim of speeding up inferences [LFS97]. However, such *meta-level reasoning* is only worthwhile if it does not outweigh the cost of the reasoning that it is deciding whether to avoid. Meta-level reasoning comes into its own in combinatorially-explosive problems, where it can be used to prune away branches of enormous search-spaces (such as the use of α-β pruning in chess programs – Section 2.5.2 of [Pra94]). However, the higher-level knowledge that the HLC CCUS can extract from EXPRESS models is shallow: only one or two levels of HLCs, since constraints in EXPRESS models are small and localised.[1] Meta-level reasoning might be more appropriate for cliché-based program understanding (Section 2.1.1), where a program is understood by building up a deep hierarchy of clichés, culminating in the top-level goal of the program. Meta-level reasoning is quite unnecessary for the HLC CCUS, however.

There is *some* evidence that selective extraction has been considered in code understanding:

- Two strategies for (human) program comprehension are systematically examining the code (and the available documentation) to gain a complete understanding of the whole program, and the *as-needed* approach: studying the program only to the extent necessary to gather the information required to solve the problem at hand [CC98].

- Program representations (such as abstract syntax trees and control-flow graphs) can be constructed *on demand* [AG96], so that only those portions of a program that are relevant to a specific analysis are considered, thus saving space and time (program-slicing (Section 2.1.2) is an example of an analysis for which demand-driven construction is appropriate, e.g. to find the statements which affect a given variable). Applying demand-driven

[1] Constraints in an EXPRESS model are *localised* in the sense that they do not tend to combine with other constraints to form a deep hierarchy. This is in contrast to a program, which is typically made up of a hierarchy of tasks and subtasks, from which it is appropriate to extract the overall goal of the program. It would not make sense to attempt to extract the overall goal of an information model by building up a hierarchy of constraints.

construction to the HLC CCUS would imply not only selective extraction of HLCs, but that the frame-representation of the EXPRESS models (Section 3.2.3) could be constructed partially. For example, the source texts of the models could be compared using a basic text comparator such as `diff` [dif] (Section 2.2.1). The differences reported by `diff` would trigger partial construction of the frame-representation, and differences found in the frame-representation would still trigger the extraction and comparison of HLCs. Demand-driven construction would generally be more advantageous the larger the codes being compared, and the sparser the differences.

- Program slicing (Section 2.1.2) is related to selective extraction in that it selects the parts of a code that are relevant to a given slicing criterion, and discards the irrelevant parts. In fact, one way to perform selective extraction would be to slice a program and then to extract knowledge (unselectively) from the slice – only worthwhile if the cost of slicing and extracting does not outweigh the cost of extracting from the whole program.

- The use of indexing to avoid combinatorial explosion in cliché-based code understanding [Qui94] (Section 2.1.1) also relates to selective extraction, although the goal there is to avoid matching code-fragments to clichés that they would not match anyway, rather than selecting which clichés to extract for a specific purpose.

As introduced in Section 6.3.1, *extraneous HLCs* (ones which are extracted but serve no useful purpose for inferring abstract correspondences) are one of the biggest problems for the HLC CCUS, because not only do they waste time to extract and space to store, they tend to build up in the remembrance structures for HLC-comparison, resulting in a quadratic explosion of comparisons. An application which uses the CCUS as a building block might also think there is a semantic difference if it encounters an extraneous HLC that does not correspond to anything in the other model.

The extraction of extraneous HLCs is due to extraction prompts which are too coarse-grained, especially when submitting extraction prompts together rather than separately. Although the most desirable solution would be to eliminate the extraction of extraneous HLCs, their effects can be alleviated by the use of selective remembrance heuristics, discussed in the next section.

7.2 Selective Comparison

The HLC CCUS is also selective in the way that it does not compare every HLC extracted from one model with every HLC extracted from the other model.

This section follows on from Section 6.3.2, which introduced the comparison of the HLCs extracted from two models.

The use of hash-table partitioning (Section 6.3.2) means that comparison is already very selective: each bout of extraction returns the HLCs in a hash-table whose keys are frame-classes, so that only pairs of HLCs of the same frame-class are compared.

Hash-table partitioning thus prevents the HLC CCUS from considering exactly those pairs of HLCs that would be vetoed by the frame-class identicality correspondence heuristic (Section 4.2.2). It also means that frame-class equivalence with a thesaurus (Section 4.2.4) cannot be used for HLCs, since 'synonymous' frame-classes are in different partitions of the hash-table. The possible solution of comparing HLCs from different partitions would defeat the object of hash-table partitioning. Hence the solution adopted is to write the frame-class model (Appendix D, Section D.2) such that comparable HLCs will be of the same frame-class. For example, the HLC-classes formerly known as `exactly_one` and `at_most_one` [Bro99] have been merged into a common frame-class `choice_set` (Appendix A, Section A.1), with an additional attribute `is_optional` (false for 'exactly one', true for 'at most one'). Representing them by a common frame-class has also enabled a semantic-equivalence/strength case (Section 5.3) to be written for `choice_set` (Figure 5.10).

As described in Section 6.3, the HLC CCUS walks the two models in parallel, extracting HLCs in localised bouts of extraction whenever it encounters a difference, and comparing HLCs from corresponding bouts of extraction. This process is illustrated in Figure 7.1.[2] Since comparing two sets of HLCs with cardinalities n and m entails $O(nm)$ HLC-comparisons, it is most efficient to compare HLCs from corresponding bouts of extraction only.

However, it is possible for corresponding HLCs to be in non-corresponding bouts of extraction, due to delocalised differences, as illustrated in Figure 7.2. A *delocalised difference* is one which can be resolved only by comparing HLCs from non-corresponding bouts of extraction. In order to compare HLCs from

[2]This diagram is grossly oversimplified – in particular, it portrays the recursive walking of the models (Section 3.3) as though it were linear.

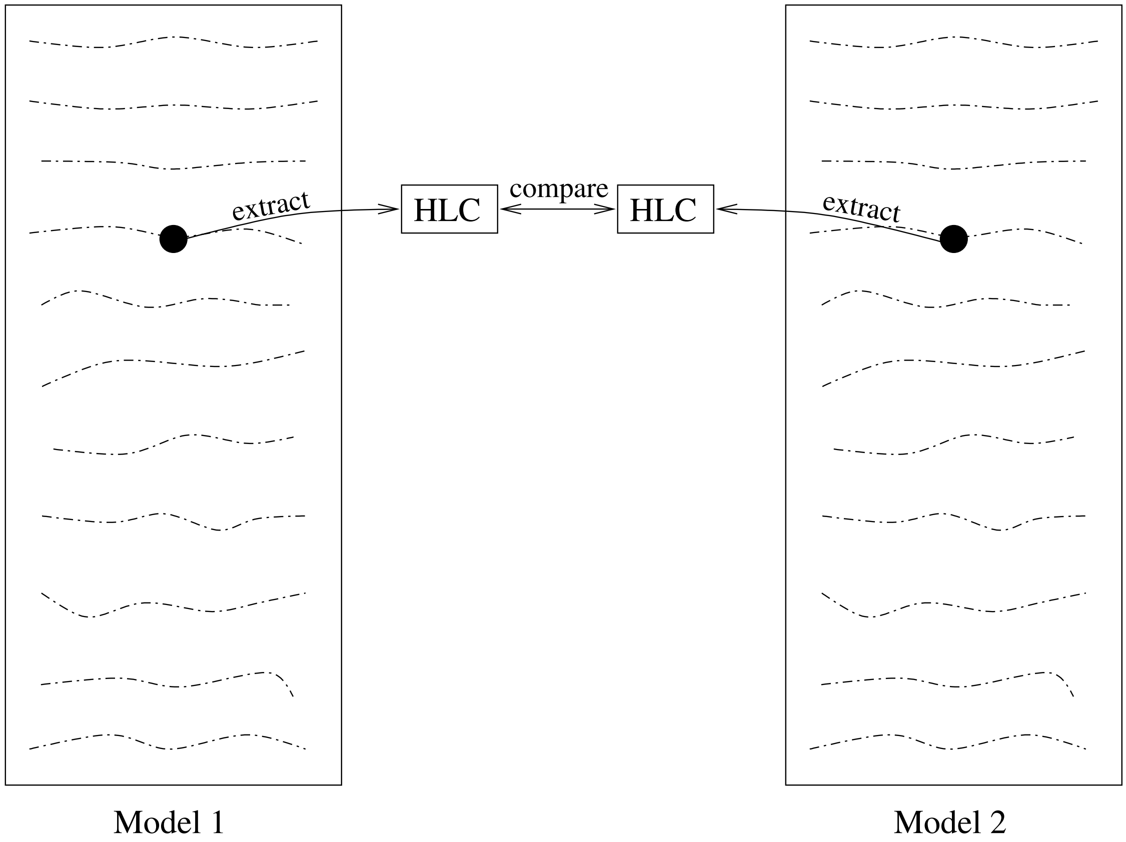


Figure 7.1: Comparing HLCs from corresponding bouts of extraction – a localised difference.

non-corresponding bouts of extraction, HLCs extracted in earlier bouts must be *remembered* so that they can be compared with HLCs extracted in later bouts.

Figure 7.3 gives an example of two EXPRESS models with a delocalised difference. The second model expresses the constraint that `name` must not have a null value, in a way unorthodox but semantically equivalent to the first model. This equivalence can be seen by extracting `non_null` HLCs from both models:

- When comparing the `name` attributes (`explicit_attribute#1` and `explicit_attribute#2`), the CCUS encounters a difference in the `is_optional` slot (`name` is mandatory in the first model but optional in the second). This triggers two corresponding bouts of extraction, from `explicit_attribute#1` and from `explicit_attribute#2` – the HLC `non_null#1` is extracted from the former.

- When comparing the `person` entities (`entity_type#1` and `entity_type#2`),

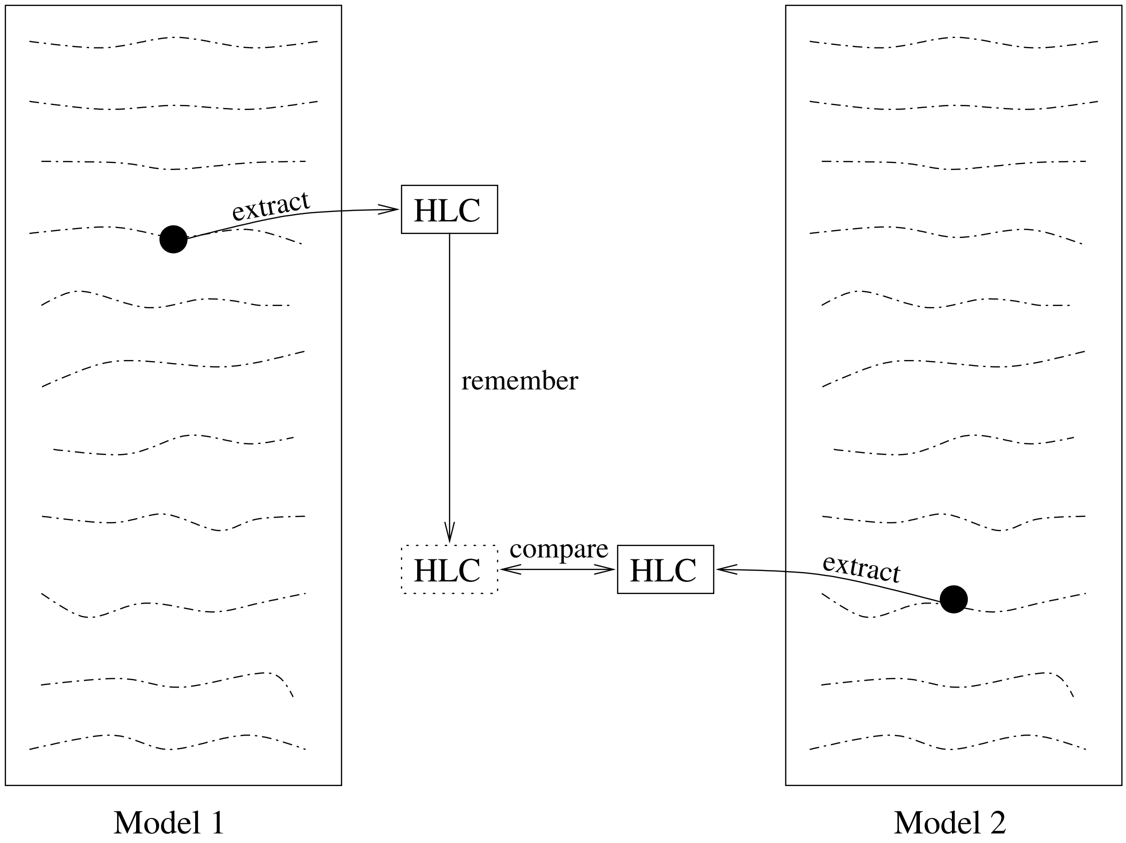


Figure 7.2: Comparing HLCs from non-corresponding bouts of extraction – a delocalised difference.

ENTITY person; name : STRING; END_ENTITY;	ENTITY person; name : OPTIONAL STRING; WHERE EXISTS(name); END_ENTITY;

INSTANCE entity_type#1;
name = "person";
explicit_attributes = {explicit_attribute#1};
END_INSTANCE;
 	
INSTANCE explicit_attribute#1;
name = "name";
tipe = simple_type(string_type);
is_optional = false;
hlcs_extracted = {non_null#1};
END_INSTANCE;
	INSTANCE entity_type#2;
	name = "person";
	explicit_attributes = {explicit_attribute#2};
	domain_rules = {domain_rule#1};
	END_INSTANCE;
	
	INSTANCE explicit_attribute#2;
	name = "name";
	tipe = simple_type(string_type);
	is_optional = true;
	END_INSTANCE;
	
	INSTANCE domain_rule#1;
	expression = function_call#1;
	END_INSTANCE;
	
	INSTANCE function_call#1;
	function_called = built_in_function#6; (* EXISTS *)
	actual_parameters = [explicit_attribute#2];
	hlcs_extracted = {non_null#2};
	END_INSTANCE;

INSTANCE non_null#1; extracted_from = {explicit_attribute#1}; attribute = explicit_attribute#1; END_INSTANCE;	INSTANCE non_null#2; extracted_from = {function_call#1}; attribute = explicit_attribute#2; END_INSTANCE;

Figure 7.3: Two EXPRESS models (and their partial FIL representations) with a delocalised difference.

> the CCUS encounters a difference in the `domain_rules` slot (`person` has a domain rule in the second model that does not have a corresponding domain rule in the first). This triggers a bout of extraction from `domain_rule#1` – which is separate from the corresponding bouts of extraction above – and the HLC `non_null#2` is extracted from `function_call#1`.[3]

For the HLC CCUS to compare `non_null#1` and `non_null#2`, which are extracted from non-corresponding bouts of extraction, it must *memorise* (store in a remembrance structure) the first HLC so that it can compare it with the second HLC when the second HLC is extracted later.[4] This entails storing the HLCs to remember, and merging the remembered HLCs with those from the latest bout of extraction (immediately prior to the process of HLC-comparison).

The extreme approach would be for the HLC CCUS to memorise every HLC it extracts (*unselective remembrance*), but this would be wasteful of space, and even more wasteful of time since comparing sets of HLCs is an $O(nm)$ operation.

[3]A navigation case tells the CUS to submit the expression of a domain rule as an extraction prompt.

[4]The above two HLCs could actually be extracted in either order, depending on whether the CCUS happens to compare the explicit attributes of an entity before or after it compares the entity's domain rules.

It is more efficient to use *remembrance heuristics* that memorise HLCs selectively, discarding some HLCs that, under certain assumptions, will not need to be remembered. Remembrance heuristics are an example of active selectivity, whereas selective extraction in the HLC CCUS (Section 7.1) is passive.

Remembrance heuristics are discussed in the following subsections, starting with the two extremes: no remembrance and unselective remembrance.

7.2.1 No Remembrance

One extreme remembrance heuristic is not to remember HLCs from previous bouts of extraction at all. This has the advantages that it is simple and burden-free, since there are no remembrance structures to get clogged up. It is most efficient for HLC comparison, as the HLC CCUS simply compares pairs of HLCs from corresponding bouts of extraction, without having to merge any remembered HLCs into the sets of HLCs to be compared.

However, with no remembrance, the HLC CCUS misses all delocalised pairs of HLCs, which are in non-corresponding bouts of extraction. For example, in Figure 7.3, the HLC CCUS with no remembrance fails to compare `non_null#1` and `non_null#2`.

Using no remembrance is appropriate for pairs of models that differ only in simple ways, rather than for pairs of models that differ more radically, such that they have delocalised differences. Thus it is likely to be more appropriate for comparing different revisions of the same code, as opposed to transformations or different models of the same domain.

Remembrance is either enabled or disabled for a particular run of the HLC CCUS.

7.2.2 Unselective Remembrance

The other extreme remembrance heuristic is *unselective remembrance*, where all HLCs extracted are memorised. After each pair of bouts of extraction, for each of the two models, the HLC CCUS merges the remembered HLCs for that model with the ones from the latest bout of extraction from that model:

$$r_1' = `r_1 \cup h_1$$

$$r_2' = `r_2 \cup h_2$$

where

- ‘r_n is the remembrance structure for the nth model before it is updated;
- h_n is the set of HLCs extracted in the latest bout of extraction from the nth model;
- r'_n is the remembrance structure for the nth model after it is updated.

The HLC CCUS could compare r'_1 and r'_2 (using the unordered aggregation-matching policy of Section 3.4.1), but that would subsume comparing ‘r_1 and ‘r_2, which have already been compared after previous bouts of extraction. So, to avoid all these repeated HLC-comparisons, the HLC CCUS compares h_1 with r'_2, and compares ‘r_1 with h_2.

The remembrance structures are implemented as hash-tables of HLCs, whose keys are frame-classes, and whose values are sets of HLCs of the same frame-classes. So the above merging and comparison operations actually apply to each set of HLCs within each hash-table, rather than to flat sets of HLCs of different frame-classes.

Unselective remembrance has the advantage that it is complete, in that it is guaranteed not to miss comparing HLCs that correspond.[5] However, it is inefficient in that it has to compare every HLC extracted from each model with every HLC of the same frame-class extracted from the other model, even though this is ameliorated by hash-table partitioning and by avoiding repeated HLC-comparisons. The bigger the models, the more the remembrance structures will get clogged up with HLCs, many of which will never successfully be compared with HLCs extracted later, and the more these stagnant HLCs will be compared futilely with the fresh HLCs extracted from the other model.

Thus the HLC CCUS should use remembrance heuristics to decide whether an HLC is worth storing in the remembrance structure for the model from which it was extracted, and to remove HLCs from the remembrance structures when they are no longer worth remembering. These remembrance heuristics are the subject of the rest of this section.

[5]According to the CCUS's correspondence heuristics – namely frame-class identicality (Section 4.2.2) and semantic equivalence/strength (Section 4.2.5 – HLCs are unnamed objects).

7.2.3 Monogamy

The only selective remembrance heuristic that has been implemented is *monogamy*, which is based on the assumption that HLC-correspondences can only be one-to-one. This is a very reasonable assumption, because polygamous (one-to-many and many-to-many) HLC-correspondences would only occur in cases of semantic redundancy – cf. Section 4.1.2.

Thus, when an HLC is inferred to correspond to another HLC, these two HLCs are *forgotten* (removed from the remembrance structures for their respective models). Note that whether to memorise HLCs in the first place is not an issue for monogamy, since the HLC CCUS memorises the HLCs from the latest pair of extraction bouts before it compares HLCs, therefore all HLCs are single at the time that they are memorised.

Since the remembrance structures are implemented as hash-tables of sets of HLCs, forgetting an HLC is operationalised by removing the HLC from the set for its frame-class, and removing that set from the hash-table if it becomes empty.

Note that the use of monogamy does *not* rule out polygamous correspondences between HLCs from corresponding bouts of extraction – recall from Section 7.2.2 that the HLC CCUS compares h_1 with r'_2, and compares 'r_1 with h_2. When two HLCs are inferred to correspond, they are removed from the remembrance structures and not from the pair of sets of HLCs currently being compared.

Consider Figure 7.4 for an example in which monogamy makes a behavioural difference from unselective remembrance (as opposed to just a performance difference). In short, there is a one-to-two correspondence between the `non_null` HLC that can be extracted from the attribute `age` in the first model (`non_null#1`), and the two `non_null` HLCs that can be extracted from the attribute `age` (`non_null#2`) and the domain rule (`non_null#3`) in the second model. However, whether these HLCs are extracted, and whether they are compared, depends on the following quirky features of the HLC CCUS:

- The reason `age` is of type `INTEGER` in the first model but `REAL` in the second is that if they were of the same type, there would be no difference to trigger extraction. A corollary of this is that the HLC CCUS would think that `EXISTS(age)` does not semantically correspond to anything in the first model. However, the HLC CCUS is not tailored to the detection of semantic redundancy (a task which would best be accomplished by a system which extracts and compares HLCs from a single model, looking for correspondences between HLCs from the *same* model). Bottom-up/top-down comparative extraction (Section 6.2.2) could deal more gracefully with such semantic redundancy, which is fairly unlikely to occur in non-contrived examples anyway.

- Whether the two `non_null` HLCs from the second model are extracted in the same bout depends on whether the CCUS happens to compare explicit attributes or domain rules first:
  - If it compares explicit attributes first, they are extracted in separate bouts: first the explicit attributes (which differ in type) are submitted as extraction prompts and the `non_null` HLCs from the attributes (`non_null#1` and `non_null#2`) are extracted, compared, inferred to correspond, and, under monogamy, forgotten; then the entities (which differ in the existence of domain rules) are submitted as extraction prompts and the `non_null` HLC from `EXISTS(age)` in the second model (`non_null#3`) is extracted. This last HLC is compared and inferred to correspond to `non_null#1` under unselective remembrance, but not under monogamy.
  - If it compares domain rules first, the `non_null` HLCs are extracted in the same bout: the entities (which differ in the existence of domain rules) are submitted as extraction prompts, leading to the extraction of HLCs both from the domain rules and from the explicit attributes. Thus, even under monogamy, correspondences are inferred between `non_null#1` and `non_null#2`, and between `non_null#1` and `non_null#3`.

Monogamous remembrance is almost complete, because polygamous HLC-correspondences are very rare, and will be inferred anyway if the HLCs are in corresponding bouts of extraction. The remembrance structures still get clogged up with uncoupled HLCs, but unselective remembrance is much worse because it never forgets HLCs. The problem with monogamy is that the remembrance structures still get clogged up with HLCs that will never be coupled. The more delocalised differences there are, and the more delocalised they are, the more the HLC-comparison algorithm will be spread by remembered HLCs, and it is still quadratic.

```
ENTITY person;
  age : INTEGER;
END_ENTITY;
```

```
ENTITY person;
  age : REAL;
WHERE
  EXISTS(age);
END_ENTITY;
```

```
INSTANCE non_null#1;
  extracted_from = {explicit_attribute#1};
  attribute = explicit_attribute#1; (* age *)
END_INSTANCE;
```

```
INSTANCE non_null#2;
  extracted_from = {explicit_attribute#2};
  attribute = explicit_attribute#2; (* age *)
END_INSTANCE;

INSTANCE non_null#3;
  extracted_from = {function_call#1}; (* EXISTS(age) *)
  attribute = explicit_attribute#2; (* age *)
END_INSTANCE;
```

Figure 7.4: An example that exposes monogamy.

7.2.4 Remembrance Experiments

Some simple experiments show that the less selective remembrance heuristics increase comparison effort, but can find more correspondences. The following measurements are used to quantify this:

- the number of HLCs extracted from the first model;
- the number of HLCs extracted from the second model;
- the total number of comparisons performed (measured as slot-filler comparisons including frames and aggregation elements) – a measure of comparison effort (fewer comparisons mean greater efficiency);
- the number of HLC-comparisons performed (a subset of the total number of comparisons, in which the fillers being compared are HLC frames);
- the total number of correspondences inferred (not including retracted plausible correspondences) – a measure of effectiveness;
- the number of HLC-correspondences inferred (a subset of the total number of correspondences inferred);
- the number of differences found (when two frames have non-matching values for the same slot) – another important part of the HLC CCUS's effectiveness along with inferring correspondences.

Table 7.1 shows these measurements for comparing the two models in Figure 7.5 using no remembrance (NR), monogamy (M), and unselective remembrance (UR). The correspondence that is missed using no remembrance is between the `non_null` HLC extracted from the domain rule `EXISTS(age)` in the

<pre>SCHEMA experiment1;
  ENTITY person;
    name : STRING;
    age : OPTIONAL INTEGER;
  WHERE
    EXISTS(age);
  END_ENTITY;
END_SCHEMA;</pre>
	<pre>SCHEMA experiment1;
	  ENTITY person;
	    name : STRING;
	    age : INTEGER;
	  END_ENTITY;
	END_SCHEMA;</pre>

Figure 7.5: The first remembrance experiment.

Remembrance heuristic	NR	M	UR
HLCs extracted from first model	2	2	2
HLCs extracted from second model	2	2	2
Comparisons	25	27	27
HLC-comparisons	1	2	2
Correspondences	6	7	7
HLC-correspondences	1	2	2
Differences	2	2	2

Table 7.1: Experimental results for the models in Figure 7.5.

first model and the `non_null` HLC extracted from the attribute `age` in the second model, which are extracted in non-corresponding bouts.

Note that remembrance heuristics can avoid code-level comparisons as well as HLC-comparisons (in Table 7.1, no remembrance performs three fewer comparisons than unselective remembrance, and only two of these are HLC-comparisons). This is because comparing HLCs entails comparing the code-level frames which have frame-references in the HLC frames.

Table 7.2 shows the measurements for comparing the two models in Figure 7.6. The extra differences found under unselective remembrance were just differences encountered in plausible-correspondence mode where the plausible correspondences were retracted, rather than 'true' differences.

Table 7.3 shows the measurements for comparing the two models in Figure 7.7.

Table 7.4 shows the measurements for comparing the two models in Figure 7.8. Once again, the different number of differences are simply due to the differences that cause plausible correspondences to be retracted (when they truly do not correspond, rather than when they do and it fails to infer this) rather than 'real'

```
SCHEMA experiment2;
  ENTITY person;
    name : STRING;
    age : INTEGER;
  WHERE
    (age >= 0) AND (age < 130);
  END_ENTITY;
END_SCHEMA;
```

```
SCHEMA experiment2;
  ENTITY person;
    name : STRING;
    age : INTEGER;
  WHERE
    {0 <= age < 130};
  END_ENTITY;
END_SCHEMA;
```

Figure 7.6: The second remembrance experiment.

Remembrance heuristic	NR	M	UR
HLCs extracted from first model	3	3	3
HLCs extracted from second model	3	3	3
Comparisons	59	59	59
HLC-comparisons	7	7	7
Correspondences	12	12	12
HLC-correspondences	3	3	3
Differences	3	3	3

Table 7.2: Experimental results for the models in Figure 7.6.

```
SCHEMA experiment3;
  ENTITY person;
    middle_names : LIST [0 : ?] OF STRING;
  END_ENTITY;
END_SCHEMA;
```

```
SCHEMA experiment3;
  ENTITY person;
    middle_names : OPTIONAL LIST [1 : ?] OF STRING;
  END_ENTITY;
END_SCHEMA;
```

Figure 7.7: The third remembrance experiment.

Remembrance heuristic	NR	M	UR
HLCs extracted from first model	3	3	3
HLCs extracted from second model	2	2	2
Comparisons	29	29	29
HLC-comparisons	2	2	2
Correspondences	8	8	8
HLC-correspondences	2	2	2
Differences	3	3	3

Table 7.3: Experimental results for the models in Figure 7.7.

```
SCHEMA experiment4;
  ENTITY person;
    name : STRING;
    age : INTEGER;
    gender : gender_type;
  END_ENTITY;

  TYPE gender_type = ENUMERATION OF (male, female);
  END_TYPE;
END_SCHEMA;
```

```
SCHEMA experiment4;
  ENTITY person ABSTRACT SUPERTYPE OF (ONEOF (man, woman));
    name : STRING;
    age : INTEGER;
  END_ENTITY;

  ENTITY man SUBTYPE OF (person);
  END_ENTITY;

  ENTITY woman SUBTYPE OF (person);
  END_ENTITY;
END_SCHEMA;
```

Figure 7.8: The fourth remembrance experiment.

Remembrance heuristic	NR	M	UR
HLCs extracted from first model	10	10	10
HLCs extracted from second model	8	8	8
Comparisons	79	91	99
HLC-comparisons	17	25	33
Correspondences	14	14	14
HLC-correspondences	7	7	7
Differences	22	26	26

Table 7.4: Experimental results for the models in Figure 7.8.

differences.

See Appendix F for fuller versions of these four experiments (not just focusing on remembrance) (Sections F.5 to F.8), and for more experiments which demonstrate the relationships between remembrance heuristics, comparison effort and correspondences inferred.

These results show that the comparison effort increases from no remembrance to monogamy to unselective remembrance, but while remembrance enables correspondences to be inferred that cannot be inferred using no remembrance, monogamy rarely detracts from the correspondences that can be inferred using unselective remembrance.

7.2.5 Other Remembrance Heuristics Not Implemented

While only no remembrance, unselective remembrance and monogamy have actually been implemented, there are more remembrance heuristics that could be used.

Locality is a remembrance heuristic that would discard HLCs when the CCUS moves out of the scopes from which they were extracted, to avoid (or rather, localise) the remembrance structures getting clogged up with stagnant HLCs, which is a problem for unselective remembrance and monogamy. For example, *entity-locality* would remember HLCs only while the CCUS is comparing the pair of entities from which they are extracted; when it finishes comparing those entities, it would forget those HLCs that were extracted while comparing those entities. This is not as narrow as it sounds, because of the recursive way that the CCUS walks the models it is comparing. This means that it could remember HLCs while it walks other entities that are subtypes, supertypes, or the types of the attributes of the pair of entities it is comparing. The remembrance structures would thus be represented as hash-table stacks,[6] to facilitate the forgetting of HLCs when popping out of comparing a pair of entities – an instance of chronological backtracking using a stack-oriented context mechanism (Section 5.1.3 of [FdK93]).

Locality would achieve further selectivity, especially when used in conjunction with monogamy. The larger the models, the more advantageous locality would be, in terms of forgetting stagnant HLCs. However, locality is quite inflexible, and would miss HLCs that fall outside its spreading beam (ultra-delocalised differences), and thus locality would be significantly less complete than unselective remembrance, and than monogamy. However, the lack of inter-entity HLCs in the current implementation means that entity-locality should be complete anyway.

More sophisticated and flexible remembrance heuristics could use other criteria for forgetting HLCs, and also for deciding whether to memorise HLCs in the first place (which the remembrance heuristics so far discussed do unselectively). Another criterion for forgetting HLCs is *recency* – an HLC could be forgotten after it has been waiting in a remembrance structure for a specified period of time, or until it is joined by a specified number of HLCs.

Heuristics for deciding whether to memorise an HLC in the first place need to be based on factors that are predictive of whether an HLC is likely to correspond to an HLC that will be extracted later. While the only way the HLC CCUS can know whether a given HLC will correspond to an HLC extracted later is to try it and see, it could use memorisation heuristics such as *case-based*

[6] A *hash-table stack* is a stack of hash-tables, with methods to push and pop hash-tables, to store a (key, value) pair in the top hash-table, and to look up a key in each hash-table on the stack (Section D.2 of [Bro99]).

[Kol93] (Section 2.3.3): memorise this HLC if memorising a similar HLC in similar circumstances was fruitful. Thus the HLC CCUS could learn from experience which HLCs to memorise and which to discard. Of course, such sophisticated heuristics are difficult to implement, and could well outweigh the costs of having stagnant HLCs in the remembrance structures, hence no attempt has been made to implement them at present.

It would be desirable not to memorise extraneous HLCs (Section 6.3.1), but unfortunately it is too difficult to formalise how to detect extraneous HLCs immediately after they have been extracted.[7] The best way to alleviate the problem of extraneous HLCs would be to improve selective extraction to avoid extracting them in the first place.

In summary of Section 7.2, there are several remembrance heuristics which have been or could be implemented, and they form a spectrum in accordance with the trade-off between the completeness of inferring HLC-correspondences, and the degree of selectivity (hence time and space efficiency):

1. At one end of the spectrum is *no remembrance*, which is the most selective and the least complete.

2. *Locality* would achieve selectivity by remembering HLCs only during the comparison of localised parts of the models, at the expense of a significant loss of completeness. *Recency* (forgetting HLCs after a specified period of time in the remembrance structures) would have a similar but more haphazard effect.

3. *Monogamy* achieves selectivity by assuming that HLC-correspondences are one-to-one and thus forgetting HLCs once they are coupled, at the expense of missing polygamous HLC-correspondences, which only occur in rare cases of semantic redundancy anyway.

4. At the other end of the spectrum is *unselective remembrance*, which achieves completeness at the cost of remembering all HLCs extracted.

[7] A sufficient condition for an HLC to be extraneous is that it is still single at the end of the CCUS run (which means that it has not been used to resolve any differences). This is not a necessary condition for an HLC to be extraneous, since an extraneous HLC could have a corresponding extraneous HLC in the other model.

7.3 Conclusion

This chapter has addressed the issues of selective extraction and selective comparison – two properties of the HLC CCUS that constitute the fourth subcontribution of this thesis.

Section 7.1 discussed the selective extraction of knowledge from code. The vast majority of existing code-understanding systems extract knowledge unselectively, because they are not driven by a specific purpose which the code understanding is intended to support. In contrast, the HLC CCUS extracts only the HLCs that are needed to assess semantic equivalence when it cannot be assessed at the code level. Extraction is triggered by differences that cannot be resolved at the code level, so the selectivity is achieved passively.

Section 7.2 discussed the selective comparison of HLCs. Selective extraction begets selective comparison to the extent that the comparison of extraneous HLCs can be avoided by not extracting them in the first place. HLC-comparison is made even more selective by the use of hash-table partitioning, and by only comparing HLCs from corresponding bouts of extraction. However, it is sometimes necessary to compare HLCs from non-corresponding bouts of extraction in order to resolve delocalised differences. Therefore some HLCs have to be remembered from previous bouts of extraction. This remembrance can be unselective (Section 7.2.2), which is complete (infers all sanctioned HLC-correspondences) but inefficient (performs more HLC-comparisons than necessary), or it can be made more efficient at the expense of completeness by the use of selective remembrance heuristics. The only such remembrance heuristic that has been implemented is monogamy (Section 7.2.3), but there are other remembrance heuristics that could be developed (Section 7.2.5).

This chapter has presented the last of the four subcontributions of this thesis on comparative code understanding – inferring correspondences (Chapter 4), assessing semantic equivalence and semantic strength (Chapter 5), combining comparison and extraction (Chapter 6), and selective extraction and comparison (Chapter 7). The final chapter pulls these four contributions together and concludes the thesis.

Chapter 8

Conclusion

This chapter concludes the thesis by summarising it (Section 8.1), analysing the contributions (Section 8.2), and discussing the potential for future work (Section 8.3).

8.1 Thesis Summary

This thesis concerns comparative code understanding, which it aspires to establish as a field in its own right. Comparative code understanding has its roots in code understanding (Section 2.1) and code comparison (Section 2.2). As well as comparing code, inferring correspondences and reporting differences, it can understand the code at higher levels of abstraction to facilitate seeing that different ways of writing things are semantically equivalent.

Because comparative code understanding is a cross-fertilisation of code comparison and code understanding, it can be considered to contribute to both of these fields – the contribution is mainly *to* code comparison, *using* techniques adapted from code understanding (in particular the cliché-based approaches to extracting higher-level knowledge from code – Section 2.1.1). This thesis imposes a new structure on previous work, bringing previously unrelated areas together.

This thesis focuses on EXPRESS information models as a particular type of code, with particular emphasis on the constraints in the models (the application of code understanding to information models is believed to be novel, since existing work on code understanding (Section 2.1) focuses on program code). The Comparative Code-Understanding System (CCUS) was implemented as an experimental prototype: a test-bed for comparative code understanding. The purpose of the CCUS is to introduce comparative code understanding and show that the ideas can work (proof of concept). The CCUS compares two EXPRESS models, inferring correspondences, reporting differences, and assessing to what extent the constraints in the two models are semantically equivalent, even though they may be expressed in different ways in the two models. The main behaviour of the CCUS is described in Chapter 3.

The CCUS is motivated by the particular application context of schema-to-schema mapping (Section 1.1), and should be judged according to how well it would be able to support the task of transferring constraints in that context (Section 1.3).

Comparative code understanding is the primary contribution of this thesis, which has four subcontributions:

- *Inferring correspondences* between the corresponding parts of two codes (Chapter 4), using *correspondence heuristics* such as frame-class equivalence and name equivalence – which can (at the option of the user) be enhanced by the use of thesauri – and semantic equivalence.
  - A mini-contribution is a method to align a top-level expression (e.g. `age >= 0`) with a corresponding subexpression in the other code (e.g. `(age >= 0) AND (age < 130)`) using *subexpression cases* (Section 4.3).
- *Assessing semantic equivalence and semantic strength* (Chapter 5) between the constraints in two models, using *semantic equivalence/strength (SE/S) cases* to show the equivalence between constraints with irrelevant syntactic differences, or to show that one constraint is stronger than another of its type:
  - two constraints are *semantically equivalent* if whenever one is satisfied, the other is satisfied (e.g. `age >= 0` is equivalent to `0 <= age`);
  - one constraint is *semantically stronger* (more constrained) than another if whenever the first is satisfied then the second is satisfied, but it is possible for the second to be satisfied but not the first (e.g. `age >= 21` is stronger than `age >= 18`);
  - two constraints are *semantically incomparable* if it is neither the case that they are equivalent, nor that one is stronger than the other (e.g. `age >= 0` is incomparable to `age < 130`).

- *Combining comparison with the extraction of higher-level knowledge* (Chapter 6) in order to assess the semantic equivalence of two constraints that are written differently (other than the simple differences that the SE/S cases resolve). When the CCUS encounters differences that it cannot resolve at the code level, it extracts and compares *higher-level constraints (HLCs)*, and infers *abstract correspondences* (code-level correspondences that cannot be inferred at the code level itself) between the constraints if they are indeed assessed to be equivalent at the HLC level. The extraction of HLCs is performed by an existing code-understanding system (CUS) for EXPRESS written by the Author [Bro99]. Combining comparison with extraction is the most important of the four subcontributions.
  - A mini-contribution is a method to propagate constraints from defined types to entities through attributes (Section 6.4).
- *Selective extraction* and *selective comparison* (Chapter 7):
  - Extraction of HLCs is selective in that a comparative code-understander only needs to extract that higher-level knowledge which is needed to infer abstract correspondences (achieved by extracting HLCs in localised bouts of extraction from the vicinity of each difference), rather than all higher-level knowledge which it is capable of extracting.
  - Comparison of HLCs is selective in that rather than comparing every HLC extracted from one model with every HLC extracted from the other model, the CCUS only compares HLCs of the same frame-class (by dint of hash-table partitioning). Furthermore, it only compares HLCs from corresponding pairs of bouts of extraction, plus selected HLCs remembered from earlier bouts of extraction. The CCUS uses *remembrance heuristics*, which forget HLCs that are predicted not to correspond to HLCs from later bouts of extraction. The CCUS can use either *no remembrance*, *unselective remembrance*, or *monogamy* (forgetting an HLC once it is inferred to correspond to an HLC from the other model, based on the assumption that all HLC-correspondences are one-to-one).

The CCUS is the first system to combine all four of these subcontributions (see Table 2.1 for a summary of which subsets of these subcontributions are embodied in existing comparison methods). The contributions are discussed and analysed in the next section.

8.2 Conclusions

This section discusses the extent to which the contributions of this thesis (as declared in Section 1.5) have been achieved. The subcontributions are discussed individually in Sections 8.2.1 to 8.2.4.

First, the following general claims can be made:

- A method for comparative code understanding has been devised.
- A working CCUS has been implemented for EXPRESS information models.
- The advantages of automated comparative code understanding over human comparative code understanding are speed (especially for large codes) and the ability to be systematically complete – especially for models with a high branching factor for entity associations.[1] It is difficult for humans to compare such models because there is a lot to put on the mental stack, so humans would tend to compare pairs of entities in isolation rather than recursively. On the other hand, humans have a knack of 'remembering' their understanding, whereas computers operate one instruction at a time. Both would have trouble with radically different models of the same thing – humans because of the branching factor and delocalised differences, and the CCUS because of inadequate correspondence heuristics, since computers cannot intuitively understand real-world concepts as humans can.
- The CCUS is data-driven, which makes it flexible and extensible: without modifying the program code of the CCUS, its HLC-extraction capabilities can be extended by adding more HLC-classes to the frame-class model (Appendix D, Section D.2) and by adding new extraction cases (Appendices A and B); new SE/S cases (Section 5.3) and subexpression cases (Section 4.3) can be added with equal facility.

[1] i.e. when the type of an attribute is an entity.

- Combinatorial explosion, although a very real threat for bottom-up cliché-based program understanding (Section 2.1.1), has not proved to be a problem for the code-understanding system which extracts HLCs from EXPRESS models, because:
  - each (extraction) case is indexed by the frame-class of the top-level frame in its problem part, so only frames of that frame-class have to be matched to it (Appendix B);
  - many fewer extraction cases are needed for extracting HLCs for the purpose of assessing the semantic equivalence of constraints at a higher level than for program understanding;
  - constraints in EXPRESS models are very localised, so the HLCs extractable from a model form a collection of shallow trees;
  - selective extraction (Section 7.1) further reduces the threat of combinatorial explosion.
- The method can be generalised to other kinds of code, as discussed in Section 8.3.2.1.

The results of the CCUS have not been compared directly with the results of other systems because of its uniqueness. The best that can be done in terms of comparing the current CCUS with other systems is to compare their apparent capabilities, as summarised in Table 2.1 and discussed throughout the literature review in this thesis. To compare results directly would entail adapting the CCUS to perform the specific tasks that other systems perform:

- The CUS (i.e. extraction of higher-level knowledge from a single code) could be adapted to emulate such program-understanding systems as those reviewed in Section 2.1.1, but the purpose of the CCUS is to compare the semantics of code-fragments at a higher-level of abstraction, rather than to extract a complete understanding of the semantics of a code.
- The CCUS could perform schema comparison as in Section 2.2.4, but it would have to be adapted from EXPRESS models to database schemata before its results could be compared with others'.
- The CCUS could be used as a building block for tasks such as program integration and program transformation (Section 2.2.3), data warehousing

(Section 2.3.2), schema-to-schema mapping (Section 2.4.2), information integration and mediation (Section 2.4.3), in order to compare its results with those of the existing systems for performing these tasks.

The most important intrinsic evaluation criteria in comparative code understanding are:

- *soundness*: a sound CCUS would *only* infer correct correspondences, and *only* assess two expressions to be semantically equivalent when they are;
- *completeness*: a complete CCUS would infer *all* correct correspondences, and would *always* assess two expressions to be semantically equivalent when they are.

In the context of transferring constraints for schema-to-schema mapping (Section 1.3), unsoundness could lead to constraints in the target schema being violated because they were not transferred to the source schema, whereas incompleteness would merely lead to unnecessary constraint-checks (due to constraints being transferred from the target schema to the source schema when they already hold on the source schema).

The CCUS errs on the side of soundness – it is only unsound in the matter of name-based correspondence heuristics (Section 4.2), which can always be fooled by homonyms. Other than that, it is sound to the extent of the assumption that syntactic identicality implies semantic equivalence (Sections 3.3 and 4.1.1).

The most important issue is therefore completeness – the more complete the CCUS is, the better it could support tasks such as transferring constraints for schema-to-schema mapping (Section 1.3). Although the CCUS is very conservative, improving completeness does not necessarily mean sacrificing soundness (for example, adding new cases does not detract from soundness, unless of course they contain mistakes), although some techniques (such as the use of thesauri for frame-class and name equivalence – Section 4.2.4) do open new avenues for unsoundness – adding a pair of synonyms may sanction incorrect as well as correct correspondences.

Efficiency is of much lesser concern, since the tasks that the CCUS could support would tend to be 'one-offs' (e.g. transferring constraints for schema-to-schema mapping only needs to be done once: during the generation of the mapping program, which can then be used many times). Efficiency has not been overlooked, however – the threat of combinatorial explosion has been avoided as noted above, and selective extraction and comparison are an efficiency-related contribution.

8.2.1 Inferring Correspondences

The problem addressed in this subcontribution (Chapter 4) is the sound and complete inference of correspondences. A *sound* correspondence heuristic would only infer correspondences between objects that truly correspond, and a *complete* correspondence heuristic would infer correspondences between all objects that truly correspond.

Many comparison methods have to be provided with correspondences rather than inferring them automatically, as summarised in Table 2.1 – see Section 4.1.1 for a discussion of this issue. However, because inferring correspondences cannot be automated in full generality (because the synonymy problem militates against completeness, and the homonymy problem militates against soundness), it is desirable to provide the CCUS with some guidance, such as thesauri.

Frame-class equivalence (Section 4.2.2) is an initial test which can veto but not sanction correspondence between two frames – a necessary but insufficient condition for them to correspond. Frame-class equivalence is sound (vetoes all the correspondences it should) and complete (does not veto any correspondences it should not) *with respect to the frame-class model*; it can be augmented with a frame-class-thesaurus (Section 4.2.4) to allow the frame-class model to be written without having to merge into a common frame-class the frame-classes of frames that could correspond.

Name equivalence (Section 4.2.3) could fundamentally never be sound and complete because of the synonymy and homonymy problems – it could always be fooled. Two objects in an information model truly correspond if and only if they represent the same real-world concept, and names are often the best available indicators of concept, but names can always be abused. To combat the synonymy problem, name equivalence can be augmented with a thesaurus (Section 4.2.4), which makes it more complete, though with potential new avenues for unsoundness.

The thesaurus could be extended to veto homonyms (gaining some soundness at the expense of completeness), but homonyms are actually not common enough in code to justify implementing this; moreover, vetoing homonyms may veto perfectly good correspondences – especially since *WordNet* represents homonyms implicitly (by the fact that a word is in multiple synsets) – and vetoing homonyms would incur the cost of a thesaurus-lookup for every pair of identical names (as well as for every pair of different names).

Name equivalence is a limited approach to inferring correspondence between objects that are supposed to represent the same real-world concept, because it infers correspondences between objects based only on their names. However, the use of model-specific thesauri means that at least the user can make it sanction correspondence between objects that represent the same concept, by declaring their names as synonyms. Nevertheless, it would be desirable to have more sophisticated correspondence heuristics that do not rely on humans to tell them (through the thesaurus) which objects represent the same real-world concept.

The use of thesauri also detracts from efficiency – in terms of the space needed to store the thesaurus, the time needed to load the thesaurus into memory at the start of a CCUS run, and the thesaurus-lookup time incurred by every failure of the name- or frame-class-identicality correspondence heuristic. Thesauri may not be necessary for comparing different revisions or transformations of the same code (Section 1.4.2), but the CCUS would not get far with comparing two very different models without the aid of thesauri.

This thesis distinguishes two ways of inferring correspondences:

- Infer that two objects correspond, and then compare them (as in name equivalence).

- Compare two objects to see whether they correspond (as in semantic equivalence/strength), setting up a *plausible correspondence* which will have to be retracted if they turn out to be semantically incomparable. The recursive structure of the frame-comparison algorithm (Section 3.3) means that just as the inference of correspondences naturally propagates inwards, the retraction of correspondences propagates back outwards.

The use of semantic equivalence and semantic strength as a correspondence heuristic (Section 4.2.5) is sound to the extent of the assumption that if two expressions are syntactically identical then they are semantically equivalent (this assumption does not hold in cases of computational mapping – Section 4.1.1), and provided of course that the SE/S cases (Section 5.3) and extraction cases (Section 6.3) are correct by construction. The soundness and completeness of this correspondence heuristic are the soundness and completeness of assessing semantic equivalence and semantic strength (Section 8.2.2).

Inferring correspondences can fundamentally never be complete because of the synonymy problem, and cannot even be guaranteed to be sound, because of the homonymy problem. In effect, when asked whether two objects correspond, a correspondence heuristic can only answer "yes, I think so" or "not that I know of" (it cannot distinguish between "no" and "I do not know").

The representation of correspondences is limited – it does not say *how* things correspond, just that they do. The introduction of `correspondence` frames (Section 4.1.2) is a step towards saying how things correspond, because it allows correspondences to be tagged with extra information: namely whether an expression is semantically equivalent to, or stronger or weaker than, an expression to which it corresponds. However, it is currently limited in its ability to express more sophisticated correspondences such as partial correspondences, conditional correspondences (e.g. the representation that `person` corresponds to `man` when the value of `gender` is `male`, and to `woman` when the value of `gender` is `female` (Section 6.3.3) seems inadequate), or the complex mappings discussed in Section 4.2.6.5.

Inferring correspondences is experimentally evaluated by comparing the number of correspondences inferred with various features of the CCUS disabled: the use of the thesauri, the use of subexpression cases, the use of SE/S cases, the use of extracting and comparing HLCs (which enables HLC-correspondences and abstract correspondences to be inferred), and the choice of remembrance heuristic for comparing HLCs (the more selective the remembrance heuristic, the fewer comparisons it performs and hence the fewer correspondences it infers), and whether extraction prompts are submitted separately or together (the latter extracts more HLCs and hence infers more correspondences, but extracts many extraneous HLCs). Experimental results are presented in Appendix F.

Such experiments constitute *intrinsic evaluation*: how the various features of the CCUS contribute to its capabilities (particularly inferring correspondence) and performance (how much effort it expends – the number of comparisons in particular). Inferring correspondences should also be evaluated *extrinsically*, by judging whether the CCUS infers the correspondences it should for particular models (completeness), and whether it infers any it should not (soundness), and whether the correspondences it does infer are what are needed for tasks that the

CCUS could support, such as schema-to-schema mapping (Sections 1.1 and 1.3).

The novelty of the contribution of inferring correspondences is the unusual and possibly unique combination of correspondence heuristics. In particular, the use of semantic equivalence and strength *as a correspondence heuristic* is believed to be novel (although Moormann Zaremski and Wing [ZW97] come close – see Section 5.2.2.4).

The contribution of inferring correspondences could generalise to any task which entails comparison or matching (Chapter 2) – particularly code comparison (Section 2.2), which entails name equivalence and semantic equivalence.

8.2.2 Assessing Semantic Equivalence and Strength

The problem addressed in this subcontribution (Chapter 5) is the sound and complete assessment of semantic equivalence and semantic strength.

- Assessing semantic equivalence *soundly* means that two expressions are never assessed to be equivalent (or one assessed to be stronger than the other) if they are in fact incomparable, and that the correct choice between equivalence and strength is made. The CCUS is sound in its assessment of semantic equivalence to the extent of the assumption that syntactically identical expressions are semantically equivalent (Sections 3.3 and 4.1.1).

- Assessing semantic equivalence *completely* means that two expressions are always assessed to be equivalent (or one assessed to be stronger than the other) if they are. Full completeness could never be achieved, and improving completeness is always an ongoing quest.

Semantic equivalence and strength have applications in such areas as behavioural subtyping (Section 5.2.2.2), the semantics of programming languages (Section 5.2.2.3), specification matching (Section 5.2.2.4), and the equivalence and containment of database queries (Section 5.2.2.5).

The CCUS uses SE/S cases (Section 5.3) as the first attempt to resolve a difference between two frames (if this fails to resolve the difference, it extracts and compares HLCs as the second attempt, which will be discussed in the next subsection).

Each SE/S case specifies *necessary slots* and *semantic-strength slots* (all slots not specified are *unnecessary slots*). If two frames differ only in unnecessary slots, then they are semantically equivalent. If two frames differ in a semantic-strength slot (and no necessary slots), then one is stronger than the other – the SE/S case specifies whether the filler of a semantic-strength slot affects semantic strength positively or negatively. If two frames differ in necessary slots, or differ in semantic-strength slots that conflict as to which frame is stronger, then they are incomparable.

SE/S cases are sound by construction, but the set of SE/S cases is incomplete – the set of code-level SE/S cases could in principle be completed (since EXPRESS has a finite number of constructs), but not the set of HLC-level SE/S cases as new HLC-classes could always be invented.

Semantic equivalence and strength propagate from subexpressions to super-expressions, as discussed in Section 5.1.4. Although sound, this propagation is inflexible – for example, it makes sense that `(age >= 0) AND (age < 100)` corresponds to `(age >= 1) AND (age < 101)`, even though technically they are semantically incomparable and thus correspondence between them is vetoed. *Fuzzy* assessment of semantic strength would be desirable in cases such as this (where an expression would be equivalent to (or stronger than) another expression to a numeric degree, rather than being either totally equivalent or not equivalent at all), depending on the task which the CCUS is intended to support:

- Transferring constraints for schema-to-schema mapping (Section 1.3) is a very black-and-white task, for which fuzziness would be inappropriate.
- However, fuzziness would be useful for finding vague overlaps between codes, such as two EXPRESS models having similar though semantically incomparable constraints – it may aid in inferring correspondences between entities with similar constraints in different models of the same domain.

The CCUS also enhances the completeness of assessing semantic equivalence in the way that the frame-representation of EXPRESS models uses a *semantics-oriented representation* which, like standard forms (Section 5.2.3.1), normalises out certain semantically irrelevant syntactic differences (Section 3.2.2):

- commutative and associative expressions such as additions and multiplications are represented as bags (or sets, in the case of conjunctions, inclusive disjunctions and equations);

- a subtraction (division) is represented as an addition (multiplication) whose second operand is negated (reciprocated), i.e. $a - b$ has the same representation as $a + -b$, and c/d has the same representation as $c(1/d)$;

- comparators are represented such that semantically equivalent comparators written different ways round (e.g. $a > b$ and $b < a$) are structurally equivalent (Section 5.3). This drastically reduces the number of FIL-C cases required to handle comparators – both the extraction cases for extracting the `bound` HLC from comparators (Appendix A, Section A.2) and the SE/S cases for the comparators (Section 5.3).

The contribution is that existing comparison methods often ignore the semantics of what they are comparing (as in those methods in Table 2.1 that do not assess semantic equivalence/strength). Schema comparison and schema integration methods (Section 2.2.4) often ignore integrity constraints [Tür99], which are the main focus of the CCUS. As noted in Section 8.2.1, the CCUS is the first known system to use semantic equivalence/strength *as a correspondence heuristic*.

The assessment of semantic equivalence and semantic strength is evaluated by enabling or disabling various features of the CCUS that support it – SE/S cases in particular, and the extraction and comparison of HLCs – to see what effect this has on the number of correspondences inferred (since two code-fragments are inferred to correspond if they are semantically equivalent or if one is stronger than the other, but not if they are semantically incomparable). Experimental results are presented in Appendix F. Assessing semantic equivalence and strength should also be evaluated extrinsically, by judging whether the assessments are sound and complete, and whether they would support applications such as transferring constraints for schema-to-schema mapping (Section 1.3).

In summary, the assessment of semantic equivalence and strength is inherently sound (to the extent of the assumption that syntactically identical code-fragments are semantically equivalent), and its completeness is enhanced by various features of the CCUS.

This thesis also demonstrates another, highly novel approach to assessing semantic equivalence: by extracting and comparing HLCs in order to see at an abstract level that two expressions written differently are equivalent. This constitutes the next of the subcontributions.

8.2.3 Combining Comparison with Extraction

The problem addressed in this subcontribution (Chapter 6) is how to determine that two things written differently are semantically equivalent, at an abstract level when it cannot be inferred at the code level.

As discussed in Section 6.1, there are fundamentally two approaches to assessing semantic equivalence: *equational reasoning* and *extract-and-compare*. This thesis explores the use of extract-and-compare. Extract-and-compare has the following advantages over equational reasoning:[2]

- It represents the semantics of code-fragments at an abstract level, making implicit semantics explicit, so that two semantically equivalent but syntactically different code-fragments are represented in the same terms at the abstract level. This has similarities with standard forms of mathematical expressions (Section 5.2.3.1), but higher-level knowledge is more understandable to humans than the code-fragments it covers, whereas an expression in standard form is typically more abstruse than the original expression.

- Traditional equational reasoning applies only to expressions; extract-and-compare applies to all kinds of code-fragment.

- Extract-and-compare is more efficient in space *and* time (requiring n extraction cases for n ways of writing something) than equational reasoning, which faces a trade-off between space and time (the two extremes are having $n-1$ equivalence laws, all of which have to be applied in the worst case, and having $\frac{1}{2}n(n-1)$ equivalence laws to allow each possible equivalence to be proved in one step).

- Equational reasoning is prone to combinatorial explosion if an equivalence cannot be proved in one step, because it has to try out various ways of rewriting the expressions, and there are exponentially many possible sequences of rewrites. Extract-and-compare, on the other hand, has the focus of working towards the abstract level at which the two code-fragments can be seen to be equivalent.

[2] Combining extract-and-compare with equational reasoning is beyond the scope of this thesis, though doubtlessly a very fertile area of future research (Section 8.3.2.2).

- Extract-and-compare can handle complexity better than equational reasoning for complex code-fragments (e.g. comparing two sorting algorithms).

Section 6.2 categorises the possible methods of comparative extraction along three dimensions:

- *Comparison-driven extraction* was chosen over *extraction-driven comparison* (Section 6.2.1) because extracting all HLCs unselectively from both models and then comparing only the HLCs would be both unnecessary and insufficient (HLCs do not cover the complete semantics of the code as this is not their purpose – their purpose is to facilitate the assessment of semantic equivalence at a higher-level of abstraction).
- *Bottom-up/bottom-up* (BUBU) extraction was chosen over *bottom-up/top-down* (BUTD) extraction (Section 6.2.2) because:
  - The CUS only performs bottom-up extraction. Top-down extraction would be difficult to implement because the code-level frames, which already exist, have to be *matched*, whereas any intermediate HLCs would have to be *generated*, and the choice of ways to express each HLC at the code-level could lead to combinatorial explosion.
  - BUBU is commutative (the HLCs it extracts do not depend on which way round the models are compared) whereas BUTD would miss all HLCs extractable from the second model for which no HLCs are extracted from the first model.
- For each pair of corresponding bouts of extraction, the CCUS works *extraction first* rather than *comparison first* (Section 6.2.3) because there are only one or two levels of HLCs to go up through, intermediate levels of HLCs are *not* discarded for comparison (the CUS returns all the HLCs it extracts), and the extraction-first approach would be better for radically different codes. Another major reason for working extraction first is to maintain the simplicity of the CCUS/CUS interface: the CCUS calls the CUS, which extracts all the HLCs it can from the given extraction prompt inwards – comparison-first extraction would entail a much more complex interaction between the CCUS and the CUS.

Inferring correspondences and assessing semantic equivalence/strength apply at the abstract level as well as at the code level – those contributions are orthogonal to combining comparison and extraction, and it is to them that the issues of soundness and completeness pertain. The set of extraction cases affects the soundness and completeness of assessing semantic equivalence and semantic strength (and hence of inferring correspondences using semantic equivalence/strength as a correspondence heuristic) in that any incorrect extraction cases would compromise soundness, and completeness can be improved (though never fully achieved) by adding extraction cases.

The main weaknesses of the CCUS are that:

- The sets of HLC-classes and of extraction cases are small, with limited coverage of the EXPRESS language. Coverage could be improved by adding more (HLC-classes are identified by analysing worked examples by hand – see Appendix A – which is a demand-driven approach).

- The CCUS is still very inflexible in its comparative-understanding capabilities compared with a human, even though some attempts to add flexibility (such as propagating constraints from defined types to entities – Section 6.4) have been made. Since EXPRESS allows arbitrarily expressive constraints to be written, for example using Pascal-like user-defined functions (Appendix F, Section F.10), there is a long way to go before the CCUS could compete with human understanding abilities.

The extraction and comparison of HLCs is evaluated by running the CCUS with various features enabled or disabled (the extraction of HLCs can be disabled, and there is the choice of which remembrance heuristic to use, and whether to submit extraction prompts separately or together), to compare the number of HLCs extracted from each model and the number of correspondences inferred. Experimental results are presented in Appendix F. These suggest that extraneous HLCs can be a major efficiency barrier even for relatively small models, since they result in a quadratic explosion of comparisons. This problem is significantly alleviated by the use of no remembrance or monogamy as opposed to unselective remembrance of HLCs, and is greatly exacerbated by submitting extraction prompts together – although this can be needed to extract the HLCs needed to infer certain abstract correspondences.

The CCUS is the first known system to combine code comparison and code understanding, although combining comparison and extraction does appear in different guises in computer science (see Section 6.1). The most similar exemplar seems to be the use of semantic enrichment in schema comparison, although it is not clear that semantic enrichment adds any semantics that would not be present in EXPRESS models anyway (semantic enrichment entails tasks such as transforming relational database schemata to object-oriented database schemata [HP96], but EXPRESS already supports subtype hierarchies). However, semantic enrichment could certainly be used to add semantics that *could* be expressed in EXPRESS, but *are not* for a particular model.

Combining comparison with extraction could generalise to any code-comparison methods, which could be augmented with code understanding in order to infer abstract correspondences that cannot be inferred at the code level. In fact, combining comparison with extraction could generalise to any comparison methods, which could be augmented with the extraction of higher-level knowledge in order to see abstract similarities that cannot be seen in surface-level comparison.

8.2.4 Selective Extraction and Comparison

The object of this subcontribution (Chapter 7) is to avoid unnecessary effort in extracting and comparing HLCs – without detracting from completeness, which could happen if the CCUS failed to extract HLCs that are necessary for inferring abstract correspondences.

Selective extraction (Section 7.1) is achieved in a very passive manner, as the CCUS extracts HLCs in localised bouts of extraction from the vicinity of each difference encountered.

The extraction prompts submitted are quite coarse-grained, resulting in some *extraneous HLCs* being extracted (which do not contribute to the inference of any abstract correspondences). To improve selective extraction would be to eliminate or reduce the extraction of extraneous HLCs (provided that the time-cost of doing so does not exceed the benefit).

Selective extraction is evaluated by counting the HLCs extracted from the models being compared, with the number of HLCs extracted by the stand-alone Comparative Code-Understanding System (CUS) [Bro99] as the basis for comparison (Appendix F), to see what selective extraction saves. It should also be evaluated by analysing whether each HLC extracted is extraneous.

Selective extraction begets selective HLC-comparison in that extraneous HLCs cannot be compared if they have not been extracted in the first place. However, the CCUS takes a much more active approach to selective comparison (Section 7.2) than it does to selective extraction:

- The CCUS uses *hash-table partitioning* to ensure that only sets of HLCs of the same frame-class are compared. The frame-class model is written such that all HLCs that could correspond will be of the same frame-class, so as not to sacrifice any completeness.

- The CCUS compares only HLCs from corresponding bouts of extraction, except that it can use *remembrance of HLCs* to include in the comparison selected HLCs from previous bouts. To avoid repeated comparisons, remembered HLCs from one model are only compared with HLCs from the latest bout of extraction from the other model (Section 7.2.2). The following remembrance heuristics have been implemented and experimentally compared (Section 7.2.4):

  - *No remembrance* (Section 7.2.1): No HLCs are remembered from previous bouts of extraction at all, which is the most selective (and hence efficient) remembrance heuristic, but the least complete.
  - *Unselective remembrance* (Section 7.2.2): All HLCs extracted so far are remembered, which is the most complete but least selective remembrance heuristic (it results in more and more useless HLC-comparisons as the remembrance structures get clogged up with stagnant HLCs in the course of comparing two models).
  - *Monogamy* (Section 7.2.3): This heuristic assumes that HLC-correspondences are one-to-one, so when two HLCs are inferred to correspond, it removes them from their respective remembrance structures. As shown in Section 7.2.4 and Appendix F, monogamy significantly reduces the comparison effort with virtually no sacrifice of completeness.
  - Section 7.2.5 discusses other remembrance heuristics, not currently implemented, that could achieve significantly greater selectivity at the expense of completeness. However, one must not lose sight of the fact that completeness is more important than efficiency in comparative code understanding: more sophisticated remembrance heuristics could slow the system down so much that they outweigh the benefit of selective comparison.

Selective extraction is very rare in code understanding, although it does occur in various guises in computer science, often under the banner of *meta-level reasoning* (Section 7.1). The reason it is rare is that it only makes sense when driven by some purpose – in the case of *comparative* code understanding, the only knowledge which needs to be extracted is that which is needed for inferring abstract correspondences. Selective extraction could generalise beyond comparative code understanding by being driven by other purposes, such as user queries for example, or the reengineering or reuse task that the code understanding is intended to support.

Selective HLC-comparison is almost certainly unique, although selective comparison at the code level *could* be performed at the user's request (as in "compare these two entities, not the whole models"). Comparison is also selective in the way code is navigated – it does not entail comparing every frame in one code with every frame in the other code.

8.3 Future Work

This section discusses the future developments that should follow from this thesis. It is categorised into minor improvements (Section 8.3.1), major extensions (Section 8.3.2), and applications of comparative code understanding (Section 8.3.3).

8.3.1 Minor Improvements

8.3.1.1 Further Testing

The most urgently-required further work is to test the CCUS on a wider range of models, since it has, to date, only been tested on the small set of artificial models which have driven development (a representative sample of which are presented in Appendix F). The CCUS needs to be tested on more and bigger models. Evaluation is a major difficulty because it is simply not feasible for a human to judge the CCUS's comparison of large models (whether each correspondence it infers is sound and complete, and useful for the chosen application context of schema-to-schema mapping; the way it reports and resolves differences, and so on). Compiling tables of statistics as in Section 7.2.4 and Appendix F is useful for comparing the performance of the CCUS with various features enabled or disabled, but is meaningless for judging the quality of its performance on a specific pair of models – the only way to do so is to draw up a list of the expected correspondences and differences with which to compare the CCUS's results, and this is only feasible for models up to a certain size, or for selected parts of large models.

The CCUS has been tested on two pairs of 'real' models, as reported in Sections F.13 and F.14.

8.3.1.2 Improving the Coverage of EXPRESS

The completeness of the CCUS (and of the CUS) needs to be improved by adding more HLC-classes and extraction cases for those classes, and more SE/S cases. This would improve coverage of the EXPRESS language (which is finite), and would accommodate more of the possible different ways of expressing the same semantics (which is infinite).

Inter-instance and inter-entity constraints (also known as *inter-object* and *inter-class* constraints respectively – Section 2.5 of [Tür99]) are mostly beyond the scope of the current CCUS, and, for completeness, should be handled:

- An *inter-instance* constraint is one which must be satisfied by a set of instances collectively, rather than by a single instance (for example, the constraint that each instance of an entity must have a unique value for a specified attribute (Appendix A, Section A.7); another example is a global rule for an entity – e.g. Figure 5.6). The CCUS does, in fact, already have an SE/S case for uniqueness rules (Appendix E, Section E.3.9).

- An *inter-entity* constraint is an inter-instance constraint which must be satisfied by a set of instances of more than one entity. In EXPRESS, these occur as global rules for more than one entity (Section A.6).

8.3.1.3 More Flexible Case Matching

Adding new cases can only improve the completeness so far before barriers in the flexibility of the CCUS are reached. For example, more flexible case matching would be required to handle certain examples. This could be achieved by extensions to the FIL-C (case-representation) language (Appendix B, Section B.1), such as:

- the ability to modify frames matched in the *problem* part of a case, other than by filling slots in the solution part which have inverse slots in the problem part (e.g. when HLCs are extracted from a frame, its `hlcs_extracted` and `covered_by` slots could be filled directly – currently, these slots have to be inverse slots, so that they get filled indirectly when the HLCs' `extracted_from` and `covers` slots are filled);

- the ability to specify how a particular slot in the problem part of a case should be matched (for example, which aggregation-matching policy should be used (cf. Section 3.4), noting that this is *CUS case-matching* as opposed to CCUS frame-comparison; another example is whether a null slot-filler should fail the match);

- the ability to specify slots *not* to be matched, to avoid extracting certain extraneous HLCs – for example, the way that the CCUS represents $x = z - y$ as $x = z + -y$ means that not only does it extract an `adder` HLC with `sum` z and `addends` $\{x, y\}$ as desired (Appendix A, Section A.3), it also extracts a second adder with `sum` x and `addends` $\{z, -y\}$, which could be suppressed by specifying `addends <>` {`minus#1`} in addition to `addends = ?addends` in the problem part of the extraction case for `adder` (Appendix E, Section E.1.11);

- the ability to specify constraints in the problem part that must be satisfied for the match to succeed – literally having `WHERE` rules in the frames in the problem part which constrain the variables (e.g. `WHERE ?c > 0.5`, as needed for certainty factors – Section 8.3.2.3), which the case-matching algorithm must check (this entails evaluating expressions);

- the ability to transform slot-fillers matched in the problem part when instantiating the solution part, for example in matching `variable >= ?limit` (where `variable : INTEGER` and `?limit` is an integer constant) and extracting `variable > (?limit - 1)`. This would require FIL-C to support arbitrary expressions and the CUS to evaluate them when extracting the solution part of a case that they match.

It would also be desirable for the EXPRESS code in the problem part of FIL-C cases to be simply written in EXPRESS rather than in the frame-representation of EXPRESS models (Section 3.2.3), but the reason for that is that the solution part needs to refer to frames in the problem part.

8.3.1.4 The Correspondence Problem

The correspondence problem (Chapter 4) is a major obstacle to flexibility and completeness, and even to soundness, because a true correspondence is between objects that represent the same real-world concept, and there is no fail-safe way to infer this from code – soundness and completeness rely on the quality of the correspondence heuristics. Some of the suggestions for other correspondence heuristics (Section 4.2.6) could be implemented. Perhaps the most realistic way to address the correspondence problem would be for the CCUS to accept some guidance from a human – either a partial specification of the correspondences, given before the CCUS run (which it can already take in the form of a thesaurus – for named objects only), or interactively asking the user to sanction or veto particular (selected) correspondences. Ideally it could learn from these experiences so as not to have to ask again, perhaps in a case-based manner [Kol93] – Section 2.3.3.

Improving soundness would entail:

- vetoing homonyms in the name-thesaurus (Section 4.2.4), which is a problem because *WordNet* does not support explicit homonyms – only implicit homonyms (words that are in more than one synset);
- relaxing the assumption that syntactic identicality implies semantic equivalence by supporting computational mapping (Section 4.1.1). The computational mapping would have to be provided as input because it would usually be impossible to infer it automatically. In fact, the name-thesaurus could even be augmented with knowledge of computational mapping, e.g. the synset {`fahrenheit,celsius,`} could be associated with the conversion formula $F = 32 + \frac{9}{5}C$, although *WordNet* does not support such augmentation so another thesaurus system might have to be considered (as discussed in Section 4.2.4).

8.3.1.5 Methods of Comparative Extraction

It would be interesting to implement the different approaches to comparative extraction identified in Section 6.2 (BUTD in particular, although top-down extraction is difficult to implement as noted in Section 8.2.3) and to compare them experimentally rather than merely analytically. Top-down extraction would entail the use of *open variables*[3] for values which are unknown until the growing tree of hypothesised HLCs reaches down to the code level.

8.3.1.6 Improving Selective Extraction and Comparison

Selective extraction could be improved by doing it in a finer-grained (more controlled) manner so as to eliminate or at least reduce the extraction of extraneous HLCs. It would also be interesting to experiment with selective extraction which is driven by other purposes, such as user queries (e.g. extracting HLCs from a given entity). User queries could also be used for selective comparison (e.g. comparing two given entities rather than the whole models).

Selective HLC-comparison could be improved by using other remembrance heuristics such as those discussed in Section 7.2.5, although these are difficult to implement, detract from completeness, and could outweigh the benefit of selective comparison if they are too computationally expensive.

8.3.1.7 User Interface

The CCUS would need a proper user interface if it is to interact with the user (Section 8.3.1.4).

The current CCUS simply takes a command-line which specifies the two models to be compared, an optional thesaurus, and various command-line flags to enable or disable the various features of the CCUS.

The CCUS then compares the models without interruption, reporting correspondences, differences, extraction of HLCs and retraction of plausible correspondences.

- The CCUS output could be improved by clustering these reports by structure, rather than listing them in the order they are inferred by the recursive CCUS frame-comparison algorithm in Section 3.3.

- It would also be desirable to suppress (at the option of the user) the output that the CCUS generates while matching in plausible-correspondence mode, for plausible correspondences that are subsequently retracted (Section 4.2.5). A substantial proportion of the CCUS output tends to pertain

[3] Open variables are discussed in Section 2.3.2 of [Pra94] in the context of backward chaining (of inferences), which is to forward chaining as top-down extraction is to bottom-up extraction.

to plausible correspondences that are inferred during aggregation-matching (Section 3.4) and in most cases subsequently retracted.

At the end of the run, the CCUS optionally dumps the frames in memory as HTML files [MK97] (with hypertext links for frame-references) for the user to browse. It could dump the CCUS output in HTML format, too.

8.3.2 Major Extensions

This subsection discusses some major ways forward for comparative code understanding, as opposed to minor improvements to the CCUS.

8.3.2.1 Languages other than EXPRESS

Comparative code understanding could generalise to codes in languages other than EXPRESS, and indeed other kinds of code than information models:

- Other information-modelling languages such as EXPRESS-2 [SS99], UML [Rat], XML [xml97] (cf. Section 4.2.4) and OML [FGHS98].
- Database schemata and database query languages such as SQL (Chapter 7 of [EN94]). There is a strong duality between queries and constraints, as discussed in Section 5.1.2.
- Specification languages such as VDM (Appendix B of [vV93]) and Z [Spi94] for *model-oriented specifications* in which constraints occur as preconditions, postconditions and invariants, and *algebraic specifications* (in which constraints occur in the axioms).
- Declarative[4] programming languages such as Standard ML [MCP93], LISP [Wil86] and Prolog [CM94]. In this context, comparative code understanding would be about whether two programs compute the same function.
- Imperative[5] programming languages such as Java [Gra97] and C/C++ [CLL92, Str91] (a frame-representation of Java programs had already been implemented before this line of research was put on hold – see [Bro] for the frame-class model). Constraints appear in the guises of the conditions of

[4]i.e. the superclass of functional and logic programming languages.

[5]i.e. the superclass of procedural and object-oriented programming languages.

if-statements,[6] and subclass hierarchies. In comparative code understanding of programs, it is important to concentrate not just on expressions, but on higher-level aspects such as structure, sequencing, types, and interfaces between objects.

- Scanner- and parser-generators such as lex and yacc (for C) [LMB92], and JavaCC [Sun] (for Java). Comparative code understanding of grammars could be used to assess the *syntactic* equivalence of two languages, since the same language can be described by different grammars; moreover, grammars often have to be written in abstruse forms to resolve ambiguities (Section 4.3 of [ASU86]), so code understanding of grammars could restore perspicuity.

Inter-language comparative code understanding is also possible, where the codes being compared are written in different languages (e.g. comparing a Java program with a C++ program, comparing an EXPRESS model with a database schema, comparing a knowledge-base schema with an application data-structure [Pun91] (Section 2.2.2), verifying whether an implementation conforms to an information model [Teg97], and indeed comparing the implementation of an abstract data-type with its algebraic specification [AH00] (Section 2.2.5) can be seen as inter-language comparative code understanding). In fact the extraction of higher-level knowledge facilitates inter-language comparative code understanding, enabling the codes to be compared at a higher level in places where the languages are too different for them to be compared at the code level – that is, a preliminary level of language-independent representation of the codes could be extracted before comparison commences.

8.3.2.2 Equational Reasoning and Exchange Cases

The current CCUS is a test-bed for extract-and-compare, without equational reasoning (Section 6.1). It would improve flexibility, and hence the completeness of assessing semantic equivalence, to augment the CCUS with the ability to perform traditional equational reasoning – the ability to rewrite expressions. Equational reasoning would then enhance rather than replace extract-and-compare, which comes into its own in assessing the equivalence of things other than expressions.

[6] Note that there is a distinction between domain constraints, and conditions that merely constrain the behaviour of the program, whereas all constraints in information models are domain constraints (apart from conditions in the bodies of functions and procedures in EXPRESS).

Equational reasoning could also be extended from expressions to code-fragments in general, as already hinted in Section 6.1. Pairs of extraction cases for the same HLC-class could be combined into a generalisation of equivalence laws called exchange cases.

An *exchange case* associates two interchangeable code-fragments (such as an enumeration type and a `ONEOF` supertype hierarchy). Exchange cases could be used to see that two such code-fragments are equivalent, and they could also be used to perform equivalence-preserving transformations. An exchange case would capture the abstract semantic equivalence of two code-fragments directly. Exchange cases could still extract HLCs as a by-product, and thus still build up hierarchies of higher-level knowledge – really extracting from both models together as opposed to rigidly extracting HLCs from each model in isolation and then comparing the HLCs. Possible benefits of exchange cases include:

- Exchange cases could drive the extraction and comparison of HLCs, possibly avoiding the time-penalties of delocalised differences by 'gluing' corresponding bouts of extraction together.

- Exchange cases may be able to eliminate the extraction of extraneous HLCs by more fine-grained control of extraction; they might even be able to tell the CCUS where in the other model to look for a corresponding HLC.

- Exchange cases could just state that two code-fragments are equivalent without extracting HLCs, when it is difficult, unnatural or not useful to characterise the abstract semantics. Sometimes it is more natural just to specify that two expressions are equivalent and leave it at that, e.g. that for integers, $x > c$ is equivalent to $x \geq c + 1$.

- Exchange cases could be used to set abstract correspondences (Section 6.3.3). This would be a more direct and natural way to do it than the current method of inferring correspondence between any frames covered by a pair of HLCs when they are inferred to correspond.

FIL-C and the CUS case retrieval/matching algorithms would have to be extended to support exchange cases, as the problem part would consist of two frame-networks that the code-fragments from the two models must match, rather than just one frame-network in the problem part and a simple retrieval scheme that indexes cases by the frame-class of the top-level frame in the problem part.

```
ENTITY hotel;
  number_of_floors : INTEGER;
  bedrooms : SET [0 : ?] OF bedroom;
WHERE
  number_of_floors >= 1;
END_ENTITY;

ENTITY bedroom;
  room_name : STRING;
  floor_number : INTEGER;
INVERSE
  the_hotel : hotel FOR bedrooms;
UNIQUE
  room_name;
WHERE
  {1 <= floor_number <= the_hotel.number_of_floors};
END_ENTITY;
```

```
ENTITY hotel;
  floors : LIST [1 : ?] OF floor;
END_ENTITY;

ENTITY floor;
  bedrooms : SET [0 : ?] OF bedroom;
INVERSE
  the_hotel : hotel FOR floors;
END_ENTITY;

ENTITY bedroom;
  room_name : STRING;
INVERSE
  the_floor : floor FOR bedrooms;
UNIQUE
  room_name;
END_ENTITY;
```

Figure 8.1: Two EXPRESS models of a hotel.

```
ENTITY book;
  name : STRING;
  author : STRING;
  number_of_copies : INTEGER;
END_ENTITY;
```

```
ENTITY book;
  name : STRING;
  author : STRING;
  physical_location : STRING;
END_ENTITY;
```

Figure 8.2: EXPRESS models of a book and of a copy of a book.

Exchange cases would have to be indexed by the top-level frames of both sides of the problem part, which would complicate case retrieval and matching.

8.3.2.3 Extracting Deeper Meaning from Code

Another major extension would be to extract deeper meaning from code – semantics which EXPRESS does not express, but which can be inferred by taking other knowledge into account (such as names, and real-world knowledge from an ontology[7]). For example:

- Consider the first model in Figure 8.1. The attributes `number_of_floors` and `floor_number` are both of type `INTEGER` but have very different meanings: the former specifies the number of floors, while the latter specifies a specific floor.

- Consider both models in Figure 8.1. The second model represents floors

[7] A thesaurus can be thought of as a cut-down ontology.

as an explicit entity `floor`, whereas the first model represents floors implicitly through the attributes `number_of_floors` and `floor_number`, and the domain rule {`1 <= floor_number <= the_hotel.number_of_floors`}. The two models are semantically equivalent in the sense that no constraints could be violated by mapping in either direction (Section 1.2).

- Another example of extracting deeper meaning is interpreting whether an entity represents a 'definition' or an 'individual'. For example, the first model in Figure 8.2 represents a book, whereas the second model represents a copy of a book (the entity names are homonyms). Humans would make this distinction by applying real-world knowledge to understand the attributes `number_of_copies` and `physical_location`. The distinction would be even more apparent if the first model had an explicit entity `book_copy`, and if the attribute `number_of_copies` was instead `copies : SET [1 : ?] OF book_copy`. Note that the distinction between 'definition' and 'individual' is relative – an entity `book` may represent a specific book, a specific edition of a book, or a specific copy of an edition of a book.

- Another example of extracting deeper meaning is interpreting the semantics of relationships. EXPRESS is very weak in this regard – apart from supertype/subtype relationships, they are all represented by the attributes of entities, which may themselves be entities. There is no explicit distinction between *containment* (*aggregation*) (e.g. "a garden has a greenhouse") and *reference* (*association*) (e.g. "a student has a supervisor"). Constraints such as inverse attributes may provide clues to the nature of the relationship (the inverse attributes in Figure 8.1 suggest that the relationship between a hotel and its bedrooms may be containment, since a bedroom cannot exist without a hotel), but to do the interpretation properly requires real-world knowledge.[8]

- A relationship may actually be represented by an entity, so that the relationship itself may have attributes (cf. the two different ways that the CCUS represents correspondences – Section 4.1.2).

To extract such meanings is a very non-trivial task which can only be done heuristically, i.e. it is open to error because a method for doing so could always be

[8] "Ruth is part of an orchestra, and Ruth's liver is part of her, so is Ruth's liver part of the orchestra? What instrument does it play?" (Ian Pratt).

fooled. Such heuristics might take names into account (e.g. `number_of_Xs` versus `X_number`), the structure of the model, and the constraints in the model – but these are only 'evidence' to support particular interpretations, not 'proof' in the sense of all the currently-implemented extraction cases. It is an ill-defined task which could be tackled using artificial-intelligence techniques such as case-based reasoning [Kol93] (Section 2.3.3), which would allow it to use previous experiences and to learn from experience.

Because of the fuzziness of extracting such HLCs, they could be tagged with numeric *certainty factors* between 0 and 1 – each extraction case could have a certainty factor which is assigned to any HLCs it extracts, but if the problem part of the case is matched by HLCs with certainty factors less than 1, they would combine multiplicatively with the certainty factor of the extraction case to reduce the certainty factor of the extracted HLCs. This is just like the way that the rule-based system MYCIN uses certainty factors for propositions and rules (Section 8.2 of [RK91]).

8.3.2.4 Automatic Case Acquisition

It may even be possible for the CCUS to learn HLC-classes and their extraction cases automatically (cf. concept learning – Section 7.4 of [Pra94]). As it is, HLC-classes and extraction cases are identified by a human, by comparing EXPRESS models by hand, or thinking of different ways to express a constraint (Appendix A). This requires knowledge of the semantics of EXPRESS and of the real-world concepts being modelled. Such knowledge could be captured as an ontology, although to some extent having an ontology of real-world concepts is a 'chicken-and-egg' problem because EXPRESS models themselves describe real-world concepts. Of course such knowledge could be used to perform comparative code understanding directly, but it would be more efficient to apply learned extraction cases than to work it out from scratch every time.

A major difficulty with automatically acquiring cases from frame-networks is how to know where the boundaries of the case are (cf. extraction-time substitution in Section 6.4; see also Section 5.4.3 of [Kol93]).

8.3.3 Applications of Comparative Code Understanding

Implementing applications which use the CCUS as a building block constitutes the third major category of future work.

The application which has motivated this research from the start is schema-to-schema mapping (Section 1.1), with the idea that constraints could be transferred at compile-time from the target schema to the source schema (operationalised by planting constraint-checks into the mapping program) so that instances in the source repository can be checked, prior to mapping them to the target repository, that they do not violate constraints on the target (Section 1.3).

Transferring constraints for schema-to-schema mapping is an example of *comparative code generation*: modifying one (or both) of two existing codes to bridge the semantic gap, which entails comparative code understanding as a subprocess.

Another application would be comparing a code with a transformation of that code (Section 1.4.2.1) to assess the extent to which the transformation has preserved the (constraint) semantics of the original code. If semantic differences are found, comparative code generation could then fix the transformed code (or even the transformation algorithm) to respect the semantics of the original code. This would require the CCUS to be sound and complete with respect to the specific classes of transformations to be handled (unfortunately, soundness and completeness are unachievable in general (Section 8.2), so a sound and complete generic solution for all possible transformations is not viable).

- If the CCUS is unsound in its assessment of semantic equivalence, a transformation could be reported to respect the semantics of the original code when it does not.

- If the CCUS is incomplete in its assessment of semantic equivalence, a transformation could be reported not to respect the semantics of the original code when it does, hence the transformation could be modified unnecessarily (introducing semantic redundancy).

As discussed in Section 8.2, the CCUS could also be used as a building block for systems which perform other tasks, to enhance the capabilities of existing systems which perform these tasks.

8.4 Epilogue

By bringing together ideas from code understanding and comparison, inferring correspondences and assessing semantic equivalence, this thesis has sought to establish a firm foundation for the future fields of comparative code understanding and comparative code generation.

Bibliography

[AFMS96] D. Abramson, I. Foster, J. Michalakes, and R. Sosič. Relative debugging: A new methodology for debugging scientific applications. *Communications of the ACM*, 39(11):69–77, November 1996.

[AG96] D.C. Atkinson and W.G. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering (ICSE-96), Berlin, Germany, 25th–29th March 1996*, pages 16–27, 1996. `http://www.cse.scu.edu/~atkinson/papers/icse-96.pdf` (modified 13th October 1999).

[AH00] S. Antoy and D. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, January 2000.

[AIS93] R. Agrawal, T. Imielinski, and A.N. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 1993), Washington, District of Columbia, United States of America, 26th–28th May 1993*, pages 207–216. ACM Press, 1993.

[AK00] J.L. Ambite and C.A. Knoblock. Flexible and scalable cost-based query planning in mediators: A transformational approach. *Artificial Intelligence*, 118(1–2):115–161, April 2000.

[AP94] A. Aamodt and E. Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AICom – Artificial Intelligence Communications*, 7(1):39–59, March 1994. http://www.iiia.csic.es/People/enric/AICom.html (modified 1st December 1997).

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, United States of America, 2nd edition, 1986. ISBN 0-201-10194-7.

[Bai95] I. Bailey. *EXPRESS-M User Guide.* Cimio Corporation, July 1995. http://www.cimio.com/Documents/STEP/exm_ug.html (modified 9th October 1998).

[BF99] A. Broad and N. Filer. Applying case-based reasoning to code understanding and generation. In I. Watson, editor, *Proceedings of the 4th UK Case-Based Reasoning Workshop (UKCBR4), University of Salford, Salford, England, 15th September 1999*, pages 35–48, 1999. http://www.cs.man.ac.uk/~broada/cs/papers/ukcbr4.ps.gz (modified 17th September 1999).

[BF00] A. Broad and N. Filer. Extracting constraint knowledge from code: A case-based reasoning approach. In M. Bramer, A. Macintosh, and F. Coenen, editors, *Research and Development in Intelligent Systems XVI: Proceedings of the 19th SGES International Conference on Knowledge Based Systems and Applied Artificial Intelligence (ES99), University of Cambridge, Cambridge, England, 13th–15th December 1999*, BCS Conference Series, pages 83–96, London, England, 2000. Springer. ISBN 1-85233-231-X. http://www.cs.man.ac.uk/~broada/cs/papers/es99.ps.gz (modified 20th October 1999).

[BFM95] M. Brown, N. Filer, and Z. Moosa. KRU – Generic operators for schema-to-schema mapping. Technical Report JCF/MAN/108-01, Department of Computer Science, University of Manchester, April 1995.

[BHR95] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, January 1995.

[BLN86] C. Batini, M. Lenzerini, and S.B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[BM77] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

[BMW94] T.J. Biggerstaff, B.G. Mitbander, and D.E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–83, May 1994.

[Bro] A. Broad. *Andrew Broad's Website: Computer Science: PhD Project*. Department of Computer Science, University of Manchester. `http://www.cs.man.ac.uk/~broada/cs/phd/` (modified 21st September 2002).

[Bro99] A.P. Broad. The application of case-based reasoning to the understanding of constraints on information models. MPhil thesis, Department of Computer Science, University of Manchester, 1999. `http://www.cs.man.ac.uk/~broada/cs/mphil/thesis/` (modified 15th May 2002).

[CAD92] CAD Framework Initiative (CFI), Inc., Austin, Texas, United States of America. *Design Representation Programming Interface: Electrical Connectivity – Version 1.0.0*, 1992. ISBN 1-882750-01-2.

[CC98] G. Canfora and A. Cimitile. Program comprehension. Technical report, Faculty of Engineering at Benevento, University of Sannio, May 1998. `http://www.ing.unico.it/ftp/GII/canfora-testo.pdf` (modified 24th September 1998).

[CGM97] S.S. Chawathe and H. García-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 1997), Tucson, Arizona, United States of America, 13th–15th May 1997*, pages 26–37. ACM Press, 1997.

[CI90] E.J. Chikofsky and J.H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[Cla85] W.J. Clancey. Heuristic classification. *Artificial Intelligence*, 27(2):289–350, November 1985.

[Cle91] A. Clements. *The Principles of Computer Hardware.* Oxford University Press, Oxford, England, 2nd edition, 1991. ISBN 0-19-853764-6.

[CLL92] C. Charlton, P. Leng, and J. Little. *A Course on C.* McGraw-Hill, London, England, 1992. ISBN 0-07-707433-5.

[CLM95a] A. Cimitile, A. De Lucia, and M. Munro. Identifying reusable functions using specification driven program slicing: A case study. In *Proceedings of the IEEE International Conference on Software Maintenance, Nice, France, 17th-20th October 1995*, pages 124–133, 1995.

[CLM95b] A. Cimitile, A. De Lucia, and M. Munro. An overview of structural and specification driven candidature criteria for reuse reengineering processes. Technical Report 7/95, Department of Computer Science, University of Durham, 1995.

[CM94] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, Germany, 4th edition, 1994. ISBN 3-540-58350-5.

[Cog] Cognitive Science Laboratory, Princeton University. *WordNet – a Lexical Database for English.* `http://www.cogsci.princeton.edu/~wn/` (accessed 9th October 2002).

[Col] CollabNet. *Concurrent Versions System.* http://www.cvshome.org/ (accessed 20th September 2001).

[CPM96] R. Cochrane, H. Pirahesh, and N. Mattos. Integrating triggers and declarative constraints in SQL database systems. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB 1996), Mumbai (Bombay), India, 3rd–6th September 1996*, pages 567–577, 1996.

[CTL97] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report #148, Department of Computer Science, University of Auckland, July 1997. http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow97a/ (modified 13th June 1999).

[CV94] S. Chaudhuri and M.Y. Vardi. On the complexity of equivalence between recursive and nonrecursive Datalog programs. In *Proceedings of the 13th ACM-SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1994), Minneapolis, Minnesota, United States of America, 24th–26th May 1994*, pages 107–116, 1994.

[dif] *UNIX man pages : diff ().* http://unixhelp.ed.ac.uk/CGI/man-cgi?diff (accessed 15th September 2001).

[DKM+93] P. Drew, R. King, D. McLeod, M. Rusinkiewicz, and A. Silberschatz. Report of the Workshop on Semantic Heterogeneity and Interoperation in Multidatabase Systems. *ACM SIGMOD RECORD*, 22(3):47–56, September 1993.

[DS99] P.A. Dearnley and D.J. Smith. Discovering and using entity mappings in federated databases. *Software–Practice and Experience*, 29(1):17–42, January 1999.

[Ede97] H. Edelstein. Data mining: Exploiting the hidden trends in your data. *DB2® online magazine*, 2(1), Spring 1997. http://www.db2mag.com/db_area/archives/1997/q1/9701edel.shtml (accessed 9th October 2002).

[EN94] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, California, United States of America, 2nd edition, 1994. ISBN 0-8053-1753-8.

[Fan02] J. Fang. Model based data translation. MPhil thesis, Department of Computer Science, University of Manchester, 2002.

[FB94] R.W. Floyd and R. Beigel. *The Language of Machines: An Introduction to Computability and Formal Languages*. Computer Science Press: An imprint of W.H. Freeman and Company, New York, New York, United States of America, 1994. ISBN 0-7167-8266-9.

[FCGCGAHY96] C. Fernández-Chamizo, P.A. González-Calero, M. Gómez-Albarrán, and L. Hernández-Yáñez. Supporting object reuse through case-based reasoning. In I. Smith and B. Faltings, editors, *Advances in Case-Based Reasoning: Proceedings of the 3rd European Workshop (EWCBR-96), Lausanne, Switzerland, November 1996*, volume 1168 of *Lecture Notes in Computer Science*, pages 135–149, Berlin, Germany, 1996. Springer-Verlag. ISBN 3-540-61955-0.

[FdK93] K.D. Forbus and J. de Kleer. *Building Problem Solvers*. MIT Press, Cambridge, Massachusetts, United States of America, 1993. ISBN 0-262-06157-0.

[FFD96] S. Fäustle, M.G. Fugini, and E. Damiani. Retrieval of reusable components using functional similarity. *Software–Practice and Experience*, 26(5):491–530, May 1996.

[FFG89] B. Falkenhainer, K.D. Forbus, and D. Gentner. The structure-mapping engine: Algorithm and examples. *Artificial Intelligence*, 41(1):1–63, November 1989.

[FGHS98] D.G. Firesmith, I. Graham, and B. Henderson-Sellers. *Open Modeling Language (OML) Reference Manual*. Cambridge University Press, 1998. ISBN 0521648238.

[FO96] C. Faloutsos and D. Oard. A survey of information retrieval and filtering methods. Technical Report CS-TR-3514, University of Maryland, 1996.

[FOW87] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[FPSM92] W.J. Frawley, G. Piatetsky-Shapiro, and C.J. Matheus. Knowledge discovery in databases: An overview. *AI Magazine*, 13(3):57–70, Fall 1992.

[GB99] P. Gomes and C. Bento. Converting programs into cases for software reuse. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99), Stockholm, Sweden, 31st July – 6th August 1999*, 1999. `http://eden.dei.uc.pt/~pgomes/Documents/ijcai99ws.pdf` (accessed 9th October 2002).

[GBMS99] C.H. Goh, S. Bressan, S. Madnick, and M. Siegel. Context Interchange: New features and formalisms for the intelligent integration of information. *ACM Transactions on Information Systems*, 17(3):270–293, July 1999.

[GKD97] M.R. Genesereth, A.M. Keller, and O.M. Duschka. Infomaster: An information integration system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 1997), Tucson, Arizona, United States of America, 13th–15th May 1997*, pages 539–542. ACM Press, 1997.

[Gra97] M. Grand. *Java Language Reference.* O'Reilly & Associates, Sebastopol, California, United States of America, 1997. ISBN 1-56592-326-X.

[GSUW94] A. Gupta, Y. Sagiv, J.D. Ullman, and J. Widom. Constraint checking with partial information. In *Proceedings of the 13th ACM-SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1994), Minneapolis, Minnesota,* *United States of America, 24th–26th May 1994*, pages 45–55, 1994.

[Hak98] S. Hakkarainen. Dynamic aspects and semantic enrichment in schema comparison. In *Proceedings of the Conference on Advanced Systems Engineering (CAiSE*98): 5th Doctoral Consortium, Pisa, Italy, 8th–9th June 1998*, 1998. `http://www4.informatik.tu-muenchen.de/~thurner/CAiSE98DC/Proceedings/hakkarainen.ps` (accessed 9th October 2002).

[Has96] W. Haslam. Mapping between EXPRESS models. Technical Report STEED/T6/02/1, Department of Computer Science, University of Manchester, May 1996.

[Haz94] M. Hazan. EXPRESS language modelling. MSc thesis, Department of Computer Science, University of Manchester, 1994.

[HN90] M.T. Harandi and J.Q. Ning. Knowledge-based program analysis. *IEEE Software*, 7(1):74–81, January 1990.

[Hor97] I.R. Horrocks. *Optimising Tableaux Decision Procedures for Description Logics.* PhD thesis, Department of Computer Science, University of Manchester, 1997. `http://www.cs.man.ac.uk/~horrocks/Publications/download/1997/phd-2sss.ps.gz` (modified 17th November 1999).

[HP96] U. Hohenstein and V. Plesser. Semantic enrichment: A first step to provide database interoperability. In *Workshop Föderierte Datenbanken, Magdeburg*, pages 3–17, 1996.

[Inm96] W.H. Inmon. The data warehouse and data mining. *Communications of the ACM*, 39(11):49–50, November 1996.

[ISO94] ISO, Switzerland. *Industrial Automation Systems and Integration – Product Data Representation and Exchange – Part 11: Description Methods: The EXPRESS Language Reference Manual (ISO TC184/SC4/WG5)*, 1994. ISO Standard, reference number ISO 10303-11.

[JBB+97] R.J. Bayardo Jr., W. Bohrer, R. Brice, A. Cichocki, J. Fowler, A. Helal, V. Kashyap, T.Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. Infosleuth: Agent-based semantic integration of information in open and dynamic environments. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 1997), Tucson, Arizona, United States of America, 13th–15th May 1997*, pages 195–206. ACM Press, 1997.

[Kas96] F.S. Kasadha. An approach to the application of information models. MSc thesis, Department of Computer Science, University of Manchester, 1996.

[KFWW01] H. Kahn, N. Filer, A. Williams, and N. Whitaker. A generic framework for transforming EXPRESS information models. *Computer-Aided Design*, 33(7):501–510, June 2001.

[Kol93] J. Kolodner. *Case-Based Reasoning.* Morgan Kaufmann, San Mateo, California, United States of America, 1993. ISBN 1-55860-237-2.

[KS96] P. Katranuschkov and R.J. Scherer. Schema mapping and object matching: A STEP-based approach to engineering data management in open integration environments. In *Proceedings of the CIB-W78 Workshop on Construction of the Information Highway (Bled'96), Bled, Slovenia, 10th-12th June 1996*, pages 335–346, 1996. `http://itc.fgg.uni-lj.si/bled96/papers/katra.pdf` (accessed 4th August 2001).

[Lab96] The Laboratory for Industrial Information Infrastructure of the New York State Center for Advanced Technology in Robotics, Automation, and Manufacturing, Rensselaer Polytechnic Institute, Troy, New York, United States of America. *The EXPRESS-X Reference Manual*, 1996. ISO TC184/SC4/WG11/N002.

[Lau96] R.Y.W. Lau. *EDIF: Electronic Design Interchange Format Version 4 0 0 Level 0 Information Model.* Electronic Industries Association, EDIF Steering Committee, 1996.

[LFS97] A.Y. Levy, R.E. Fikes, and Y. Sagiv. Speeding up inferences using relevance reasoning: A formalism and algorithms. *Artificial Intelligence*, 97(1–2):83–136, December 1997.

[LGM96] W.J. Labio and H. García-Molina. Efficient snapshot differential algorithms for data warehousing. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB 1996), Mumbai (Bombay), India, 3rd–6th September 1996*, pages 63–74, 1996.

[LK93] R.Y.W. Lau and H.J. Kahn. Information modelling of EDIF. In *Proceedings of the 30th Design Automation Conference, Dallas, Texas, United States of America, 14th–18th June 1993*, pages 278–283, 1993.

[LMB92] J.R. Levine, T. Mason, and D. Brown. *Lex & yacc*. O'Reilly & Associates, Sebastopol, California, United States of America, 2nd edition, 1992. ISBN 1-56592-000-7.

[Lou93] K.C. Louden. *Programming Languages: Principles and Practice.* PWS Publishing Company, Boston, Massachusetts, United States of America, 1993. ISBN 0-534-93277-0.

[LS86] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3):41–49, May 1986.

[LTV95] G. Lindén, H. Tirri, and A.I. Verkamo. ALCHEMIST: A general purpose transformation generator. Technical Report C-1995-43, Department of Computer Science, University of Helsinki, September 1995. `ftp://ftp.cs.helsinki.fi/pub/Reports/by_Project/Vital/ALCHEMIST:_A_General_Purpose_Transformation_Generator.ps.gz` (modified 3rd December 1996).

[LTV96] G. Lindén, H. Tirri, and A.I. Verkamo. ALCHEMIST: A general purpose transformation generator. *Software–Practice and Experience*, 26(6):653–675, June 1996.

[MBO93] S.A. Mamrak, J. Barnes, and C.S. O'Connell. Benefits of automating data translation. *IEEE Software*, 10:82–88, July 1993.

[MCP93] C. Myers, C. Clack, and E. Poon. *Programming with Standard ML*. Prentice Hall, New York, New York, United States of America, 1993. ISBN 0-13-722075-8.

[Mey88] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, New York, United States of America, 1988. ISBN 0-13-629031-0.

[Mey92] B. Meyer. *Eiffel: The Language*. Prentice Hall, New York, New York, United States of America, 1992. ISBN 0-13-247925-7.

[Mil98] E. Miller. An introduction to the Resource Description Framework. *D-Lib Magazine*, May 1998. `http://www.dlib.org/dlib/may98/miller/05miller.html` (modified 24th August 1998).

[Mir93] S. Mir. *Heuristic Reasoning for an Automatic Commonsense Understanding of Logic Electronic Design Specifications*. PhD thesis, Department of Computer Science, University of Manchester, 1993.

[MK97] C. Musciano and B. Kennedy. *HTML: The Definitive Guide*. O'Reilly & Associates, Cambridge, Massachusetts, United States of America, 2nd edition, 1997. ISBN 1-56592-235-2.

[MOTU93] H.A. Müller, M.A. Orgun, S.R. Tilley, and J.S. Uhl. A reverse engineering approach to subsystem structure identification. *Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[MS98] E. Melis and A. Schairer. Similarities and reuse of proofs in formal software verification. In *Advances in Case-Based Reasoning: Proceedings of the 4th European Workshop (EWCBR-98), Dublin, Ireland, 23rd–25th September 1998*, volume 1488 of *Lecture Notes in Computer Science*, pages 76–87, Berlin, Germany, 1998. Springer-Verlag. ISBN 3-540-64990-5.

[MSSM95] P. Maguire, R. Szegfue, V. Shankararaman, and L. Morss. Application of case-based reasoning (CBR) to software reuse. In I.D. Watson, editor, *Progress in Case-Based Reasoning: Proceedings of the 1st United Kingdom Workshop (EWCBR-95), Salford, England, January 1995*, volume 1020 of *Lecture Notes in Computer Science*, pages 166–174, Berlin, Germany, 1995. Springer-Verlag. ISBN 1-55860-462-6.

[NEK94] J.Q. Ning, A. Engberts, and W. Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5):50–57, May 1994.

[Pra94] I. Pratt. *Artificial Intelligence.* Macmillan, London, England, 1994. ISBN 0-333-59755-9.

[Pro] A. Protasov. *Schema Comparison 3.2 User's Guide.* COMTEK International.

[PSU98] L. Palopoli, D. Saccà, and D. Ursino. Semi-automatic, semantic discovery of properties from database schemes. In *Proceedings of the 1998 International Database Engineering and Applications Symposium (IDEAS 1998), Cardiff, Wales, 8th–10th July 1998*, pages 244–253. IEEE Computer Society, 1998. ISBN 0-8186-8307-4.

[Pun91] P.K.C. Pun. *Knowledge-Based Applications = Knowledge-Base + Mappings + Applications.* PhD thesis, Department of Computer Science, University of Manchester, 1991.

[Qui94] A. Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, May 1994.

[Rat] Rational Software Corporation. *Unified Modeling Language Resource Center*. `http://www.rational.com/uml/` (accessed 9th October 2002).

[RK91] E. Rich and K. Knight. *Artificial Intelligence*. McGraw-Hill, New York, New York, United States of America, 2nd edition, 1991. ISBN 0-07-100894-2.

[RW88] C. Rich and R.C. Waters. The Programmer's Apprentice: A research overview. *IEEE Computer*, 21(11):10–25, November 1988.

[RW90] C. Rich and L.M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, January 1990.

[Sam90] P. Samuelson. Reverse-engineering someone else's software: Is it legal? *IEEE Software*, 7(1):90–96, January 1990.

[SC92] B. Smyth and P. Cunningham. A hierarchical case-based reasoning system for software design. In B. Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 92), Vienna, Austria, 3rd–7th August 1992*, pages 587–589, Chichester, England, 1992. John Wiley & Sons.

[SK98] B. Smyth and M.T. Keane. Adaptation-guided retrieval: Questioning the similarity assumption in reasoning. *Artificial Intelligence*, 102(2):249–293, 31st July 1998.

[Spi94] J.M. Spivey. *The Z Notation*. Prentice Hall, New York, New York, United States of America, 1994. ISBN 0-13-983768-X.

[Squ95] C. Squire. Data extraction and transformation for the data warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 1995), San Jose, California, United States of America, 22nd–25th May 1995*, pages 446–447. ACM Press, 1995.

[SS99] P. Spiby and D. Sanderson. *Industrial Automation Systems and Integration – Product Data Representation and Exchange – Part 11: Description Methods: The EXPRESS Language Reference Manual, Second Edition (ISO TC184/SC4/WG11-N081)*, 1999. `http://www.nist.gov/sc4/wg_qc/wg11/n081/n81.pdf` (modified 8th October 1999).

[Sta94] G. Staub. *EXPRESS-C: A Language for Conceptual Information Modelling, version 1.0.0.* University of Karlsruhe, May 1994.

[STEa] STEPWISE Project. *EXPRESS Models: Gerber.* `http://www.stepwise.org/private/gerber/` (modified 22nd July 1999).

[STEb] STEPWISE Project. *EXPRESS Models: Mr Jones's Garden.* `http://www.stepwise.org/private/mr_jones/` (modified 22nd July 1999).

[STEc] STEPWISE Project. *EXPRESS Models: VHDL.* `http://www.stepwise.org/private/vhdl/` (modified 22nd July 1999).

[Str91] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, United States of America, 2nd edition, 1991. ISBN 0-201-53992-6.

[Sun] Sun Microsystems, Inc.. *Java Compiler Compiler (JavaCC) – The Java Parser Generator.*

[SW94] D.A. Schenck and P.R. Wilson. *Information Modeling: The EXPRESS Way.* Oxford University Press, Oxford, England, 1994. ISBN 0195087143.

[Tal98] C. Talcott. Reasoning about functions with effects. In A.D. Gordon and A.M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 347–390. Cambridge University Press, Cambridge, England, 1998. ISBN 0-521-63168-8.

[Teg97] A.C. Teganeanu. A system for configurable, information model based implementations. MPhil thesis, Department of Computer Science, University of Manchester, 1997.

[TL98] H.B.K. Tan and T.W. Ling. Correct program slicing of database operations. *IEEE Software*, 15(2):105–112, March/April 1998.

[Tür99] C. Türker. *Semantic Integrity Constraints in Federated Database Schemata.* infix, Sankt Augustin, Germany, 1999. Dr-Ing thesis, die Fakultät für Informatik, der Otto-von-Guericke-Universität Magdeburg (Computer Science Department, University of Magdeburg). ISBN 3-89601-463-3.

[TW96] J. Tiuryn and M. Wand. Untyped lambda-calculus with input-output. In H. Kirchner, editor, *Trees in Algebra and Programming: CAAP'96, Proceedings of the 21st International Colloquium, Linköping, Sweden, 22nd–24th April 1996*, volume 1059 of *Lecture Notes in Computer Science*, pages 317–329, Berlin, Germany, 1996. Springer-Verlag. ISBN 3-540-61064-2.

[TWP98] B. Tessem, R.A. Whitehurst, and C.L. Powell. Retrieval of Java classes for case-based reuse. In B. Smyth and P. Cunningham, editors, *Advances in Case-Based Reasoning: Proceedings of the 4th European Workshop (EWCBR-98), Dublin, Ireland, 23rd–25th September 1998*, volume 1488 of *Lecture Notes in Computer Science*, pages 148–159, Berlin, Germany, 1998. Springer-Verlag. ISBN 3-540-64990-5.

[VHG] VHG. *Virtual HyperGlossary (VHG).* `http://www.vhg.org.uk/` (modified 16th June 2000).

[vMV95] A. von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, August 1995.

[vV93] J.C. van Vliet. *Software Engineering: Principles and Perspective.* John Wiley & Sons, Chichester, England, 1993. ISBN 0-471-93611-1.

[Wil86] R. Wilensky. *Common LISPcraft*. Norton Press, New York, New York, United States of America, 1986. ISBN 0-393-95544-3.

[Wil95] P. Wilson. *Industrial Automation Systems and Integration – Product Data Representation and Exchange – Part 12: Description Methods: The EXPRESS-I Language Reference Manual (ISO TC184/SC4 N440)*, 1995. `http://www.nist.gov/sc4/step/parts/part012/tech_rpt/part12.pdf` (modified 20th May 1998).

[Win92] P.H. Winston. *Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, United States of America, 3rd edition, 1992. ISBN 0-201-53377-4.

[WK00] C. Walther and T. Kolbe. Proving theorems by reuse. *Artificial Intelligence*, 116(1–2):17–66, January 2000.

[Wol91] S. Wolfram. *Mathematica®: A System for Doing Mathematics by Computer*. Addison-Wesley, Redwood City, California, United States of America, 2nd edition, 1991. ISBN 0-201-51507-5.

[WSC+97] J.T.-L. Wang, D. Shasha, G.J.S. Chang, L. Relihan, K. Zhang, and G. Patel. Structural matching and discovery in document databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 1997), Tucson, Arizona, United States of America, 13th–15th May 1997*, pages 560–563. ACM Press, 1997.

[WWK94] N.A. Whitaker, A.R. Williams, and H.J. Kahn. EDIF/CFI interoperability. In *Proceedings of EDIF World '94, Las Vegas, Nevada, United States of America, October 1994*, pages 34–39, 1994.

[WZS95] J.T.-L. Wang, K. Zhang, and D. Shasha. Pattern matching and pattern discovery in scientific, program, and document databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 1995), San*

Jose, California, United States of America, 22nd–25th May 1995, page 487. ACM Press, 1995.

[xml97] Extensible markup language (XML). Technical Report PR-xml-971208, World Wide Web Consortium (W3C®), December 1997. `http://www.w3.org/TR/PR-xml-971208` (modified 8th September 2000).

[YHR92] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Transactions on Software Engineering and Methodology*, 1(3):310–354, July 1992.

[ZW97] A. Moormann Zaremski and J.M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.

Appendix A

Higher-Level Constraints for EXPRESS

This appendix presents a set of Higher-Level Constraints (HLCs) for EXPRESS, which characterise the abstract semantics of various pairs of models. Each class of HLCs has one or more *extraction cases* for extracting it from EXPRESS code (Appendix E, Section E.1).

This appendix also serves to expose more of the EXPRESS language than Chapter 1 does, although for a proper overview of EXPRESS (without HLCs), Section 2.2 of [Kas96] is recommended.

This appendix derives heavily from Appendix B of [Bro99], but is more up to date.

A.1 Choice-Set Constraints

The two EXPRESS models in Figure A.1 illustrate two ways of expressing the constraint that a person must have exactly one of two genders (see Section 1.2). The Higher-Level-Constraint-extracting Comparative Code-Understanding System (HLC CCUS) sees that they are equivalent by extracting a `choice_set` HLC from both models, comparing these HLCs and inferring them to correspond.

This constraint can also be expressed using a SELECT type or as constrained Boolean flags (Figure A.2):

- The SELECT type means that a person's value of gender must be either an instance of the entity `male` or an instance of the entity `female`.

```
SCHEMA test;
  ENTITY person;
    name : STRING;
    age : INTEGER;
    gender : gender_type;
  END_ENTITY;

  TYPE gender_type = ENUMERATION OF (man, woman);
  END_TYPE;
END_SCHEMA;
```

```
SCHEMA test;
  ENTITY person ABSTRACT SUPERTYPE OF (ONEOF (man, woman));
    name : STRING;
    age : INTEGER;
  END_ENTITY;

  ENTITY man SUBTYPE OF (person);
  END_ENTITY;

  ENTITY woman SUBTYPE OF (person);
  END_ENTITY;
END_SCHEMA;
```

```
INSTANCE choice_set#7;
  extracted_from = {defined_type#36};
  corresponds_to = {correspondence#5};
  choice_set = {enumeration_symbol#42, enumeration_symbol#43};
  is_optional = false;
  constrained_entity = entity_type#188;
  attribute = explicit_attribute#292;
END_INSTANCE;
```

```
INSTANCE choice_set#8;
  extracted_from = {oneof_supertype_expression#26};
  corresponds_to = {correspondence#5};
  choice_set = {entity_type#190, entity_type#191};
  is_optional = false;
  constrained_entity = entity_type#189;
END_INSTANCE;
```

Figure A.1: A person must have exactly one of two genders.

```
SCHEMA test;
  ENTITY person;
    name : STRING;
    gender : gender_type;
  END_ENTITY;

  TYPE gender_type = SELECT (male, female);
  END_TYPE;

  ENTITY male;
  END_ENTITY;

  ENTITY female;
  END_ENTITY;
END_SCHEMA;
```

```
SCHEMA test;
  ENTITY person;
    name : STRING;
    is_male : BOOLEAN;
    is_female : BOOLEAN;
  WHERE
    is_male XOR is_female;
  END_ENTITY;
END_SCHEMA;
```

Figure A.2: Two more ways of expressing that a person must have exactly one of two genders.

- The domain rule `is_male XOR is_female` constrains a person to be either male or female, but not both.

Extraction cases have been implemented for extracting `choice_set` from an ENUMERATION type (Appendix E, Section E.1.1), from a SELECT type (Section E.1.2), and from a supertype (Sections E.1.3 and E.1.4), but not currently for constrained Boolean flags.

A.2 Bound and Range Constraints

The models in Figure A.3 show two ways of expressing a range constraint:

- as a conjunction of two comparators such that one is a lower bound and the other is an upper bound: the HLC CCUS extracts a `bound` HLC from

```
SCHEMA test;
  ENTITY person;
    name : STRING;
    age : INTEGER;
  WHERE
    (age >= 0) AND (age < 130);
  END_ENTITY;
END_SCHEMA;
```

```
SCHEMA test;
  ENTITY person;
    name : STRING;
    age : INTEGER;
  WHERE
    {0 <= age < 130};
  END_ENTITY;
END_SCHEMA;
```

```
INSTANCE range#4;
  extracted_from = {bound#8, bound#9};
  covers = conjunction#5;
  corresponds_to = {correspondence#6};
  lower_bound = bound#8;
  upper_bound = bound#9;
  variable = explicit_attribute#291;
  discrete = discrete_or_continuous(discrete);
END_INSTANCE;

INSTANCE bound#8;
  extracted_from = {comparator#8};
  covers = comparator#8;
  corresponds_to = {correspondence#2};
  hlcs_extracted = {range#4}; (* set indirectly *)
  kind = lower;
  inclusive = inclusive_or_exclusive(inclusive);
  discrete = discrete_or_continuous(discrete);
  variable = explicit_attribute#291;
  limit = integer_literal#97;
END_INSTANCE;

INSTANCE bound#9;
  extracted_from = {comparator#9};
  covers = comparator#9;
  corresponds_to = {correspondence#4};
  hlcs_extracted = {range#4}; (* set indirectly *)
  kind = upper;
  inclusive = inclusive_or_exclusive(exclusive);
  discrete = discrete_or_continuous(discrete);
  variable = explicit_attribute#291;
  limit = integer_literal#98;
END_INSTANCE;
```

```
INSTANCE range#5;
  extracted_from = {interval#2};
  covers = interval#2;
  corresponds_to = {correspondence#6};
  lower_bound = bound#10;
  upper_bound = bound#11;
  variable = explicit_attribute#293;
  discrete = discrete_or_continuous(discrete);
END_INSTANCE;

INSTANCE bound#10;
  corresponds_to = {correspondence#2};
  kind = lower;
  inclusive = inclusive_or_exclusive(inclusive);
  discrete = discrete_or_continuous(discrete);
  variable = explicit_attribute#293;
  limit = integer_literal#99;
END_INSTANCE;

INSTANCE bound#11;
  corresponds_to = {correspondence#4};
  kind = upper;
  inclusive = inclusive_or_exclusive(exclusive);
  discrete = discrete_or_continuous(discrete);
  variable = explicit_attribute#293;
  limit = integer_literal#100;
END_INSTANCE;
```

Figure A.3: Two ways of expressing a range constraint.

each of the two comparators, and then extracts a `range` HLC from the conjunction of two `bound`s (Sections E.1.5, E.1.6 and E.1.7);

- as an interval expression: the HLC CCUS extracts a `range` and two `bounds` from the interval expression using a single extraction case (Section E.1.8).

A further possibility to consider would be that a *numeric range constraint* such as {`3 <= SELF <= 6`} could be equivalent to a *symbolic range constraint* such as `ENUMERATION OF (three, four, five, six)`.

A.3 Equations

Figure A.4 shows two ways that an equation can be represented in EXPRESS:

- as a domain rule, which is represented directly as an `equation`;
- as a *derived attribute*, i.e. an attribute whose value is computed from other

`SCHEMA test;`
`  ENTITY rectangle;`
`    width : REAL;`
`    height : REAL;`
`    area : REAL;`
`  WHERE`
`    area = width * height;`
`  END_ENTITY;`
`END_SCHEMA;`
	`SCHEMA test;`
	`  ENTITY rectangle;`
	`    width : REAL;`
	`    height : REAL;`
	`  DERIVE`
	`    area : REAL := width * height;`
	`  END_ENTITY;`
	`END_SCHEMA;`

Figure A.4: Two ways of expressing equations.

attributes using a specified formula: an `equation` can be extracted from a derived attribute (Section E.1.9).

From `equations`, the CCUS can extract:

- `adder` HLCs which capture the semantic equivalence of $x+y=z$, $x=z-y$ and $y=z-x$ by representing them each as an adder with `addends` $\{x, y\}$ and `sum` z (Sections E.1.10 and E.1.11);
- `multiplier` HLCs which capture the semantic equivalence of $x*y=z$, $x=z/y$ and $y=z/x$ by representing them each as a multiplier with `multiplicands` $\{x, y\}$ and `product` z (Sections E.1.12 and E.1.13).

A.4 Existence Constraints

Figure A.5 shows two ways of expressing that an attribute must have a value other than *indeterminate* (null):

- the conventional way to express this is to make the attribute mandatory: a `non_null` HLC can be extracted from a mandatory attribute (Section E.1.15);
- an unorthodox but semantically equivalent way to express this is to use the built-in function `EXISTS`, which returns true if a variable has a non-null value, false otherwise: a `non_null` HLC can be extracted from a call to `EXISTS` (Section E.1.16).

The usefulness of the `EXISTS` function is that it can be used to express more complex existence constraints, such as `EXISTS(age) OR EXISTS(date_of_birth)`; see Section 1.4.2.1 for another example.

```
SCHEMA test;
  ENTITY person;
    name : STRING;
    age : OPTIONAL INTEGER;
  WHERE
    EXISTS(age);
  END_ENTITY;
END_SCHEMA;
```

```
SCHEMA test;
  ENTITY person;
    name : STRING;
    age : INTEGER;
  END_ENTITY;
END_SCHEMA;
```

```
INSTANCE non_null#5;
  extracted_from = {function_call#4};
  covers = function_call#4;
  corresponds_to = {correspondence#2};
  attribute = explicit_attribute#291;
END_INSTANCE;
```

```
INSTANCE non_null#4;
  extracted_from = {explicit_attribute#293};
  corresponds_to = {correspondence#2};
  attribute = explicit_attribute#293;
END_INSTANCE;
```

Figure A.5: Two ways of expressing that `age` must have a non-null value.

```
SCHEMA test;
  ENTITY person;
    middle_names : LIST [0 : ?] OF STRING;
  END_ENTITY;
END_SCHEMA;
```

```
SCHEMA test;
  ENTITY person;
    middle_names : OPTIONAL LIST [1 : ?] OF STRING;
  END_ENTITY;
END_SCHEMA;
```

```
INSTANCE optional_xor_empty_aggregation#3;
  extracted_from = {explicit_attribute#290};
  corresponds_to = {correspondence#3};
  attribute = explicit_attribute#290;
END_INSTANCE;
```

```
INSTANCE optional_xor_empty_aggregation#4;
  extracted_from = {explicit_attribute#291};
  corresponds_to = {correspondence#3};
  attribute = explicit_attribute#291;
END_INSTANCE;
```

Figure A.6: Two ways of modelling zero or more items.

Another existence constraint is that an aggregation can have zero-or-more elements (Figure A.6), which can be expressed either as an optional aggregation of one or more elements, which can be null, or as a mandatory aggregation of zero or more elements, which can be empty. Their semantic equivalence is captured by the `optional_xor_empty` HLC (Sections E.1.17 and E.1.18).

Note that the bounds of aggregation-types in EXPRESS are specified in the form `[l : u]`, where `l` is the lower bound (minimum number of elements) and `u` is the upper bound (maximum number of elements – `?` denotes an indeterminate upper bound).

A.5 Inter-Entity Relationships

The CUS can recognise one-to-one, one-to-many, many-to-one and many-to-many relationships between entities (Figure A.7). It does so by extracting the HLCs `one_to_one`, `one_to_many` (for both one-to-many and many-to-one relationships) and `many_to_many` (Sections E.1.19 to E.1.22).

Figure A.7 contains *inverse attributes*, which are another kind of existence

```
SCHEMA one_to_one;
  ENTITY student;
    name : STRING;
    supervisor : staff;
  END_ENTITY;

  ENTITY staff;
    name : STRING;
  INVERSE
    supervisee : student FOR supervisor;
  END_ENTITY;
END_SCHEMA;
```

```
SCHEMA one_to_many;
  ENTITY student;
    name : STRING;
    supervisor : staff;
  END_ENTITY;

  ENTITY staff;
    name : STRING;
  INVERSE
    supervisees : SET [0 : ?] OF student FOR supervisor;
  END_ENTITY;
END_SCHEMA;
```

```
SCHEMA many_to_one;
  ENTITY student;
    name : STRING;
    supervisors : SET [0 : ?] OF staff;
  END_ENTITY;

  ENTITY staff;
    name : STRING;
  INVERSE
    supervisee : student FOR supervisor;
  END_ENTITY;
END_SCHEMA;
```

```
SCHEMA many_to_many;
  ENTITY student;
    name : STRING;
    supervisors : SET [0 : ?] OF staff;
  END_ENTITY;

  ENTITY staff;
    name : STRING;
  INVERSE
    supervisees : SET [0 : ?] OF student FOR supervisor;
  END_ENTITY;
END_SCHEMA;
```

Figure A.7: Four kinds of inter-entity relationship.

constraint. The one in `one_to_one`, for example (`supervisee`), means that an instance of `staff` cannot exist without an instance of `student` whose value for the attribute `supervisor` is that instance of `staff`.

Inverse attributes were not always present in the EXPRESS language. Furthermore, some real models (such as the Electronic Design Interchange Format information model [Lau96]) choose to express this constraint as a domain rule instead. Figure A.8 shows how an inverse attribute can be simulated by a domain rule:

- The built-in function USEDIN(i, a) returns the set of instances that have the instance i as a value for the attribute a, thus the one in Figure A.8 returns the set of students for whom a given instance of `staff` is the supervisor.

- The built-in function `SIZEOF` returns the number of elements in an aggregation, thus the one in Figure A.8 returns the *number* of students for whom a given instance of `staff` is the supervisor.

A.6 Universal and Existential Quantification

Finally, the HLCs `for_all` (Section E.1.23) and `there_exists` (Section E.1.24) have been implemented to recognise universal and existential quantification (Section 5.2.2) in EXPRESS (for example, Figure A.9 encodes "all men are mortal" and "some men are married"):

```
SCHEMA test;
  ENTITY student;
    name : STRING;
    supervisor : staff;
  END_ENTITY;

  ENTITY staff;
    name : STRING;
  WHERE
    supervises_one_student:
      SIZEOF(USEDIN(SELF, 'test.student.supervisor')) = 1;
  END_ENTITY;
END_SCHEMA;
```

Figure A.8: Another way to write the constraint implied by an inverse attribute.

- `all_men_are_mortal` and `some_men_are_married` are *global rules* which apply to the entity `everything` – they are thus *inter-instance constraints* (global rules can apply to more than one entity, in which case they are *inter-entity constraints*). In the body of the rule, `everything` refers to the set of instances of `everything`.

- A *query expression*, of the form `QUERY (x <* s | p(x))`, returns the subset of a given set `s` such that each element `x` satisfies the predicate `p(x)`.

  – The first `QUERY` expression in Figure A.9 returns the set of things such that if one is a man then one is mortal. The set returned by the query has to be equal to the full set of instances for them *all* to satisfy the condition.

  – The second `QUERY` expression in Figure A.9 returns the set of things that are both man and married, and the cardinality of this set has to be greater than or equal to one for *some* elements to satisfy the condition.

A.7 Uniqueness Rules

Another kind of inter-instance constraint in EXPRESS is the uniqueness rule, which specifies that no two instances may have the same value for a specified

```
SCHEMA test;
   ENTITY everything;
      man : BOOLEAN;
      mortal : BOOLEAN;
      married : BOOLEAN;
   END_ENTITY;

   RULE all_men_are_mortal FOR (everything);
   WHERE
      QUERY (x <* everything | NOT x.man OR x.mortal) = everything;
   END_RULE;

   RULE some_men_are_married FOR (everything);
   WHERE
      SIZEOF (QUERY (x <* everything | x.man AND x.married)) >= 1;
   END_RULE;
END_SCHEMA;
```

Figure A.9: Universal and existential quantification in EXPRESS.

```
SCHEMA test;
  ENTITY person;
    forename : STRING;
    surname : STRING;
  UNIQUE
    forename;
  END_ENTITY;
END_SCHEMA;
```

```
SCHEMA test;
  ENTITY person;
    forename : STRING;
    surname : STRING;
  UNIQUE
    forename, surname;
  END_ENTITY;
END_SCHEMA;
```

Figure A.10: Uniqueness rules.

attribute, or the same set of values for a set of attributes. For example, the first model in Figure A.10 constrains a person's forename to be unique, while the second model constrains a person's forename and surname to be jointly unique. Thus the second model, but not the first, would accept both "Jelena Dokić" and "Jelena Pandzić" – so the first is semantically stronger than the second (Section E.3.9).

It is also possible to simulate `UNIQUE` by a global rule.

Appendix B

The Code-Understanding System

This appendix explains how the Code-Understanding System (CUS) works, in particular how it matches code-fragments to extraction cases, which tell it what higher-level constraints (HLCs) to extract. The CUS was first presented in [Bro99].

The case-matching engine applies not just to extraction cases but to the other kinds of cases in the Comparative Code-Understanding System (CCUS), including subexpression cases (Section 4.3) and semantic-equivalence/strength cases (Section 5.3).

This appendix is best read after Chapter 3, which explains the frame-representation of EXPRESS models and of HLCs (Section 3.2), and also the CCUS frame-comparison algorithm(Section 3.3), which is similar to the CUS case-matching algorithm presented here.

B.1 Extraction Cases and Case Matching

To recap from Section 3.1, extraction cases are used to extract HLCs from an EXPRESS model. Each extraction case consists of two parts:

- *problem part*: a template code-fragment which a code-fragment in the input model must match for the case to be applicable;
- *solution part*: an HLC (or more than one) which, if an input code-fragment matches the problem part, the CUS extracts from that code-fragment, i.e. it reinstantiates the HLC for the input code, substituting values for variables that were matched in the problem part.

The cases are represented in *FIL-C*, an extension of the frame-instance language (FIL) presented in Section 3.2.1. Figure B.1 shows an extraction case for extracting a lower bound from a comparator such as `age >= 0`. The complete set of extraction cases is in Appendix E, Section E.1.

FIL-C was first presented in Section 6.3 of [Bro99] (although it was just called FIL then), from which most of the material in this section derives.

Notes on the FIL-C case-representation:

- Each case consists of a top-level *case-frame* (of frame-class `extraction_case` in the case of extraction cases such as the one in Figure B.1), which includes the slots:
  - `problem`: the top-level frame of the problem part of the case (the problem part consists of all frames in the transitive closure of frame-references from this slot);
  - `solution`: the top-level frame of the solution part of the case; it can also be a set of frames – for cases that extract more than one HLC from the same code-fragment (e.g. extracting a `non_null` and a `type_hlc` from a mandatory attribute) – although HLCs can also be nested under a single top-level solution frame (e.g. extracting a `range` and two `bounds` from an interval expression such as {`0 <= age < 130`}).
- Every frame in the problem part of the case must be matched by a corresponding frame in the input, and every slot-filler specified in the frame from the case must be matched by the corresponding slot-filler in the input frame.
- If a slot is unfilled in the problem part of a case, that slot is ignored (yes, it is valid to omit slots which are mandatory attributes in the frame-class model). If a slot in the input **must** be unfilled, this is explicitly specified in the case ("`slot = null`").
- FIL-C extends FIL with *variables* (`?inclusive` and `?discrete` in Figure B.1), to which slot-fillers in the input code are matched using *unification* in the same way as pattern-directed inference systems (Section 4.1 of [FdK93]):

```
(*************************************)
(* bound-lower.filc 2000:11:09 15:45 *)
(*************************************)
(* COMPARATOR                        *)
(*      greater = EXPRESSION#1       *)
(*      lesser = EXPRESSION#2        *)
(*      inclusive = ?inclusive       *)
(*      discrete = ?discrete         *)
(*                                   *)
(* EXPRESSION#1                      *)
(*      is_variable = true           *)
(*                                   *)
(* EXPRESSION#1                      *)
(*      is_variable = false          *)
(*************************************)
(* BOUND                             *)
(*      kind = lower                 *)
(*      variable = EXPRESSION#1      *)
(*      limit = EXPRESSION#2         *)
(*      inclusive = ?inclusive       *)
(*      discrete = ?discrete         *)
(*      extracted_from = COMPARATOR  *)
(*      covers = COMPARATOR          *)
(*************************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting a lower bound from a comparator";
  problem = comparator#1;
  solution = bound#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE comparator#1;
  greater = expression#1;
  lesser = expression#2;
  inclusive = ?inclusive;
  discrete = ?discrete;
END_INSTANCE;

INSTANCE expression#1;
  is_variable = true;
END_INSTANCE;

INSTANCE expression#2;
  is_variable = false;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE bound#1;
  kind = lower;
  variable = expression#1;
  limit = expression#2;
  inclusive = ?inclusive;
  discrete = ?discrete;
  extracted_from = {comparator#1};
  covers = comparator#1;
END_INSTANCE;
```

Figure B.1: Extraction case for extracting a lower bound from a comparator.

- when a slot-filler (in the input) is matched to an *unbound* variable in the problem part, the variable is bound to that filler (the match always succeeds);
- when a slot-filler is matched to a *bound* variable in the problem part, it is matched to the value to which the variable is bound (thus the match can succeed or fail);
- when instantiating the solution part for the input, variables in the solution part are replaced with the values to which they are bound.

For example, consider extracting from `age >= 0` a lower inclusive bound of 0 on `age`, using the extraction case in Figure B.1. The input code-fragment `age >= 0` is represented by the following frame:

```
INSTANCE comparator#8;
  description = "age >= 0";
  tipe = simple_type(logical_type);
  greater = explicit_attribute#291;
  lesser = integer_literal#97;
  inclusive = inclusive_or_exclusive(inclusive);
  discrete = discrete_or_continuous(discrete);
  which_way_round = way_round(greater_first);
END_INSTANCE;
```

When the CUS submits `comparator#8` as an extraction prompt (Section B.2), it retrieves the extraction case in Figure B.1 because that case is indexed by the frame-class of the top-level frame in the problem part, `comparator`. It is not the only such case retrieved, but since the case library is written without overlap, the code-fragment fails to match any other case – it only succeeds in matching this one.

First `comparator#8` is matched to the top-level frame in the problem part, `comparator#1`.[1] Only the slots specified in `comparator#1` (i.e. `greater`, `lesser`, `inclusive` and `discrete`) are matched (the other slots filled in `comparator#8` are ignored):

- In matching the slot `greater`, `explicit_attribute#291` is matched to `expression#1`;
- In matching the slot `lesser`, `integer_literal#97` is matched to `expression#2`;

[1]Note that `comparator#8` has an *internal instance-number* whereas `comparator#1` has an *external instance-number* (cf. Section 3.2.1).

- In matching the slot `inclusive`, `?inclusive` is bound to `inclusive_or_exclusive(inclusive)`;
- In matching the slot `discrete`, `?discrete` is bound to `discrete_or_continuous(discrete)`;

The matching of `explicit_attribute#291` to `expression#1` is now considered in detail:

```
INSTANCE explicit_attribute#291;
  tipe = simple_type(integer_type);
  is_variable = true;
  name = "age";
  is_optional = true;
  the_entity = entity_type#188;
END_INSTANCE;
```

The frame-class of `explicit_attribute#291` matches the frame-class of `expression#1` because `explicit_attribute` is a subclass of `expression`. The only slot specified in `expression#1` is `is_variable`, which is trivially matched by `explicit_attribute#291` (an attribute is a variable expression).

Similarly, `integer_literal#97` matches `expression#2` (an integer literal is a constant expression).

```
INSTANCE integer_literal#97;
  tipe = simple_type(integer_type);
  is_variable = false;
  literal_value = 0;
END_INSTANCE;
```

When the CUS successfully matches a frame from the input code to a frame in the problem part of a case, it records a *frame-association* between them.

Since all frames in the problem part of the case are matched, the frames in the solution part (in this case, just one frame) can be extracted. This entails reinstantiating each solution frame, making substitutions according to the variable-bindings and frame-associations that were recorded when matching the problem part. Thus `bound#1` would be reinstantiated as follows:

```
INSTANCE bound#8;
  kind = lower;
  variable = explicit_attribute#291;
  limit = integer_literal#97;
  inclusive = inclusive_or_exclusive(inclusive);
  discrete = discrete_or_continuous(discrete);
  extracted_from = {comparator#8};
  covers = comparator#8;
END_INSTANCE;
```

The case-matching algorithm is a variation of the CCUS frame-comparison algorithm presented in Section 3.3, with the following differences:

- Whereas the CCUS frame-comparison algorithm compares two codes, which are 'equal in status', in the CUS case-matching algorithm the input code-fragment must match the problem part of a case (but not vice versa). Thus the CCUS is performing bidirectional comparison, whereas the CUS is performing unidirectional matching.

- If the CUS case-matching algorithm encounters a difference (i.e. if a slot-filler specified in the problem part of a case is not matched by the corresponding slot-filler in the input), then the match fails immediately. Whereas the CCUS frame-comparison algorithm is comparing two codes to find as many differences and correspondences as it can, and the only situation in which it fails a match is in plausible-correspondence mode, when a difference is encountered that cannot be resolved (i.e. semantic equivalence despite the difference cannot be shown).

- These differences are reflected in the aggregation-matching policies used by the CUS case-matching algorithm, as contrasted in Section 3.4 with those used by the CCUS frame-comparison algorithm.

- The CUS case-matching algorithm does not record, in the frame-networks, the correspondences it infers, though it *does* record the variable-bindings and frame-associations (needed for instantiating the solution part) in a separate data-structure.

B.2 The Case Library

The previous section explained how a code-fragment is matched to a given case. This section explains how an appropriate case, to which to match a given code-fragment, is retrieved from the *case library*, and how cases are added to the case library.

The case library is represented as a hash-table whose keys are frame-classes, and whose values are sets of cases. Each case is indexed by the frame-class of the top-level frame in its problem part, thus the case in Figure B.1 is indexed by `comparator`, i.e. when it is added to the case library, it is added to the set of cases whose hash-table key is `comparator`.

When `comparator#8` is submitted to the CUS as an extraction prompt (either internally by the CUS itself, or externally by the CCUS, as explained in Section 6.3.1), the CUS retrieves from the case library the set of cases whose top-level problem frames are of frame-class `comparator`. It matches `comparator#8` to each case in this set, extracting HLCs from a case it matches successfully.

What about when the prompt frame is a subclass of a case's top-level problem frame-class rather than the frame-class itself? For example, if there were subclasses of `comparator` (there are no such subclasses), then they too should match the case in Figure B.1. To achieve this, when a case is added to the case library, it is indexed not only by its top-level problem frame-class, but also by any subclasses (and their subclasses, and so on) of that frame-class. This means that submitting an instance of one of these subclasses as an extraction prompt will cause the case to be retrieved.

Cases are written by humans as FIL-C text files, and are added to the case library via the FIL-C parser, which adds each case to the case library whenever it recognises a frame whose frame-class is a subclass of `case_frame`. In fact there is a separate case library for each kind of case:

- *extraction cases* (of frame-class `extraction_case`);
- *navigation cases* (`navigation_case`): these are used by the CUS to walk inwards from a given extraction prompt, internally submitting further extraction prompts (e.g. attributes of an entity, subexpressions of an expression) – see Section 6.3.1;
- *semantic-equivalence/strength cases* (`semantic_case`): see Section 5.3;
- *subexpression cases* (`subexpression_case`): see Section 4.3;
- *pretty-print cases* (`pretty_print_case`): see Appendix E, Section E.5.

All the cases except the pretty-print cases are in Appendix E, and **all** the cases are on the Author's website [Bro].

Appendix C

Algorithmic Details of the CCUS

This appendix presents pseudocode algorithms for:

- the main comparison algorithm of the Comparative Code-Understanding System (CCUS) (Section C.1);
- its aggregation-matching policies (Section C.2);
- inferring correspondences (Section C.3);
- assessing semantic equivalence and semantic strength (Section C.4);
- the Higher-Level-Constraint-extracting CCUS (HLC CCUS) (Section C.5).

This appendix does not include details of the Code-Understanding System (CUS) (Appendix B; see Appendix D of [Bro99] for algorithmic details), nor of the underlying frame system (Section 3.2.1). The actual program code – in full – can be found on the Author's website [Bro].

C.1 The Basic CCUS Frame-Comparison Algorithm

The basic CCUS frame-comparison algorithm (Section 3.3) is embodied in the methods `matchFrames` (which compares two frames – Section C.1.1) and `matchFillers` (which compares two slot-fillers – Section C.1.2). The method `processDifference` (Section C.1.3) provides the hook to assessing semantic equivalence (Section C.4) and to the HLC CCUS (Section C.5).

The CCUS starts by calling `matchFrames` on the two top-level `model_frames`.

C.1.1 `matchFrames`: Algorithm to compare two frames

This method compares two frames, on the precondition that they correspond or plausibly correspond. It calls `matchFillers` (Section C.1.2), `processDifference` (Section C.1.3), `recordCorrespondence` (Section C.3.4) and `retractCorrespondence` (Section C.3.5).

This method is responsible for inferring abstract correspondences back at the code level when two HLCs are inferred to correspond (Section 6.3.3) – it records correspondence between their fillers of the `covers` slot.

```
CCUS_FMA::matchFrames (frame1 : Frame, frame2 : Frame,
                       mode : ENUMERATION OF (CORRESPONDENCE,
                                              PLAUSIBLE_CORRESPONDENCE,
                       subsForSelf : Object)
                      : boolean
{
  ///////////////////////////////////////////////
  // If the frames have been compared already, //
  // then do not repeat the effort.            //
  ///////////////////////////////////////////////
  if frame1 and frame2 correspond (see Section 4.1.2)
  then return true

  if frame1 and frame2 have already been compared but failed to correspond
  then return false

  recordCorrespondence(frame1, frame2, mode)

  (seCase : Case):= null

  //////////////////////////////////////////////////////////
  // Work out what value (if any) to substitute for SELF. //
  //////////////////////////////////////////////////////////
  substituteForSelf:= whichever of frame1."substitute_for_self"
                      and frame2."substitute_for_self" is non-null
                      (they cannot both be non-null),
                      or null if they are both null
  if subsForSelf is non-null
  then substituteForSelf:= subsForSelf

  ///////////////////////////
  // Compare slot-fillers. //
  ///////////////////////////
  for each filled slot in frame1 {
    if slot is marked --IGNORE
    then skip this slot

    if frame1.slot was set directly (cf. Section 3.2.3)
    then {
      call matchFillers(frame1.slot, frame2.slot,
                        slot, substituteForSelf)
      if matchFillers returned false
      then {
        seCase:= processDifference(frame1, frame2,
                                   slot, mode, seCase)
        if processDifference failed the match
        then {
          retractCorrespondence(frame1, frame2)
```

```
          return false
        }
      }
    }
  } // end for

  for each filled slot in frame2 that was not matched above {
    process slot as above
  }

  if frame1's "covers" slot is filled and frame2's "covers" slot is filled
  then recordCorrespondence(frame1."covers", frame2."covers", mode)
       if they have not already been recorded to correspond
}
```

C.1.2 `matchFillers`: Algorithm to compare two slot-fillers

This method compares two slot-fillers, or the underlying values of defined-type-instances, or elements of aggregations (vectors). It calls `sanctionCorrespondence` (Section C.3.1), `matchFrames` (Section C.1.1) and `matchVectors` (Section C.2.1).

```
CCUS_FMA::matchFillers (filler1 : Object, filler2 : Object,
                        slot : Slot, subsForSelf : Object)
                       : boolean
{
  ///////////////////////////////////////////
  // If one slot-filler is null           //
  // then they must both be null to match. //
  ///////////////////////////////////////////
  if filler1 is null or filler2 is null
  then return (filler1 is null and filler2 is null)

  if filler1 and filler2 are not type-compatible
  then return false

  /////////////////////////////////////////
  // Substitute for SELF if appropriate. //
  /////////////////////////////////////////
  if subsForSelf is non-null
  then {
    if filler1 is SELF but filler2 is not SELF
    then filler1:= subsForSelf
    if filler2 is SELF but filler1 is not SELF
    then filler2:= subsForSelf
  }

  if filler1 and filler2 are frames
  then {
    mode:= sanctionCorrespondence(filler1, filler2)
    if mode is CORRESPONDENCE or PLAUSIBLE_CORRESPONDENCE
    then return matchFrames(filler1, filler2, mode, subsForSelf)
  }

  if filler1 and filler2 are defined-type-instances
     (the same defined type for both - return false if not)
  then return matchFillers(underlying value of filler1,
                           underlying value of filler2,
                           null, subsForSelf)
```

```
  if filler1 and filler2 are vectors
  then return matchVectors(filler1, filler2, type of slot, subsForSelf)

  otherwise return equals(filler1, filler2)
  where equals is the equality relation for the type of filler1 and filler2
}
```

C.1.3 `processDifference`: Algorithm to process a difference

When the CCUS is matching in `PLAUSIBLE_CORRESPONDENCE` mode (i.e. using semantic equivalence as a correspondence heuristic), this method attempts to resolve a difference by using a semantic-equivalence/strength (SE/S) case (`useSemanticCase` – Section C.4.1). Failing that, it attempts to resolve the difference by extracting and comparing HLCs (`extractHLCs` – Section C.5.1) – which it does anyway whenever it encounters a difference in `CORRESPONDENCE` mode.

```
CCUS_FMA::processDifference (frame1 : Frame, frame2 : Frame, slot : Slot,
                             mode : ENUMERATION OF (CORRESPONDENCE,
                                                    PLAUSIBLE_CORRESPONDENCE),
                             seCase : Case)
                            : Case
{
  if mode == PLAUSIBLE_CORRESPONDENCE then {
    try {
      return useSemanticCase(frame1, frame2, slot, seCase)
    }
    catch (FailMatch exception) {
      extractHLCs(frame1, frame2, slot)
      if frame1.slot and frame2.slot are both frames and correspond to each other
      then return seCase
      else exit this method, throwing FailMatch exception
    }
  }
  if mode == CORRESPONDENCE then {
    extractHLCs(frame1, frame2, slot)
  }
  return seCase
}
```

C.2 Aggregation-Matching Policies

When the CCUS compares two aggregations (Section 3.4), it calls the `matchVectors` method (Section C.2.1), which calls `matchUnorderedVectors` (Section C.2.2) or `matchOrderedVectors` (Section C.2.3) as appropriate.

C.2.1 `matchVectors`: Algorithm to select the appropriate aggregation-matching policy

```
AggregationMatchingPolicies::matchVectors (vector1 : Vector, vector2 : Vector,
                                           type : Object, subsForSelf : Object)
                                          : boolean
{
  if vector1 and vector2 are both ordered vectors
  then return matchOrderedVectors(vector1, vector2, subsForSelf)
  else return matchUnorderedVectors(vector1, vector2, type, subsForSelf)
}
```

C.2.2 `matchUnorderedVectors`: Algorithm to compare two unordered aggregations

This method implements the unordered-aggregation-matching policy of Section 3.4.1. It calls `matchFillers` (Section C.1.2).

```
AggregationMatchingPolicies::matchUnorderedVectors (vector1 : Vector,
                                                    vector2 : Vector, type : Object,
                                                    subsForSelf : Object)
                                                   : boolean
{
  ///////////////////////////////////////////////////////
  // If the aggregation elements are frames with slot //
  // "name", then the match is to be cut              //
  // on finding the first pair of matching fillers.   //
  ///////////////////////////////////////////////////////
  (cutAfterFirstElementMatch : boolean):= false
  if type is a frame of frame-class "unordered_aggregation_type"
  then {
    (elementFrameClass : FrameClass)
    := look up the frame-class whose name is (type."element_type"."name")
    for each slot in elementFrameClass {
      if the slot's name is "name"
      then cutAfterFirstElementMatch:= true
    }
  }

  ////////////////////////////////////////////////////////
  // The main unordered aggregation-matching algorithm. //
  ////////////////////////////////////////////////////////

  (difference : boolean):= false // any element differences?

  ////////////////////////////////////
  // For each element in vector1... //
  ////////////////////////////////////
  if vector2 is empty
  then return true if vector1 is empty, otherwise false
  (vector2clone : Vector):= copy of vector2
  (i2 : integer):= 0 // index into vector2

  for each element (element1) in vector1 {
    (matched : boolean):= false // found a matching element?
    (i2start : integer):= i2 // the start element in vector2
    do {
      element2:= vector2[i2]
```

```
      if matchFillers(element1, element2, null, subsForSelf)
      then {
        matched:= true
        remove element2 from vector2clone
      }
      i2:= (i2 + 1) mod (size of vector2)
    } while i2<>i2start AND NOT (matched AND cutAfterFirstElementMatch)
    if NOT matched then difference:= true
  } // end for

  ///////////////////////////////////
  // For each element in vector2   //
  // that was not matched above... //
  ///////////////////////////////////
  if vector1 is empty
  then return true if vector2clone is empty, otherwise false
  (i1 : integer):= 0 // index into vector1

  for each element (element2) in vector2clone {
    (matched : boolean):= false // found a matching element?
    (i1start : integer):= i1 // the start element in vector1
    do {
      element1:= vector1[i1]
      if matchFillers(element1, element2, null, subsForSelf)
      then matched:= true
      i1:= (i1 + 1) mod (size of vector1)
    } while i1<>i1start AND NOT (matched AND cutAfterFirstElementMatch)
    if NOT matched then difference:= true
  } // end for

  return NOT difference
}
```

C.2.3 `matchOrderedVectors`: Algorithm to compare two ordered aggregations

This method implements the ordered-aggregation-matching policy of Section 3.4.2. It calls `matchFillers` (Section C.1.2).

```
AggregationMatchingPolicies::matchOrderedVectors (vector1 : Vector,
                                                  vector2 : Vector,
                                                  subsForSelf : Object)
                                                 : boolean
{
  (difference : boolean):= false
  if vector1 and vector2 are of different sizes
  then difference:= true

  for each pair of elements (element1, element2)
      in corresponding positions in vector1 and vector2 respectively
  {
    if matchFillers(element1, element2, null, subsForSelf) returns false
    then difference:= true
  }
  return NOT difference
}
```

C.3 Inferring Correspondences

This section, which relates to Chapter 4, concerns the methods used to sanction correspondence between a pair of frames before they can be compared using `matchFrames` (Section C.1.1). The top-level method is `sanctionCorrespondence` (Section C.3.1), which calls `frameClassEquivalence` (Section C.3.2) and `nameEquivalence` (Section C.3.3).

This section also contains the methods used to record (Section C.3.4) and to retract (Section C.3.5) correspondence between two frames.

C.3.1 `sanctionCorrespondence`: Algorithm to sanction correspondence between two frames

This method decides whether it is appropriate to compare two frames, and if so, whether they definitely correspond or plausibly correspond. In accordance with Section 4.2.1, it calls `frameClassEquivalence` (Section C.3.2) and then `nameEquivalence` (Section C.3.3) – which returns `PLAUSIBLE_CORRESPONDENCE` if it is appropriate to use semantic equivalence as a correspondence heuristic (Section 4.2.5).

```
CorrespondenceHeuristics::sanctionCorrespondence (frame1 : Frame, frame2 : Frame)
: ENUMERATION OF (CORRESPONDENCE, PLAUSIBLE_CORRESPONDENCE, NO_CORRESPONDENCE)
{
  if frameClassEquivalence(frame1, frame2) == NO_CORRESPONDENCE
  then return NO_CORRESPONDENCE
  else return nameEquivalence(frame, frame2)
}
```

C.3.2 `frameClassEquivalence`: Algorithm for frame-class equivalence

This method vetoes correspondence between frames unless they are either of the same frame-class (Section 4.2.2) or declared to be comparable in the frame-class-thesaurus (Section 4.2.4). If it does veto correspondence, it applies the algorithm for aligning expressions with subexpressions (Section 4.3) as a side-effect (i.e. further correspondences may be inferred, but they do not affect whether *these* two frames correspond).

```
CorrespondenceHeuristics::frameClassEquivalence (frame1 : Frame, frame2 : Frame)
: ENUMERATION OF (CORRESPONDENCE, NO_CORRESPONDENCE)
{
```

```
  if frame1 and frame2 are of the same frame-class
  then return CORRESPONDENCE

  if frame1 and frame2 are declared in the frame-class-thesaurus to be comparable
  then return CORRESPONDENCE
  else {
    apply algorithm for aligning expressions with subexpressions (Section 4.3)
    return NO_CORRESPONDENCE
  }
}
```

C.3.3 `nameEquivalence`: Algorithm for name equivalence

This method sanctions correspondence between two frames if they have identical names (Section 4.2.3), or if their names are declared to be synonyms in the name-thesaurus (Section 4.2.4). Otherwise correspondence is vetoed, except that if they do not have the slot `name`, then this method returns `PLAUSIBLE_CORRESPONDENCE` so that the CCUS will proceed to compare the two frames in plausible-correspondence mode, using semantic equivalence as a correspondence heuristic (Section 4.2.5).

```
CorrespondenceHeuristics::nameEquivalence (frame1 : Frame, frame2 : Frame)
: ENUMERATION OF (CORRESPONDENCE, PLAUSIBLE_CORRESPONDENCE, NO_CORRESPONDENCE)
{
  try {
    if frame1."name" is string-equal (ignoring case) to frame2."name"
    then return CORRESPONDENCE

    if frame1."name" and frame2."name" are declared in the name-thesaurus
       to be synonyms
    then return CORRESPONDENCE
    else return NO_CORRESPONDENCE
  }
  catch (frame1 or frame2 does not have the slot "name" in its frame-class) {
    return PLAUSIBLE_CORRESPONDENCE
  }
}
```

C.3.4 `recordCorrespondence`: Algorithm to record correspondence between two frames

This method records correspondence between two frames (after their correspondence has been sanctioned, as described in the previous subsections). If it is a definite correspondence, then the frames are added to each other's `corresponds_to` slots. If it is a plausible correspondence, then a `correspondence` frame is created, and added to both frames' `corresponds_to` slots (Section 4.1.2).

This method is also responsible for removing HLCs from the remembrance structures once they have been inferred to correspond, only under the monogamy remembrance heuristic (Section 7.2.3).

Like `matchFrames` (Section C.1.1), this method records correspondence between the frames' fillers of the `covers` slot. This recursion deals with the case that there is a chain of covered frames.

```
Correspondences::recordCorrespondence (frame1 : Frame, frame2 : Frame,
                                       mode: ENUMERATION OF (CORRESPONDENCE,
                                                PLAUSIBLE_CORRESPONDENCE)
                                      : void
{
  if mode == CORRESPONDENCE then {
    insert frame2 in frame1."corresponds_to"
    insert frame1 in frame2."corresponds_to"
  }
  if mode == PLAUSIBLE_CORRESPONDENCE then {
    (correspondence : Frame):= create frame of frame-class "correspondence"
    correspondence."frame1":= frame1
    correspondence."frame2":= frame2
    correspondence."semantic_equivalence":= semantic_equivalence(equivalent)
                   // assume equivalent until proved otherwise
    insert correspondence frame in frame1."corresponds_to"
    insert correspondence frame in frame2."corresponds_to"
  }

  if using the monogamy remembrance heuristic then {
    remove frame1 from the first model's remembrance structure
    remove frame2 from the second model's remembrance structure
  }

  if frame1's "covers" slot is filled and frame2's "covers" slot is filled
  then recordCorrespondence(frame1."covers", frame2."covers", mode)
       if they have not already been recorded to correspond
}
```

C.3.5 `retractCorrespondence`: Algorithm to retract correspondence between two frames

This method retracts correspondence between two frames, either removing the `correspondence` frame from the two frames' `corresponds_to` slots and from the frame system, or removing the two frames from each other's `corresponds_to` slots, as appropriate (Section 4.1.2). It also records (in a hash-table) the pair of frames as having been compared but failed to correspond, in order to avoid repeated comparisons.

```
Correspondences::retractCorrespondence (frame1 : Frame, frame2 : Frame)
                                       : void
{
  look up the correspondence frame between frame1 and frame2
  if there is such a correspondence frame, then {
    remove correspondence frame from frame1."corresponds_to"
    remove correspondence frame from frame2."corresponds_to"
    delete correspondence frame
  } else {
    remove frame2 from frame1."corresponds_to"
    remove frame1 from frame2."corresponds_to"
```

```
  }
  record (frame1, frame2) as already compared but failed to correspond
}
```

C.4 Assessing Semantic Equivalence and Semantic Strength

This section describes how the CCUS assesses the semantic equivalence/strength (Chapter 5) between two frames which differ in a given slot. The top-level method is `useSemanticCase` (Section C.4.1), which is called by `processDifference` (Section C.1.3).

C.4.1 `useSemanticCase`: Algorithm to use a semantic-equivalence/strength case

This method assesses the semantic equivalence (Section 5.3) between two frames which differ in a given slot. An SE/S case may be passed in as input, otherwise it attempts to retrieve one (the same SE/S case is used for all slot-differences in a particular pair of frames).

This method fails the match if the slot along which the frames differ is a necessary slot. Otherwise, if it is a semantic-strength slot, it assesses whether one frame is semantically stronger than the other, whether the frames are semantically equivalent, or whether they are semantically incomparable – in which case it fails the match. It calls `compareSSSfillers` (Section C.4.2).

```
SemanticStrength::useSemanticCase (frame1 : Frame, frame2 : Frame, slot : Slot,
                                    seCase : Case)
                                   : Case
{
  if seCase == null then {
    seCase:= retrieve a case from the semantic-case library
             (using frame1 as retrieval prompt)
    if no case was retrieved OR frame2 fails to match seCase
    then exit this method, throwing FailMatch exception
  }
  (solution : Frame):= the top-level solution frame of seCase

  //////////////////////////////////
  // Process the necessary slots. //
  //////////////////////////////////
  for each slot-name n in solution."necessary_slots"
  {
    if n is string-equal to slot's name
       (i.e. if the difference is in a necessary slot)
    then exit this method, throwing FailMatch exception
```

```
  }

  /////////////////////////////////////////
  // Process the semantic-strength slots. //
  /////////////////////////////////////////
  for each frame s in solution."semantic_strength_slots"
  {
    if s."slot_name" is string-equal to slot's name
    then {
      (first : ENUMERATION OF (GREATER, LESS, EQUAL, INCOMPARABLE))
      := compareSSSfillers(frame1.slot, frame2.slot)
      ///////////////////////////////////////////
      // frame1 is stronger/weaker than frame2: //
      ///////////////////////////////////////////
      if s."influence" == "positive" then {
        if first == GREATER
        then frame1 corresponds to, and is semantically stronger than, frame2
        if first == LESS
        then frame1 corresponds to, and is semantically weaker than, frame2
      }
      if s."influence" == "negative" then {
        if first == LESS
        then frame1 corresponds to, and is semantically stronger than, frame2
        if first == GREATER
        then frame1 corresponds to, and is semantically weaker than, frame2
      }
      ////////////////////////////////////
      // frame1 is equivalent to frame2: //
      ////////////////////////////////////
      if first == EQUAL
      then frame1 corresponds to, and is semantically equivalent to, frame2
      //////////////////////////////////////
      // frame1 is incomparable to frame2: //
      //////////////////////////////////////
      if first == INCOMPARABLE
      then exit this method, throwing FailMatch exception
    }
  } // end for
  return seCase
}
```

C.4.2 `compareSSSfillers`: Algorithm to compare semantic-strength-slot-fillers

This method decides whether the fillers (from two frames) of a semantic-strength slot are equivalent, whether one is 'greater' than the other, or whether they are incomparable. It currently handles numbers, logical literals, and unordered vectors.

```
SemanticStrength::compareSSSfillers (filler1 : Object, filler2 : Object)
                  : ENUMERATION OF (GREATER, LESS, EQUAL, INCOMPARABLE)
{
  if filler1 and filler2 are number literals then {
    if filler1 > filler2 then return GREATER
    if filler1 < filler2 then return LESS
    otherwise return EQUAL
  }
```

```
  if filler1 and filler2 are logical literals then {
    if filler1 == TRUE and (filler2 == FALSE or filler2 == UNKNOWN)
    then return GREATER
    if filler1 == UNKNOWN and filler2 == FALSE
    then return GREATER
    if filler1 == UNKNOWN and filler2 == TRUE
    then return LESS
    if filler1 == FALSE and (filler2 == TRUE or filler2 == UNKNOWN)
    then return LESS
    otherwise return EQUAL
  }

  if filler1 and filler2 are unordered vectors then {
    if filler2 is a subset of filler1 AND filler1 is a subset of filler2
    then return EQUAL
    if filler2 is a proper subset of filler1
    then return GREATER
    if filler1 is a proper subset of filler2
    then return LESS
    otherwise return INCOMPARABLE
  }
}
```

C.5 Combining Comparison and Extraction

This section describes the main algorithms of the HLC CCUS (Chapter 6). Section C.5.1 describes the top-level algorithm for extracting and comparing HLCs, and Section C.5.2 the rules for submitting frames from the vicinity of the difference in both models.

Inferring abstract correspondences back at the code level when two HLCs are inferred to correspond (Section 6.3.3) is handled by `matchFrames` (Section C.1.1).

C.5.1 `extractHLCs`: Algorithm to extract and compare HLCs

This is the top-level method of the HLC CCUS, which extracts HLCs from both models (Section 6.3.1), and compares the HLCs thus extracted (Sections 6.3.2 and 7.2.2).

It calls `submitExtractionPrompts` (Section C.5.2), `matchVectors` (Section C.2.1) and `mergeHashtables` (a function to merge two hash-tables, concatenating their corresponding values, which are vectors – note that it updates the first hash-table as a side-effect, leaving it as the final merged hash-table, which is also the return-value).

The class `HLC_CCUS` has private class-variables `_r1` and `_r2`, which are hash-tables to represent the remembrance structures for the first model and the second model.

```
HLC_CCUS::extractHLCs (frame1 : Frame, frame2 : Frame, slot : Slot) : void
{
  ////////////////////////////////////
  // Extract HLCs from both models. //
  ////////////////////////////////////
  h1 : Hashtable, h2 : Hashtable

  if considering extraction prompts separately
  then {
    h1:= submitExtractionPrompts(frame1, slot)
    h2:= submitExtractionPrompts(frame2, slot)
  }
  if considering extraction prompts together
  then {
    (h1, h2):= submitExtractionPrompts(frame1, frame2, slot)
  }
  ///////////////////////////////////
  // Remember HLCs for comparison. //
  ///////////////////////////////////
  (r1' : Hashtable):= h1
  (r2' : Hashtable):= h2
  (`r1 : Hashtable):= copy of _r1

  if remembering HLCs {
    r1':= mergeHashtables(_r1, h1)
    r2':= mergeHashtables(_r2, h2)
  }
  ///////////////////
  // Compare HLCs. //
  ///////////////////
  for each key k in r1' (the keys are frame-classes) {
    (`r1v : Vector):= `r1.get(k)
    (r1'v : Vector):= r1'.get(k)
    (r2'v : Vector):= r2'.get(k)
    (h1v : Vector):= h1.get(k)
    (h2v : Vector):= h2.get(k)

    if not remembering HLCs
    then matchVectors(copy of r1'v, copy of r2'v, null, null)

    if remembering HLCs then {
      matchVectors(h1v, copy of r2'v, null, null)
      matchVectors(copy of `r1v, h2v, null, null)
    }
  } // end for
}
```

C.5.2 `submitExtractionPrompts`: Algorithm to submit extraction prompts

This method submits frames from the vicinity of the difference in both models as extraction prompts to the CUS (Section 6.3.1), and returns the HLCs that the CUS extracts (as a hash-table for each model, whose keys are frame-classes and whose values are sets of HLCs).

It calls `mergeHashtables` (see Section C.5.1), and `CCUSextract` – the CUS's HLC-CCUS-interface. The `CCUSextract` method submits a given frame first as a navigation prompt (to walk inwards, extracting HLCs from inner frames first), then as a direct extraction prompt.

The `submitExtractionPrompts` method is *overloaded*: it has a version for submitting extraction prompts separately (which takes a frame and a slot as input – Section C.5.2.1), and a version for submitting extraction prompts together (which takes two frames and a slot as input – Section C.5.2.2).

C.5.2.1 Submitting extraction prompts separately

```
HLC_CCUS::submitExtractionPrompts (frame : Frame, slot : Slot)
                                  : Hashtable
{
  if frame.slot == null
  then return empty Hashtable

  if frame.slot is a Frame
  then return CCUSextract(frame.slot)

  if frame.slot is a Vector of Frames
  then {
    (h : Hashtable):= empty Hashtable
    for each frame f in the vector {
      h:= mergeHashtables(h, CCUSextract(f))
    }
    return h
  }

  otherwise return CCUSextract(frame)
}
```

C.5.2.2 Submitting extraction prompts together

```
HLC_CCUS::submitExtractionPrompts (frame1 : Frame, frame2 : Frame, slot : Slot)
                                  : (Hashtable, Hashtable)
{
  if frame1.slot and frame2.slot are both Frames
  then return (CCUSextract(frame1.slot), CCUSextract(frame2.slot))

  if frame1.slot and frame2.slot are both Vectors of Frames
  then {
    (h1 : Hashtable):= empty Hashtable
    for each frame e in frame1.slot {
      h1:= mergeHashtables(h1, CCUSextract(e))
    }
    (h2 : Hashtable):= empty Hashtable
    for each frame f in frame2.slot {
      h2:= mergeHashtables(h2, CCUSextract(f))
    }
    return (h1, h2)
  }

  otherwise return (CCUSextract(frame1), CCUSextract(frame2))
}
```

Appendix D

The Frame-Class Model

This appendix contains the frame-class model (Section 3.2.1) for the frame-representation of EXPRESS models (Section D.1), higher-level constraints (Section D.2), and cases (Section D.3).

This frame-class model is also available from the Author's website [Bro], including frame-classes for the representation of Java programs (see Section 8.3.2.1).

D.1 Frame-Classes for EXPRESS Models

```
(***************************************************************************)
(** SCHEMA express_frame_class_model                                      **)
(***************************************************************************)
(** Frame class model of EXPRESS (an EXPRESS metamodel).                  **)
(***************************************************************************)

SCHEMA express_frame_class_model;
USE FROM hlc_frame_class_model;

  (***************************************************************)
  (* ENTITY frame                                                *)
  (***************************************************************)
  (* The superclass of all frame classes.                        *)
  (* Now /any/ frame can be extracted, not just HLCs!            *)
  (* --IGNORE is a structured comment, which indicates that a slot *)
  (* should be ignored by the CCUS frame matching algorithm.     *)
  (***************************************************************)

  ENTITY frame ABSTRACT SUPERTYPE;
    extracted_from : OPTIONAL SET [1 : ?] OF frame; --IGNORE
    covers : OPTIONAL frame; --IGNORE
    description : OPTIONAL text; --IGNORE
    corresponds_to : OPTIONAL SET [1 : ?] OF frame; --IGNORE
  INVERSE
    hlcs_extracted : SET [0 : ?] OF frame FOR extracted_from;
    covered_by : SET [0 : ?] OF frame FOR covers;
  END_ENTITY;

  (****************************************************************)
```

```
(* ENTITY named_frame                                              *)
(******************************************************************)
(* The subclasses of frames which have the distinguished slot     *)
(* "name" (so that it's the same Slot object for comparable frame *)
(* classes). Not used where it would cause multiple inheritance.  *)
(******************************************************************)

ENTITY named_frame SUBTYPE OF (frame);
  name : STRING;
END_ENTITY;

(*******************************************************************)
(* ENTITY correspondence                                            *)
(*******************************************************************)
(* A reified correspondence between two frames, used in the case of *)
(* plausible correspondences to record the semantic equivalence/    *)
(* strength of the two frames.                                      *)
(*******************************************************************)

ENTITY correspondence SUBTYPE OF (frame);
  frame1 : frame;
  frame2 : frame;
  semantic_equivalence : semantic_equivalence;
END_ENTITY;

(*****************************************************************)
(* TYPE semantic_equivalence                                      *)
(*****************************************************************)
(* Two frames can either be semantically equivalent, one can be *)
(* stronger than the other, otherwise they are incomparable.    *)
(*****************************************************************)

TYPE semantic_equivalence =
  ENUMERATION OF (equivalent, frame1_stronger, frame2_stronger,
                  incomparable);
END_TYPE;

(*****************************************************************)
(* ENTITY model_frame                                             *)
(*****************************************************************)
(* Represents an EXPRESS model (i.e. a collection of schemata). *)
(*****************************************************************)

ENTITY model_frame SUBTYPE OF (frame);
  schemata : SET [1 : ?] OF schema_frame;
END_ENTITY;

(*******************************************)
(* ENTITY schema_frame                     *)
(*******************************************)
(* Represents a schema in an EXPRESS model. *)
(*******************************************)

ENTITY schema_frame SUBTYPE OF (named_frame);
  entities : OPTIONAL SET [1 : ?] OF entity_type;
  defined_types : OPTIONAL SET [1 : ?] OF defined_type;
  rules : OPTIONAL SET [1 : ?] OF global_rule;
  functions : OPTIONAL SET [1 : ?] OF function_definition;
  procedures : OPTIONAL SET [1 : ?] OF procedure_definition;
  constants : OPTIONAL SET [1 : ?] OF constant_frame;
  uses : OPTIONAL SET [1 : ?] OF use_clause;
  references : OPTIONAL SET [1 : ?] OF reference_clause;
INVERSE
```

```
  the_model : model_frame FOR schemata;
END_ENTITY;

(*****************************************************)
(* ENTITY named_type                                 *)
(*****************************************************)
(* Superclass of named types, namely ENTITYs and TYPEs.  *)
(*****************************************************)

ENTITY named_type ABSTRACT SUPERTYPE OF
                 (ONEOF (entity_type, defined_type))
                 SUBTYPE OF (named_frame);
  domain_rules : OPTIONAL SET [1 : ?] OF domain_rule;
END_ENTITY;

(****************************************************************)
(* ENTITY entity_type                                           *)
(****************************************************************)
(* Represents an entity, which can be used anywhere a named type is *)
(* required. The attributes capture the information from the       *)
(* corresponding ENTITY definition.                             *)
(****************************************************************)

ENTITY entity_type SUBTYPE OF (named_type);
  is_abstract : BOOLEAN;
  supertype_of : OPTIONAL supertype_expression;
  subtype_of : OPTIONAL SET [1:?] OF entity_type;
  explicit_attributes : SET [0:?] OF explicit_attribute;
  explicit_attribute_redeclarations : OPTIONAL SET [1:?] OF
                              explicit_attribute_redeclaration;
  derived_attributes : OPTIONAL SET [1:?] OF derived_attribute;
  derived_attribute_redeclarations : OPTIONAL SET [1:?] OF
                               derived_attribute_redeclaration;
  inverse_attributes : OPTIONAL SET [1:?] OF inverse_attribute;
  inverse_attribute_redeclarations : OPTIONAL SET [1:?] OF
                               inverse_attribute_redeclaration;
  unique_rules : OPTIONAL SET [1:?] OF unique_rule;
INVERSE
  the_schema : schema_frame FOR entities;
END_ENTITY;

(*************************************************************)
(* ENTITY defined_type                                       *)
(*************************************************************)
(* Represents a defined type, which can be used anywhere a named *)
(* type is required. The attributes capture the information from *)
(* the corresponding TYPE definition.                        *)
(*************************************************************)

ENTITY defined_type SUBTYPE OF (named_type);
  tipe : underlying_type; -- "type" is a reserved word
INVERSE
  the_schema : schema_frame FOR defined_types;
END_ENTITY;

(************************)
(* TYPE abstract_type    *)
(************************)
(* Represents all types. *)
(************************)

TYPE abstract_type =
  SELECT (parameter_type, underlying_type, indeterminate_type);
```

```
END_TYPE;

(*****************************************************************)
(* TYPE underlying_type                                          *)
(*****************************************************************)
(* Represents the possible underlying types for a type definition: *)
(* simple types such as INTEGER, aggregations such as SET,       *)
(* entities, defined types, plus selects and enumerations.       *)
(*****************************************************************)

TYPE underlying_type = SELECT (defined_type,
                               simple_type, aggregation_type,
                               select_type, enumeration_type);
END_TYPE;

(******************************************************************)
(* TYPE parameter_type                                            *)
(******************************************************************)
(* Represents the possible types of an attribute: simple types such *)
(* as INTEGER, aggregation types such as SET, named types (entities *)
(* and defined types) but not direct selects or enumerations.      *)
(******************************************************************)

TYPE parameter_type =
  SELECT (simple_type, aggregation_type, named_type);
END_TYPE;

(******************************************************************)
(* TYPE simple_type                                               *)
(******************************************************************)
(* A simple type is either BOOLEAN, LOGICAL, NUMBER, INTEGER, REAL, *)
(* STRING, BINARY or GENERIC.                                      *)
(******************************************************************)

TYPE simple_type = ENUMERATION OF
  (boolean_type, logical_type,
  number_type, integer_type, real_type,
  string_type, binary_type,
  generic_type);
END_TYPE;

(******************************************************************)
(* TYPE indeterminate_type                                        *)
(******************************************************************)
(* The type of indeterminate aggregation-bounds ('?').            *)
(******************************************************************)

TYPE indeterminate_type = ENUMERATION OF (indeterminate);
END_TYPE;

(*****************************************************************)
(* ENTITY aggregation_type                                       *)
(*****************************************************************)
(* The supertype of aggregation types: SET, BAG, LIST and ARRAY.   *)
(* Represents AGGREGATE if the supertype is instantiated directly. *)
(*****************************************************************)

ENTITY aggregation_type SUPERTYPE OF
                        (ONEOF (unordered_aggregation_type,
                                ordered_aggregation_type))
                        SUBTYPE OF (frame);
  element_type : parameter_type;
  lower_bound : OPTIONAL expression;
```

```
  upper_bound : OPTIONAL expression;
END_ENTITY;

(********************************************************)
(* ENTITY unordered_aggregation_type                    *)
(********************************************************)
(* Represents an unordered aggregation type (SET or BAG). *)
(********************************************************)

ENTITY unordered_aggregation_type ABSTRACT SUPERTYPE OF
                                  (ONEOF (set_type, bag_type))
                                  SUBTYPE OF (aggregation_type);
END_ENTITY;

(***************************)
(* ENTITY set_type         *)
(***************************)
(* Represents a set type. *)
(***************************)

ENTITY set_type SUBTYPE OF (unordered_aggregation_type);
END_ENTITY;

(***************************)
(* ENTITY bag_type         *)
(***************************)
(* Represents a bag type. *)
(***************************)

ENTITY bag_type SUBTYPE OF (unordered_aggregation_type);
END_ENTITY;

(*********************************************************)
(* ENTITY ordered_aggregation_type                       *)
(*********************************************************)
(* Represents an ordered aggregation type (LIST or ARRAY). *)
(* The elements could have to be unique.                 *)
(*********************************************************)

ENTITY ordered_aggregation_type ABSTRACT SUPERTYPE OF
                                (ONEOF (list_type, array_type))
                                SUBTYPE OF (aggregation_type);
  is_unique : BOOLEAN;
END_ENTITY;

(****************************)
(* ENTITY list_type         *)
(****************************)
(* Represents a list type. *)
(****************************)

ENTITY list_type SUBTYPE OF (ordered_aggregation_type);
END_ENTITY;

(**********************************************************)
(* ENTITY array_type                                      *)
(**********************************************************)
(* Represents an array type. The elements could be optional. *)
(**********************************************************)

ENTITY array_type SUBTYPE OF (ordered_aggregation_type);
  is_optional : BOOLEAN;
END_ENTITY;
```

```
(**************************************************************)
(* TYPE select_type                                           *)
(**************************************************************)
(* A select is a set of named types (entities or defined types). *)
(**************************************************************)

TYPE select_type = SET [1 : ?] OF named_type;
END_TYPE;

(***************************************************************)
(* TYPE enumeration_type                                       *)
(***************************************************************)
(* An enumeration is an ordered set of symbols (enumeration types *)
(* do have an order relation).                                 *)
(***************************************************************)

TYPE enumeration_type = LIST [1 : ?] OF UNIQUE enumeration_symbol;
END_TYPE;

(**************************************************************)
(* ENTITY enumeration_symbol                                  *)
(**************************************************************)
(* An enumeration symbol is a string, now elevated to an entity. *)
(**************************************************************)

ENTITY enumeration_symbol SUBTYPE OF (named_frame);
INVERSE
  the_type : defined_type FOR tipe;
END_ENTITY;

(******************************************************************)
(* ENTITY supertype_expression                                    *)
(******************************************************************)
(* A supertype expression is either just an entity reference, or an *)
(* intermediate ONEOF, AND or ANDOR operation.                    *)
(******************************************************************)

ENTITY supertype_expression ABSTRACT SUPERTYPE OF
                            (ONEOF (oneof_supertype_expression,
                                    and_supertype_expression,
                                    andor_supertype_expression))
                            SUBTYPE OF (frame);
END_ENTITY;

(*****************************************************)
(* ENTITY oneof_supertype_expression                 *)
(*****************************************************)
(* A ONEOF expression is a set of supertype expressions. *)
(*****************************************************)

ENTITY oneof_supertype_expression SUBTYPE OF (supertype_expression);
  expressions : SET [1 : ?] OF supertype_expression;
END_ENTITY;

(*****************************************************)
(* ENTITY and_supertype_expression                   *)
(*****************************************************)
(* An AND expression is a pair of supertype expressions. *)
(*****************************************************)

ENTITY and_supertype_expression SUBTYPE OF (supertype_expression);
  terms : SET [2 : ?] OF supertype_expression;
```

```
END_ENTITY;

(*********************************************************)
(* ENTITY andor_supertype_expression                     *)
(*********************************************************)
(* An ANDOR expression is a pair of supertype expressions. *)
(*********************************************************)

ENTITY andor_supertype_expression SUBTYPE OF (supertype_expression);
  factors : SET [2 : ?] OF supertype_expression;
END_ENTITY;

(*****************************************************************)
(* ENTITY attribute                                              *)
(*****************************************************************)
(* Represents an attribute of an entity, which can either be       *)
(* explicit, INVERSE or DERIVEd. All attributes have a name and a   *)
(* type. ‘attribute’ is a subtype of ‘expression’ so that anywhere  *)
(* an expression is expected, it can be a reference to an attribute.*)
(*****************************************************************)

ENTITY attribute ABSTRACT SUPERTYPE OF (ONEOF (explicit_attribute,
                                               derived_attribute,
                                               inverse_attribute))
                 SUBTYPE OF (expression);
  name : STRING;
END_ENTITY;

(*************************************************************)
(* ENTITY explicit_attribute                                 *)
(*************************************************************)
(* Represents an explicit attribute in an entity, which may be *)
(* OPTIONAL.                                                 *)
(*************************************************************)

ENTITY explicit_attribute SUBTYPE OF (attribute);
  is_optional : BOOLEAN;
INVERSE
  the_entity : entity_type FOR explicit_attributes;
END_ENTITY;

(*****************************************************************)
(* ENTITY inverse_attribute                                      *)
(*****************************************************************)
(* Represents an INVERSE attribute in an entity, the value of which *)
(* is an entity that has this entity as the value of its FOR       *)
(* attribute.                                                    *)
(*****************************************************************)

ENTITY inverse_attribute SUBTYPE OF (attribute);
  for_attribute : attribute;
INVERSE
  the_entity : entity_type FOR inverse_attributes;
END_ENTITY;

(****************************************************************)
(* ENTITY derived_attribute                                     *)
(****************************************************************)
(* Represents a DERIVEd attribute in an entity, the value of which *)
(* is calculated using a formula rather than stored explicitly.   *)
(****************************************************************)

ENTITY derived_attribute SUBTYPE OF (attribute);
```

```
  formula : expression;
 INVERSE
  the_entity : entity_type FOR derived_attributes;
 END_ENTITY;

 (**********************************************)
 (* ENTITY attribute_redeclaration             *)
 (**********************************************)
 (* Represents a redeclaration of an attribute. *)
 (**********************************************)

 ENTITY attribute_redeclaration ABSTRACT SUPERTYPE OF (ONEOF
                                     (explicit_attribute_redeclaration,
                                       derived_attribute_redeclaration,
                                      inverse_attribute_redeclaration))
                                SUBTYPE OF (frame);
  attribute : attribute;
  entity_originally_declared_in : entity_type;
  tipe : abstract_type; -- "type" is a reserved word in EXPRESS
 END_ENTITY;

 (*******************************************************)
 (* ENTITY explicit_attribute_redeclaration             *)
 (*******************************************************)
 (* Represents a redeclaration of an explicit attribute, *)
 (* which may be OPTIONAL.                              *)
 (*******************************************************)

 ENTITY explicit_attribute_redeclaration
        SUBTYPE OF (attribute_redeclaration);
  is_optional : BOOLEAN;
 INVERSE
  the_entity: entity_type FOR explicit_attribute_redeclarations;
 END_ENTITY;

 (******************************************************)
 (* ENTITY inverse_attribute_redeclaration             *)
 (******************************************************)
 (* Represents a redeclaration of an inverse attribute. *)
 (******************************************************)

 ENTITY inverse_attribute_redeclaration
        SUBTYPE OF (attribute_redeclaration);
  for_attribute : attribute;
 INVERSE
  the_entity : entity_type FOR inverse_attribute_redeclarations;
 END_ENTITY;

 (*****************************************************)
 (* ENTITY derived_attribute_redeclaration            *)
 (*****************************************************)
 (* Represents a redeclaration of a derived attribute. *)
 (*****************************************************)

 ENTITY derived_attribute_redeclaration
        SUBTYPE OF (attribute_redeclaration);
  formula : expression;
 INVERSE
  the_entity : entity_type FOR derived_attribute_redeclarations;
 END_ENTITY;

 (************************************************************)
 (* ENTITY local_rule                                        *)
```

```
(************************************************************)
(* Represents constraints on an entity, which are either UNIQUE *)
(* rules or domain (WHERE) rules.                               *)
(************************************************************)

ENTITY local_rule ABSTRACT SUPERTYPE OF (ONEOF (unique_rule,
                                                domain_rule))
                  SUBTYPE OF (frame);
  label : STRING;
END_ENTITY;

(*******************************************)
(* ENTITY unique_rule                      *)
(*******************************************)
(* Represents a uniqueness rule on an entity. *)
(*******************************************)

ENTITY unique_rule SUBTYPE OF (local_rule);
  attributes : SET [1 : ?] OF attribute;
INVERSE
  the_entity : entity_type FOR unique_rules;
END_ENTITY;

(***************************************)
(* ENTITY domain_rule                  *)
(***************************************)
(* Represents a domain rule on an entity. *)
(***************************************)

ENTITY domain_rule SUBTYPE OF (local_rule);
  expression : expression;
INVERSE
  the_type : named_type FOR domain_rules;
END_ENTITY;

(***************************)
(* ENTITY global_rule      *)
(***************************)
(* Represents a global RULE. *)
(***************************)

ENTITY global_rule SUBTYPE OF (named_frame);
  for_entities : SET [1 : ?] OF entity_type;
  entity_types : OPTIONAL SET [1 : ?] OF entity_type;
  defined_types : OPTIONAL SET [1 : ?] OF defined_type;
  functions : OPTIONAL SET [1 : ?] OF function_definition;
  procedures : OPTIONAL SET [1 : ?] OF procedure_definition;
  constants : OPTIONAL SET [1 : ?] OF constant_frame;
  local_variables : OPTIONAL SET [1 : ?] OF local_variable;
  statements : SET [0 : ?] OF statement;
  domain_rules : SET [1 : ?] OF domain_rule;
END_ENTITY;

(********************************)
(* ENTITY expression            *)
(********************************)
(* Superclass of all expressions. *)
(********************************)

ENTITY expression ABSTRACT SUPERTYPE
                  SUBTYPE OF (frame);
  tipe : abstract_type; -- "type" is a reserved word in EXPRESS
  is_variable : BOOLEAN;
```

```
END_ENTITY;

(******************************)
(* ENTITY literal             *)
(******************************)
(* Represents a literal value. *)
(******************************)

ENTITY literal ABSTRACT SUPERTYPE
               SUBTYPE OF (expression);
END_ENTITY;

(**************************)
(* ENTITY integer_literal *)
(**************************)
(* An integer literal.    *)
(**************************)

ENTITY integer_literal SUBTYPE OF (literal);
  literal_value : INTEGER;
END_ENTITY;

(***************************)
(* ENTITY real_literal     *)
(***************************)
(* An real-number literal. *)
(***************************)

ENTITY real_literal SUBTYPE OF (literal);
  literal_value : REAL;
END_ENTITY;

(*********************************************)
(* ENTITY logical_literal                    *)
(*********************************************)
(* A logical literal (TRUE, FALSE or UNKNOWN). *)
(*********************************************)

ENTITY logical_literal SUBTYPE OF (literal);
  literal_value : LOGICAL;
END_ENTITY;

(*************************)
(* ENTITY string_literal *)
(*************************)
(* A string literal.     *)
(*************************)

ENTITY string_literal SUBTYPE OF (literal);
  literal_value : STRING;
END_ENTITY;

(*************************)
(* ENTITY binary_literal *)
(*************************)
(* A binary literal.     *)
(*************************)

ENTITY binary_literal SUBTYPE OF (literal);
  literal_value : BINARY;
END_ENTITY;

(***********************************************************)
```

```
(* ENTITY built_in_constant                                  *)
(************************************************************)
(* Represents an occurrence of one of the built-in constants. *)
(************************************************************)

ENTITY built_in_constant SUBTYPE OF (expression);
  built_in_value : built_in;
END_ENTITY;

(**********************************************)
(* TYPE built_in                              *)
(**********************************************)
(* The built in constants are e, PI, SELF and '?'. *)
(**********************************************)

TYPE built_in = ENUMERATION OF
                (e, pi_constant, self_constant, indeterminate);
END_TYPE;

(****************************************************)
(* ENTITY equation                                  *)
(****************************************************)
(* Equality relation on a set of two or more expressions *)
(* i.e. a = b = ... = z.                            *)
(****************************************************)

ENTITY equation SUBTYPE OF (expression);
  equal_expressions : SET [2 : ?] OF expression;
END_ENTITY;

(*****************************)
(* ENTITY inequality         *)
(*****************************)
(* Represents a '<>' operation. *)
(*****************************)

ENTITY inequality SUBTYPE OF (expression);
  first_expression : expression;
  second_expression : expression;
END_ENTITY;

(************************************************************)
(* Equality relation on a set of two or more entity instances *)
(* i.e. a :=: b :=: ... :=: z.                              *)
(************************************************************)

ENTITY instance_equation SUBTYPE OF (expression);
  equal_instances : SET [2 : ?] OF expression;
END_ENTITY;

(*******************************)
(* ENTITY instance_inequality  *)
(*******************************)
(* Represents a ':<>:' operation. *)
(*******************************)

ENTITY instance_inequality SUBTYPE OF (expression);
  first_instance : expression;
  second_instance : expression;
END_ENTITY;

(****************************************)
(* ENTITY comparator                    *)
```

```
(*******************************************)
(* Represents greater-than(-or-equal-to) and *)
(* less-than(-or-equal-to) operations.      *)
(*******************************************)

ENTITY comparator SUBTYPE OF (expression);
  greater : expression;
  lesser : expression;
  inclusive : inclusive_or_exclusive;
  discrete : discrete_or_continuous;
  which_way_round : way_round;
END_ENTITY;

(************************************************************)
(* TYPE way_round                                           *)
(************************************************************)
(* Encodes which way round a comparator is written ('>' or '<'). *)
(************************************************************)

TYPE way_round = ENUMERATION OF (greater_first, lesser_first);
END_TYPE;

(*******************************************************)
(* TYPE inclusive_or_exclusive                         *)
(*******************************************************)
(* Encodes whether a comparator is inclusive ('>=' or '<=') *)
(* or exclusive ('>' or '<').                          *)
(*******************************************************)

TYPE inclusive_or_exclusive = ENUMERATION OF (inclusive, exclusive);
END_TYPE;

(*************************************************)
(* TYPE discrete_or_continuous                   *)
(*************************************************)
(* Encodes whether a comparator is discrete (integer) *)
(* or continuous (real).                         *)
(*************************************************)

TYPE discrete_or_continuous = ENUMERATION OF (discrete, continuous);
END_TYPE;

(**************************************************)
(* ENTITY interval                                *)
(**************************************************)
(* Represents an interval expression, i.e. {l < m < u} *)
(* (where '<' could be '<=' instead).             *)
(**************************************************)

ENTITY interval SUBTYPE OF (expression);
  lower_bound : expression;
  lower_inclusive : inclusive_or_exclusive;
  variable : expression;
  discrete : discrete_or_continuous;
  upper_inclusive : inclusive_or_exclusive;
  upper_bound : expression;
END_ENTITY;

(*********************************************)
(* ENTITY in_operation                       *)
(*********************************************)
(* Represents an IN (set membership) operation. *)
(*********************************************)
```

```
ENTITY in_operation SUBTYPE OF (expression);
  element : expression;
  set_frame : expression;
END_ENTITY;

(**********************************************)
(* ENTITY like_operation                      *)
(**********************************************)
(* Represents a LIKE (string match) operation. *)
(**********************************************)

ENTITY like_operation SUBTYPE OF (expression);
  source_string : expression;
  target_string : expression;
END_ENTITY;

(************************************************************)
(* ENTITY addition                                          *)
(************************************************************)
(* Represents an addition as a bag of addends (or subtractors) *)
(* i.e. a + b + ... + z, where some terms could be negative.   *)
(************************************************************)

ENTITY addition SUBTYPE OF (expression);
  addends : BAG [2 : ?] OF expression;
END_ENTITY;

(*************************************************************)
(* ENTITY minus                                              *)
(*************************************************************)
(* Represents a minus (unary minus and the right-hand side of a *)
(* binary minus are treated the same).                       *)
(*************************************************************)

ENTITY minus SUBTYPE OF (expression);
  operand : expression;
END_ENTITY;

(***************************************************************)
(* ENTITY multiplication                                       *)
(***************************************************************)
(* Represents a multiplication as a bag of multiplicands (or      *)
(* divisors)                                                     *)
(* i.e. a * b * ... * z, where some factors could be reciprocals. *)
(***************************************************************)

ENTITY multiplication SUBTYPE OF (expression);
  multiplicands : BAG [2 : ?] OF expression;
END_ENTITY;

(********************************)
(* ENTITY reciprocal            *)
(********************************)
(* Represents a reciprocal (1/x). *)
(********************************)

ENTITY reciprocal SUBTYPE OF (expression);
  operand : expression;
END_ENTITY;

(*******************************)
(* ENTITY exponentiation       *)
```

```
(********************************)
(* Represents an exponentiation. *)
(********************************)

ENTITY exponentiation SUBTYPE OF (expression);
  base : expression;
  exponent : expression;
END_ENTITY;

(**************************************************************)
(* ENTITY div_operation                                       *)
(**************************************************************)
(* Represents a DIV (quotient of an integer division) operation. *)
(**************************************************************)

ENTITY div_operation SUBTYPE OF (expression);
  dividend : expression;
  divisor : expression;
END_ENTITY;

(***************************************************************)
(* ENTITY mod_operation                                        *)
(***************************************************************)
(* Represents a MOD (remainder of an integer division) operation. *)
(***************************************************************)

ENTITY mod_operation SUBTYPE OF (expression);
  dividend : expression;
  divisor : expression;
END_ENTITY;

(************************************)
(* ENTITY disjunction              *)
(************************************)
(* Abstract disjunction (OR) class. *)
(************************************)

ENTITY disjunction ABSTRACT SUPERTYPE OF (ONEOF
                      (inclusive_disjunction, exclusive_disjunction))
                   SUBTYPE OF (expression);
  disjuncts : BAG [2 : ?] OF expression;
END_ENTITY;

(**********************************************************)
(* ENTITY inclusive_disjunction                          *)
(**********************************************************)
(* Represents an inclusive disjunction as a set of disjuncts *)
(* i.e. a OR b OR ... OR z.                              *)
(**********************************************************)

ENTITY inclusive_disjunction SUBTYPE OF (disjunction);
END_ENTITY;

(**********************************************************)
(* ENTITY exclusive_disjunction                          *)
(**********************************************************)
(* Represents an exclusive disjunction as a bag of disjuncts *)
(* i.e. a XOR b XOR ... XOR z.                           *)
(**********************************************************)

ENTITY exclusive_disjunction SUBTYPE OF (disjunction);
END_ENTITY;
```

```
(**************************************************)
(* ENTITY conjunction                             *)
(**************************************************)
(* Represents a conjunction as a set of conjuncts *)
(* i.e. a AND b AND ... AND z.                    *)
(**************************************************)

ENTITY conjunction SUBTYPE OF (expression);
  conjuncts : SET [0 : ?] OF frame;
END_ENTITY;

(**************************************)
(* ENTITY negation                    *)
(**************************************)
(* Represents a negation, i.e. NOT a. *)
(**************************************)

ENTITY negation SUBTYPE OF (expression);
  operand : expression;
END_ENTITY;

(**************************************************************)
(* ENTITY complex_constructor                                 *)
(**************************************************************)
(* Represents a complex construction as a list of two or more *)
(* entity initialisers, i.e. a || b || ... || z.              *)
(**************************************************************)

ENTITY complex_constructor SUBTYPE OF (expression);
  entity_initialisers : LIST [2 : ?] OF expression;
END_ENTITY;

(***************************)
(* ENTITY query_expression *)
(***************************)
(* Represents a query.     *)
(***************************)

ENTITY query_expression SUBTYPE OF (expression);
  variable : query_variable;
  aggregation_expression : expression;
  logical_expression : expression;
END_ENTITY;

(*********************************************)
(* ENTITY query_variable                     *)
(*********************************************)
(* Represents the bound variable in a query. *)
(*********************************************)

ENTITY query_variable SUBTYPE OF (expression);
  name : STRING;
INVERSE
  the_query : query_expression FOR variable;
END_ENTITY;

(************************************)
(* ENTITY function_call             *)
(************************************)
(* Represents a call to a function. *)
(************************************)

ENTITY function_call SUBTYPE OF (expression);
```

```
  function_called : function_definition;
  actual_parameters : LIST [0 : ?] OF expression;
END_ENTITY;

(***********************************)
(* ENTITY function_definition       *)
(***********************************)
(* Represents a FUNCTION definition. *)
(***********************************)

ENTITY function_definition ABSTRACT SUPERTYPE OF
                           (ONEOF(built_in_function,
                                  user_defined_function))
                           SUBTYPE OF (named_frame);
  formal_parameters : LIST [0 : ?] OF formal_parameter;
  return_type : abstract_type;
END_ENTITY;

(************************************************)
(* ENTITY built_in_function                      *)
(************************************************)
(* Represents the signature of a built-in function. *)
(************************************************)

ENTITY built_in_function SUBTYPE OF (function_definition);
END_ENTITY;

(*****************************************************)
(* ENTITY user_defined_function                       *)
(*****************************************************)
(* Represents the definition of a user-defined function. *)
(*****************************************************)

ENTITY user_defined_function SUBTYPE OF (function_definition);
  parameter_type : parameter_type;
  constants : OPTIONAL SET [1 : ?] OF constant_frame;
  entities : OPTIONAL SET [1 : ?] OF entity_type;
  functions : OPTIONAL SET [1 : ?] OF user_defined_function;
  procedures : OPTIONAL SET [1 : ?] OF procedure_definition;
  local_variables : OPTIONAL SET [1 : ?] OF local_variable;
  body : SET [1 : ?] OF statement;
END_ENTITY;

(***************************************************************)
(* ENTITY formal_parameter                                      *)
(***************************************************************)
(* Represents a formal parameter of a function or procedure.    *)
(* User-defined functions/procedures cannot have optional formal *)
(* parameters. The is_optional slot is for built-in functions such *)
(* as ATAN, which can be called with one parameter or with two.  *)
(* I decided to model this as one function which has an optional *)
(* parameter, rather than two functions with the same name.      *)
(***************************************************************)

ENTITY formal_parameter SUBTYPE OF (named_frame);
  tipe : parameter_type;
  is_optional : BOOLEAN;
INVERSE
  the_function : function_definition FOR formal_parameters;
END_ENTITY;

(************************************)
(* ENTITY procedure_definition       *)
```

```
(*************************************)
(* Represents a PROCEDURE definition. *)
(*************************************)

ENTITY procedure_definition SUBTYPE OF (frame);
END_ENTITY;

(**************************)
(* ENTITY qualifier       *)
(**************************)
(* Represents a qualifier. *)
(**************************)

ENTITY qualifier ABSTRACT SUPERTYPE OF (ONEOF (attribute_qualifier,
                                               group_qualifier,
                                               index_qualifier))
                 SUBTYPE OF (expression);
  qualified_factor : expression;
END_ENTITY;

(**********************************************************)
(* ENTITY attribute_qualifier                             *)
(**********************************************************)
(* Represents an attribute qualifier as a qualified factor l  *)
(* and an attribute r, i.e. "l.r". Attribute qualification is *)
(* left-associative, i.e. (c.b.a) = ((c.b).a)             *)
(**********************************************************)

ENTITY attribute_qualifier SUBTYPE OF (qualifier);
  attribute_reference : attribute;
END_ENTITY;

(****************************************************)
(* ENTITY group_qualifier                           *)
(****************************************************)
(* Represents a group qualifier as a qualified factor l *)
(* and an entity r, i.e. "l\r".                     *)
(****************************************************)

ENTITY group_qualifier SUBTYPE OF (qualifier);
  entity_reference : entity_type;
END_ENTITY;

(***************************************************)
(* ENTITY index_qualifier                          *)
(***************************************************)
(* Represents an index qualifier, i.e. a[i] or a[i:j] *)
(***************************************************)

ENTITY index_qualifier SUBTYPE OF (qualifier);
  index1 : expression;
  index2 : OPTIONAL expression;
END_ENTITY;

(***************************************************************)
(* TYPE unresolved_reference                                   *)
(***************************************************************)
(* Yes, I'm afraid the current frame system cannot resolve all      *)
(* references. So this is officially represented in this model! ;-) *)
(***************************************************************)

TYPE unresolved_reference = STRING;
END_TYPE;
```

```
  (*************************************)
  (* ENTITY aggregate_initialiser      *)
  (*************************************)
  (* Represents an aggregation literal. *)
  (*************************************)

  ENTITY aggregate_initialiser SUBTYPE OF (expression);
    elements : LIST [0 : ?] OF aggregate_initialiser_element;
  END_ENTITY;

  (*****************************************************)
  (* ENTITY aggregate_initialiser_element              *)
  (*****************************************************)
  (* Represents an element in an aggregation literal. *)
  (*****************************************************)

  ENTITY aggregate_initialiser_element;
    expression : expression;
    repeat_count : OPTIONAL expression;
  INVERSE
    the_aggregate_initialiser : aggregate_initialiser
                                FOR elements;
  END_ENTITY;

  (**************************)
  (* ENTITY constant_frame  *)
  (**************************)
  (* Represents a CONSTANT. *)
  (**************************)

  ENTITY constant_frame SUBTYPE OF (frame);
  END_ENTITY;

  (*****************************)
  (* ENTITY use_clause        *)
  (*****************************)
  (* Represents a USES clause. *)
  (*****************************)

  ENTITY use_clause SUBTYPE OF (frame);
  END_ENTITY;

  (***********************************)
  (* ENTITY reference_clause        *)
  (***********************************)
  (* Represents a REFERENCES clause. *)
  (***********************************)

  ENTITY reference_clause SUBTYPE OF (frame);
  END_ENTITY;
END_SCHEMA;
```

D.2 Frame-Classes for Higher-Level Constraints

```
(****************************************************************************)
(** SCHEMA hlc_frame_class_model                                           **)
(****************************************************************************)
(** Frame class model of higher-level constraints.                         **)
(****************************************************************************)
```

```
SCHEMA hlc_frame_class_model;
USE FROM express_frame_class_model;

  (**************************************************)
  (* ENTITY higher_level_constraint                 *)
  (**************************************************)
  (* The superclass of all higher-level constraints. *)
  (**************************************************)

  ENTITY higher_level_constraint ABSTRACT SUPERTYPE
                                 SUBTYPE OF (frame);
  END_ENTITY;

  (********************************************)
  (* ENTITY choice_set                        *)
  (********************************************)
  (* Higher-level CHOICE_SET constraint.      *)
  (* This replaces EXACTLY_ONE and AT_MOST_ONE. *)
  (********************************************)

  ENTITY choice_set SUBTYPE OF (higher_level_constraint);
    choice_set : SET [1 : ?] OF frame;
    is_optional : BOOLEAN;
    constrained_entity : entity_type;
    attribute : OPTIONAL attribute;
  END_ENTITY;

  (***************************************************************)
  (* ENTITY bound                                                *)
  (***************************************************************)
  (* Higher-level BOUND constraint on a variable expression which *)
  (* must be <, >, <= or >= some limit.                          *)
  (***************************************************************)

  ENTITY bound SUBTYPE OF (higher_level_constraint);
    kind : bound_kind;
    inclusive : inclusive_or_exclusive;
    discrete : discrete_or_continuous;
    variable : expression;
    limit : expression;
  END_ENTITY;

  (****************************************)
  (* TYPE bound_kind                      *)
  (****************************************)
  (* A bound can either be lower or upper. *)
  (****************************************)

  TYPE bound_kind = ENUMERATION OF (lower, upper);
  END_TYPE;

  (****************************************)
  (* ENTITY range                         *)
  (****************************************)
  (* Higher-level numeric RANGE constraint. *)
  (****************************************)

  ENTITY range SUBTYPE OF (higher_level_constraint);
    lower_bound : bound;
    upper_bound : bound;
    variable : expression;
    discrete : discrete_or_continuous;
  END_ENTITY;
```

```
(*******************************************************************)
(* ENTITY non_null                                                 *)
(*******************************************************************)
(* Higher-level constraint that the value of an explicit attribute *)
(* must not be null.                                               *)
(*******************************************************************)

ENTITY non_null SUBTYPE OF (higher_level_constraint);
  attribute : explicit_attribute;
END_ENTITY;

(*******************************************************************)
(* ENTITY type_hlc                                                 *)
(*******************************************************************)
(* Higher-level constraint that the value of an explicit attribute *)
(* must be of a certain type.                                      *)
(*******************************************************************)

ENTITY type_hlc SUBTYPE OF (higher_level_constraint);
  attribute : explicit_attribute;
  tipe : abstract_type;
END_ENTITY;

(*******************************************************************)
(* ENTITY domain_rules_hlc                                         *)
(*******************************************************************)
(* Higher-level constraint used to propagate domain rules from     *)
(* a defined type to an entity through an attribute.               *)
(*******************************************************************)

ENTITY domain_rules_hlc SUBTYPE OF (higher_level_constraint);
  the_entity : entity_type;
  domain_rules : OPTIONAL SET [1 : ?] OF domain_rule;
  substitute_for_self : OPTIONAL expression; --IGNORE
END_ENTITY;

(**************************************************************)
(* ENTITY optional_xor_empty_aggregation                      *)
(**************************************************************)
(* Higher-level constraint that captures the common semantics *)
(* of OPTIONAL SET [1 : ?] and SET [0 : ?].                   *)
(**************************************************************)

ENTITY optional_xor_empty_aggregation
SUBTYPE OF (higher_level_constraint);
  attribute : explicit_attribute;
END_ENTITY;

(**********************************)
(* ENTITY adder                   *)
(**********************************)
(* Higher-level adder constraint. *)
(**********************************)

ENTITY adder SUBTYPE OF (higher_level_constraint);
  addends : BAG [2 : ?] OF expression;
  sum : expression;
END_ENTITY;

(***************************************)
(* ENTITY multiplier                   *)
(***************************************)
```

```
(* Higher-level multiplier constraint. *)
(**************************************)

ENTITY multiplier SUBTYPE OF (higher_level_constraint);
  multiplicands : BAG [2 : ?] OF expression;
  product : expression;
END_ENTITY;

(****************************************)
(* ENTITY relationship                  *)
(****************************************)
(* Higher-level relationship constraint. *)
(****************************************)

ENTITY relationship ABSTRACT SUPERTYPE OF
                    (ONEOF (one_to_one_relationship,
                            one_to_many_relationship,
                            many_to_many_relationship))
                    SUBTYPE OF (higher_level_constraint);
  source_entity : entity_type;
  source_attribute : attribute;
  target_entity : entity_type;
  target_attribute : attribute;
END_ENTITY;

(****************************************************)
(* ENTITY one_to_one_relationship                   *)
(****************************************************)
(* Higher-level one-to-one relationship constraint. *)
(****************************************************)

ENTITY one_to_one_relationship SUBTYPE OF (relationship);
END_ENTITY;

(*****************************************************)
(* ENTITY one_to_many_relationship                   *)
(*****************************************************)
(* Higher-level one-to-many relationship constraint. *)
(*****************************************************)

ENTITY one_to_many_relationship SUBTYPE OF (relationship);
END_ENTITY;

(******************************************************)
(* ENTITY many_to_many_relationship                   *)
(******************************************************)
(* Higher-level many-to-many relationship constraint. *)
(******************************************************)

ENTITY many_to_many_relationship SUBTYPE OF (relationship);
END_ENTITY;

(*****************************************************************)
(* ENTITY quantifier                                             *)
(*****************************************************************)
(* Superclass for higher-level quantifier (universal/existential) *)
(* constraints.                                                  *)
(*****************************************************************)

ENTITY quantifier ABSTRACT SUPERTYPE OF (ONEOF(for_all, there_exists))
                  SUBTYPE OF (higher_level_constraint);
  bound_variable : query_variable;
  aggregation : expression;
```

```
    proposition : expression;
  END_ENTITY;

  (**************************************************)
  (* ENTITY for_all                                 *)
  (**************************************************)
  (* Higher-level universal quantifier constraint. *)
  (**************************************************)

  ENTITY for_all SUBTYPE OF (quantifier);
  END_ENTITY;

  (****************************************************)
  (* ENTITY there_exists                              *)
  (****************************************************)
  (* Higher-level existential quantifier constraint. *)
  (****************************************************)

  ENTITY there_exists SUBTYPE OF (quantifier);
  END_ENTITY;
END_SCHEMA;
```

D.3 Frame-Classes for Cases

```
(*****************************************************************************)
(** SCHEMA cbr_frame_class_model                                            **)
(*****************************************************************************)
(** Frame class model of case-based reasoning.                              **)
(*****************************************************************************)

SCHEMA cbr_frame_class_model;
USE FROM express_frame_class_model;
USE FROM hlc_frame_class_model;

  (****************************************************************)
  (* ENTITY case_frame                                            *)
  (****************************************************************)
  (* Represents a case, which has a problem and a solution.       *)
  (* The solution is no longer restricted to being a higher-level *)
  (* constraint, as was the case in Version 1. It could now be an *)
  (* EXPRESS frame (making the implicit explicit), or a cliche or *)
  (* design pattern extracted from a program. It can also be a    *)
  (* set of frames now.                                           *)
  (****************************************************************)

  ENTITY case_frame ABSTRACT SUPERTYPE OF (ONEOF (extraction_case,
                                                 navigation_case,
                                                 pretty_print_case,
                                                 subexpression_case,
                                                 semantic_case))
                  SUBTYPE OF (frame);
    problem : frame;
    solution : frame_or_set_of_frames;
    language : language;
    filename : STRING;
  END_ENTITY;

  (******************************)
  (* TYPE frame_or_set_of_frames *)
  (******************************)
```

```
TYPE frame_or_set_of_frames = SELECT (frame, set_of_frames);
END_TYPE;

(*********************)
(* TYPE set_of_frames *)
(*********************)

TYPE set_of_frames = SET [1 : ?] OF frame;
END_TYPE;

(****************************************************)
(* TYPE language                                    *)
(****************************************************)
(* Represents the various languages that the CUS knows. *)
(****************************************************)

TYPE language = ENUMERATION OF (express,java);
END_TYPE;

(*****************************************************)
(* ENTITY extraction_case                            *)
(*****************************************************)
(* Represents a case for extracting knowledge from code. *)
(*****************************************************)

ENTITY extraction_case SUBTYPE OF (case_frame);
END_ENTITY;

(**************************************************************)
(* ENTITY navigation_case                                     *)
(**************************************************************)
(* Represents a case for navigating code, suggesting which frames *)
(* to submit as extraction prompts.                           *)
(**************************************************************)

ENTITY navigation_case SUBTYPE OF (case_frame);
END_ENTITY;

(*********************************************************)
(* ENTITY subexpression_case                             *)
(*********************************************************)
(* Represents a case for matching subexpressions in the CCUS *)
(* (e.g. age >= 0, (age >= 0) AND (age <= 130)).         *)
(*********************************************************)

ENTITY subexpression_case SUBTYPE OF (case_frame);
END_ENTITY;

(**************************************************************)
(* ENTITY submit                                              *)
(**************************************************************)
(* Represents a suggestion of what frames to submit as extraction *)
(* prompts, as used in the solution part of navigation cases.    *)
(**************************************************************)

ENTITY submit SUBTYPE OF (frame);
  frames : SET [1 : ?] OF frame;
END_ENTITY;

(***********************************************)
(* ENTITY semantic_case                        *)
(***********************************************)
(* Represents a semantic-equivalence/strength case. *)
```

```
(****************************************************)

ENTITY semantic_case SUBTYPE OF (case_frame);
END_ENTITY;

(*************************************************************)
(* ENTITY semantic_match_prescription                        *)
(*************************************************************)
(* The solution part of a semantic-equivalence/strength case -    *)
(* it specifies which slots are necessary to match for the frames *)
(* correspond, and which slots affect semantic strength.          *)
(*************************************************************)

ENTITY semantic_match_prescription SUBTYPE OF (frame);
  necessary_slots : SET [0 : ?] OF slot_name;
  semantic_strength_slots : SET [0:?] OF semantic_strength_slot;
END_ENTITY;

(**********************************)
(* TYPE slot_name;              *)
(**********************************)
(* A slot name is simply a string. *)
(**********************************)

TYPE slot_name = STRING;
END_TYPE;

(************************************************************)
(* ENTITY semantic_strength_slot                            *)
(************************************************************)
(* The specification, within a semantic match prescription,     *)
(* of a slot that affects semantic strength, and whether it does *)
(* so positively (i.e. the stronger frame will have the greater  *)
(* (numerical) value for this slot) or negatively.              *)
(************************************************************)

ENTITY semantic_strength_slot SUBTYPE OF (frame);
  slot_name : slot_name;
  influence : semantic_strength_influence;
INVERSE
  the_semantic_match_prescription : semantic_match_prescription
                                    FOR semantic_strength_slots;
END_ENTITY;

(**********************************************************)
(* TYPE semantic_strength_influence                       *)
(**********************************************************)
(* A slot can influence semantic strength either positively or *)
(* negatively.                                            *)
(**********************************************************)

TYPE semantic_strength_influence = ENUMERATION OF (positive,negative);
END_TYPE;

(***********************************************)
(* ENTITY pretty_print_case                    *)
(***********************************************)
(* Represents a case for pretty-printing HLCs,     *)
(* i.e. generating a natural-language description. *)
(***********************************************)

ENTITY pretty_print_case SUBTYPE OF (case_frame);
END_ENTITY;
```

```
  (*****************************************************************)
  (* ENTITY pretty_print                                           *)
  (*****************************************************************)
  (* Represents how to generate a natural language description for *)
  (* an HLC, as used in the solution part of pretty-print cases.   *)
  (*****************************************************************)

  ENTITY pretty_print SUBTYPE OF (frame);
    text : LIST [1 : ?] OF pretty_print_type;
  END_ENTITY;

  (**************************************************)
  (* TYPE pretty_print_type;                        *)
  (**************************************************)
  (* The possible element types for a pretty_print. *)
  (**************************************************)

  TYPE pretty_print_type
    = SELECT (text, frame, pp_set, pp_bag, pp_list);
  END_TYPE;

  (*****************************)
  (* TYPE text;                *)
  (*****************************)
  (* STRING as a defined type. *)
  (*****************************)

  TYPE text = STRING;
  END_TYPE;

  (*******************************************)
  (* TYPE pp_set;                            *)
  (*******************************************)
  (* SET as a defined type for pretty_print. *)
  (*******************************************)

  TYPE pp_set = SET OF pretty_print_type;
  END_TYPE;

  (*******************************************)
  (* TYPE pp_bag;                            *)
  (*******************************************)
  (* BAG as a defined type for pretty_print. *)
  (*******************************************)

  TYPE pp_bag = BAG OF pretty_print_type;
  END_TYPE;

  (********************************************)
  (* TYPE pp_list;                            *)
  (********************************************)
  (* LIST as a defined type for pretty_print. *)
  (********************************************)

  TYPE pp_list = LIST OF pretty_print_type;
  END_TYPE;
END_SCHEMA;
```

Appendix E

The Case Library

This appendix contains the complete set of FIL-C cases (Appendix B) for extraction cases (Section E.1), navigation cases (Section E.2), semantic-equivalence/strength cases (Section E.3) and subexpression cases (Section E.4). Section E.5 introduces pretty-print cases.

All cases are available on the Author's website [Bro].

E.1 Extraction Cases

Extraction cases are discussed in Appendix A and Appendix B, Section B.1.

E.1.1 `choice_set` from ENUMERATION

```
(**********************************************)
(* choice_set-enumeration.filc 2000:10:09 17:20 *)
(**********************************************)
(* ENTITY person;                               *)
(*   sex : (OPTIONAL) gender;                   *)
(* END_ENTITY;                                  *)
(*                                              *)
(* TYPE gender = ENUMERATION OF (male,female);  *)
(* END_TYPE;                                    *)
(**********************************************)
(* CHOICE_SET                                   *)
(*      choice_set = {male,female}              *)
(*      is_optional = ?optional                 *)
(*      extracted_from = gender                 *)
(*      constrained_entity = person             *)
(*      attribute = sex                         *)
(**********************************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting choice_set from an enumeration";
  problem = explicit_attribute#1;
  solution = choice_set#1;
```

```
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE explicit_attribute#1;
  tipe = defined_type#1;
  is_optional = ?optional;
  the_entity = ?entity;
END_INSTANCE;

INSTANCE defined_type#1;
  tipe = enumeration_type(?list);
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE choice_set#1;
  choice_set = ?list;
  is_optional = ?optional;
  extracted_from = {defined_type#1};
  constrained_entity = ?entity;
  attribute = explicit_attribute#1;
END_INSTANCE;
```

E.1.2 choice_set from SELECT

```
(*********************************************)
(* choice_set-select.filc   2000:10:09 17:23 *)
(*********************************************)
(* ENTITY person;                            *)
(*   sex : (OPTIONAL) gender;                *)
(* END_ENTITY;                               *)
(*                                           *)
(* TYPE gender = SELECT (manhood,womanhood); *)
(* END_TYPE;                                 *)
(*                                           *)
(* ENTITY manhood;                           *)
(* END_ENTITY;                               *)
(*                                           *)
(* ENTITY womanhood;                         *)
(* END_ENTITY;                               *)
(*********************************************)
(* CHOICE_SET                                *)
(*      choice_set = {manhood,womanhood}     *)
(*      is_optional = ?optional              *)
(*      extracted_from = gender              *)
(*      constrained_entity = person          *)
(*      attribute = sex                      *)
(*********************************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting choice_set from a select";
  problem = explicit_attribute#1;
  solution = choice_set#1;
END_INSTANCE;

(***********)
```

```
(* Problem *)
(***********)

INSTANCE explicit_attribute#1;
  tipe = defined_type#1;
  is_optional = ?optional;
  the_entity = ?entity;
END_INSTANCE;

INSTANCE defined_type#1;
  tipe = select_type(?set);
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE choice_set#1;
  choice_set = ?set;
  is_optional = ?optional;
  extracted_from = {defined_type#1};
  constrained_entity = ?entity;
  attribute = explicit_attribute#1;
END_INSTANCE;
```

E.1.3 choice_set from ABSTRACT SUPERTYPE

```
(**********************************************************)
(* exactly_one-supertype.filc            2000:10:09 18:12 *)
(**********************************************************)
(* ENTITY person ABSTRACT SUPERTYPE OF (ONEOF (man,woman)); *)
(* END_ENTITY;                                            *)
(*                                                        *)
(* ENTITY man SUBTYPE OF (person);                        *)
(* END_ENTITY;                                            *)
(*                                                        *)
(* ENTITY woman SUBTYPE OF (person);                      *)
(* END_ENTITY;                                            *)
(**********************************************************)
(* CHOICE_SET                                             *)
(*     choice_set = {man,woman}                           *)
(*     is_optional = false                                *)
(*     extracted_from = ONEOF(man,woman)                  *)
(*     constrained_entity = person                        *)
(**********************************************************)

INSTANCE extraction_case#1;
  language = express;
  description =
   "A case for extracting exactly_one from an abstract oneof supertype";
  problem = entity_type#1;
  solution = choice_set#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE entity_type#1;
  is_abstract = true;
  supertype_of = oneof_supertype_expression#1;
END_INSTANCE;
```

```
INSTANCE oneof_supertype_expression#1;
  expressions = ?set;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE choice_set#1;
  choice_set = ?set;
  is_optional = false;
  extracted_from = {oneof_supertype_expression#1};
  constrained_entity = entity_type#1;
END_INSTANCE;
```

E.1.4 choice_set from concrete SUPERTYPE

```
(*************************************************)
(* at_most_one-supertype.filc     2000:10:09 18:12 *)
(*************************************************)
(* ENTITY person SUPERTYPE OF (ONEOF (man,woman)); *)
(* END_ENTITY;                                     *)
(*                                                 *)
(* ENTITY man SUBTYPE OF (person);                 *)
(* END_ENTITY;                                     *)
(*                                                 *)
(* ENTITY woman SUBTYPE OF (person);               *)
(* END_ENTITY;                                     *)
(*************************************************)
(* CHOICE_SET                                      *)
(*      choice_set = {man,woman}                   *)
(*      is_optional = true                         *)
(*      extracted_from = ONEOF(man,woman)          *)
(*      constrained_entity = person                *)
(*************************************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting at_most_one from a concrete oneof supertype";
  problem = entity_type#1;
  solution = choice_set#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE entity_type#1;
  is_abstract = false;
  supertype_of = oneof_supertype_expression#1;
END_INSTANCE;

INSTANCE oneof_supertype_expression#1;
  expressions = ?set;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE choice_set#1;
```

```
  choice_set = ?set;
  is_optional = true;
  extracted_from = {oneof_supertype_expression#1};
  constrained_entity = entity_type#1;
END_INSTANCE;
```

E.1.5 Lower bound

```
(************************************)
(* bound-lower.filc 2000:11:09 15:45 *)
(************************************)
(* COMPARATOR                       *)
(*      greater = EXPRESSION#1      *)
(*      lesser = EXPRESSION#2       *)
(*      inclusive = ?inclusive      *)
(*      discrete = ?discrete        *)
(*                                  *)
(* EXPRESSION#1                     *)
(*      is_variable = true          *)
(*                                  *)
(* EXPRESSION#1                     *)
(*      is_variable = false         *)
(************************************)
(* BOUND                            *)
(*      kind = lower                *)
(*      variable = EXPRESSION#1     *)
(*      limit = EXPRESSION#2        *)
(*      inclusive = ?inclusive      *)
(*      discrete = ?discrete        *)
(*      extracted_from = COMPARATOR  *)
(*      covers = COMPARATOR         *)
(************************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting a lower bound from a comparator";
  problem = comparator#1;
  solution = bound#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE comparator#1;
  greater = expression#1;
  lesser = expression#2;
  inclusive = ?inclusive;
  discrete = ?discrete;
END_INSTANCE;

INSTANCE expression#1;
  is_variable = true;
END_INSTANCE;

INSTANCE expression#2;
  is_variable = false;
END_INSTANCE;

(************)
(* Solution *)
(************)
```

```
INSTANCE bound#1;
  kind = lower;
  variable = expression#1;
  limit = expression#2;
  inclusive = ?inclusive;
  discrete = ?discrete;
  extracted_from = {comparator#1};
  covers = comparator#1;
END_INSTANCE;
```

E.1.6 Upper bound

```
(*************************************)
(* bound-upper.filc 2000:11:09 15:47 *)
(*************************************)
(* COMPARATOR                        *)
(*      greater = EXPRESSION#1       *)
(*      lesser = EXPRESSION#2        *)
(*      inclusive = ?inclusive       *)
(*      discrete = ?discrete         *)
(*                                   *)
(* EXPRESSION#1                      *)
(*      is_variable = false          *)
(*                                   *)
(* EXPRESSION#1                      *)
(*      is_variable = true           *)
(*************************************)
(* BOUND                             *)
(*      kind = upper                 *)
(*      variable = EXPRESSION#2      *)
(*      limit = EXPRESSION#1         *)
(*      inclusive = ?inclusive       *)
(*      discrete = ?discrete         *)
(*      extracted_from = COMPARATOR  *)
(*      covers = COMPARATOR          *)
(*************************************)

INSTANCE extraction_case#1;
  language = express;
  description= "A case for extracting an upper bound from a comparator";
  problem = comparator#1;
  solution = bound#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE comparator#1;
  greater = expression#1;
  lesser = expression#2;
  inclusive = ?inclusive;
  discrete = ?discrete;
END_INSTANCE;

INSTANCE expression#1;
  is_variable = false;
END_INSTANCE;

INSTANCE expression#2;
  is_variable = true;
```

```
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE bound#1;
  kind = upper;
  variable = expression#2;
  limit = expression#1;
  inclusive = ?inclusive;
  discrete = ?discrete;
  extracted_from = {comparator#1};
  covers = comparator#1;
END_INSTANCE;
```

E.1.7 range from two bounds

```
(**********************************************)
(* range-from-2-bounds.filc    2000:11:09 16:55 *)
(**********************************************)
(* CONJUNCTION                                *)
(*      conjuncts = {comparator#1,comparator#2} *)
(*                                            *)
(* COMPARATOR#1                               *)
(*      HLCs_extracted = {bound#1}            *)
(*                                            *)
(* COMPARATOR#2                               *)
(*      HLCs_extracted = {bound#2}            *)
(*                                            *)
(* BOUND#1                                    *)
(*      kind = lower                          *)
(*      variable = ?variable                  *)
(*      discrete = ?discrete                  *)
(*                                            *)
(* BOUND#2                                    *)
(*      kind = upper                          *)
(*      variable = ?variable                  *)
(*      discrete = ?discrete                  *)
(**********************************************)
(* RANGE                                      *)
(*      lower_bound = bound#1;                *)
(*      upper_bound = bound#2;                *)
(*      extracted_from = {bound#1,bound#2}    *)
(*      variable = ?variable                  *)
(*      discrete = ?discrete                  *)
(*      covers = CONJUNCTION                  *)
(**********************************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting range from two bounds in conjunction";
  problem = conjunction#1;
  solution = range#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE conjunction#1;
  conjuncts = {comparator#1,comparator#2};
```

```
END_INSTANCE;

INSTANCE comparator#1;
  hlcs_extracted = {bound#1};
END_INSTANCE;

INSTANCE comparator#2;
  hlcs_extracted = {bound#2};
END_INSTANCE;

INSTANCE bound#1;
  kind = lower;
  variable = ?variable;
  discrete = ?discrete;
END_INSTANCE;

INSTANCE bound#2;
  kind = upper;
  variable = ?variable;
  discrete = ?discrete;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE range#1;
  lower_bound = bound#1;
  upper_bound = bound#2;
  variable = ?variable;
  discrete = ?discrete;
  extracted_from = {bound#1,bound#2};
  covers = conjunction#1;
END_INSTANCE;
```

E.1.8 range from interval

```
(*********************************************)
(* range-from-interval.filc 2000:11:09 16:58 *)
(*********************************************)
(* INTERVAL                                  *)
(*      lower_bound = ?lower_bound           *)
(*      lower_inclusive = ?lower_inclusive   *)
(*      variable = ?variable                 *)
(*      upper_inclusive = ?upper_inclusive   *)
(*      upper_bound = ?upper_inclusive       *)
(*********************************************)
(* RANGE                                     *)
(*      lower_bound = BOUND#1;               *)
(*      upper_bound = BOUND#2;               *)
(*      variable = ?variable                 *)
(*      discrete = ?discrete                 *)
(*      extracted_from = {BOUND#1,BOUND#2}   *)
(*      covers = INTERVAL                    *)
(*                                           *)
(* BOUND#1                                   *)
(*      kind = lower                         *)
(*      inclusive = ?lower_inclusive         *)
(*      discrete = ?discrete                 *)
(*      variable = ?variable                 *)
(*      limit = ?lower_bound                 *)
(*                                           *)
```

```
(* BOUND#2                                 *)
(*      kind = upper                       *)
(*      inclusive = ?upper_inclusive       *)
(*      discrete = ?discrete               *)
(*      variable = ?variable               *)
(*      limit = ?upper_bound               *)
(*******************************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting a range and two bounds from an interval";
  problem = interval#1;
  solution = range#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE interval#1;
  lower_bound = ?lower_bound;
  lower_inclusive = ?lower_inclusive;
  variable = ?variable;
  discrete = ?discrete;
  upper_inclusive = ?upper_inclusive;
  upper_bound = ?upper_bound;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE range#1;
  lower_bound = bound#1;
  upper_bound = bound#2;
  variable = ?variable;
  discrete = ?discrete;
  extracted_from = {interval#1};
  covers = interval#1;
END_INSTANCE;

INSTANCE bound#1;
  kind = lower;
  inclusive = ?lower_inclusive;
  variable = ?variable;
  discrete = ?discrete;
  limit = ?lower_bound;
END_INSTANCE;

INSTANCE bound#2;
  kind = upper;
  inclusive = ?upper_inclusive;
  variable = ?variable;
  discrete = ?discrete;
  limit = ?upper_bound;
END_INSTANCE;
```

E.1.9 `equation` from derived attribute

```
(***************************************)
(* equation-derive.filc 1999:11:18 16:04 *)
```

```
(***************************************)
(* DERIVE ?x := ?y                     *)
(***************************************)
(* EQUATION                            *)
(*      equals = {?x,?y}               *)
(*      extracted_from = DERIVE ?x := ?y *)
(***************************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting an equation from a derived attribute";
  problem = derived_attribute#1;
  solution = equation#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE derived_attribute#1;
  formula = ?z;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE equation#1;
  equal_expressions = {derived_attribute#1, ?z};
  extracted_from = {derived_attribute#1};
END_INSTANCE;
```

E.1.10 adder from $x + y = z$

```
(**********************************)
(* adder-plus.filc 2000:11:16 13:40 *)
(**********************************)
(* EQUATION                       *)
(*      ?x + ?y = ?z              *)
(**********************************)
(* ADDER                          *)
(*      addends = {?x,?y}         *)
(*      sum = ?z                  *)
(*      extracted_from = EQUATION   *)
(*      covers = EQUATION         *)
(**********************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting an adder from an equation of the form x+y=z";
  problem = equation#1;
  solution = adder#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE equation#1;
  equal_expressions = {addition#1, expression#1};
END_INSTANCE;
```

```
INSTANCE addition#1;
  addends = ?addends;
END_INSTANCE;

INSTANCE expression#1;
  is_variable = true;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE adder#1;
  addends = ?addends;
  sum = expression#1;
  extracted_from = {equation#1};
  covers = equation#1;
END_INSTANCE;
```

E.1.11 adder from $x = z - y$

```
(***********************************)
(* adder-minus.filc 2000:11:16 13:49 *)
(***********************************)
(* EQUATION                        *)
(*      ?x = ?z - ?y               *)
(***********************************)
(* ADDER                           *)
(*      addends = {?x,?y}          *)
(*      sum = ?z                   *)
(*      extracted_from = EQUATION  *)
(*      covers = EQUATION          *)
(***********************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting an adder from an equation of the form x=z-y";
  problem = equation#1;
  solution = adder#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE equation#1;
  equal_expressions = {addition#1, expression#1};
END_INSTANCE;

INSTANCE addition#1;
  addends = {?z, minus#1};
END_INSTANCE;

INSTANCE minus#1;
  operand = ?y;
END_INSTANCE;

INSTANCE expression#1;
  is_variable = true;
END_INSTANCE;

(************)
```

```
(* Solution *)
(************)

INSTANCE adder#1;
  addends = {expression#1, ?y};
  sum = ?z;
  extracted_from = {equation#1};
  covers = equation#1;
END_INSTANCE;
```

E.1.12 multiplier from $x * y = z$

```
(*******************************************)
(* multiplier-multiply.filc 2000:11:16 13:57 *)
(*******************************************)
(* EQUATION                                *)
(*     ?x * ?y = ?z                        *)
(*******************************************)
(* MULTIPLIER                              *)
(*     multiplicands = {?x,?y}             *)
(*     product = ?z                        *)
(*     extracted_from = EQUATION           *)
(*     covers = EQUATION                   *)
(*******************************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting multiplier from an equation of form x*y=z";
  problem = equation#1;
  solution = multiplier#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE equation#1;
  equal_expressions = {multiplication#1, expression#1};
END_INSTANCE;

INSTANCE multiplication#1;
  multiplicands = ?multiplicands;
END_INSTANCE;

INSTANCE expression#1;
  is_variable = true;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE multiplier#1;
  multiplicands = ?multiplicands;
  product = expression#1;
  extracted_from = {equation#1};
  covers = equation#1;
END_INSTANCE;
```

E.1.13 multiplier from $x = z/y$

```
(****************************************)
(* multiplier-divide.filc 2000:11:16 13:55 *)
(****************************************)
(* EQUATION                             *)
(*      ?x = ?z / ?y                    *)
(****************************************)
(* MULTIPLIER                           *)
(*      multiplicands = {?x,?y}         *)
(*      product = ?z                    *)
(*      extracted_from = EQUATION       *)
(*      covers = EQUATION               *)
(****************************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting multiplier from an equation of form x=z/y";
  problem = equation#1;
  solution = multiplier#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE equation#1;
  equal_expressions = {multiplication#1, expression#1};
END_INSTANCE;

INSTANCE expression#1;
  is_variable = true;
END_INSTANCE;

INSTANCE multiplication#1;
  multiplicands = {?z, reciprocal#1};
END_INSTANCE;

INSTANCE reciprocal#1;
  operand = ?y;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE multiplier#1;
  multiplicands = {expression#1, ?y};
  product = ?z;
  extracted_from = {equation#1};
  covers = equation#1;
END_INSTANCE;
```

E.1.14 type_hlc from optional attribute

```
(****************************************)
(* type_hlc-optional.filc 1999:11:18 16:20 *)
(****************************************)
(* ENTITY e;                            *)
(*   a : OPTIONAL t;                    *)
(* END_ENTITY;                          *)
(****************************************)
```

```
(* TYPE_HLC                               *)
(*      extracted_from = a                *)
(*      attribute = a                     *)
(*      tipe = t                          *)
(****************************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting type_hlc from an optional attribute";
  problem = explicit_attribute#1;
  solution = type_hlc#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE explicit_attribute#1;
  is_optional = true;
  tipe = ?type;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE type_hlc#1;
  extracted_from = {explicit_attribute#1};
  attribute = explicit_attribute#1;
  tipe = ?type;
END_INSTANCE;
```

E.1.15 type_hlc and non_null from mandatory attribute

```
(*********************************************)
(* type_hlc+non_null-attr.filc 1999:11:18 16:19 *)
(*********************************************)
(* ENTITY e;                                 *)
(*   a : t;                                  *)
(* END_ENTITY;                               *)
(*********************************************)
(* TYPE_HLC                                  *)
(*      extracted_from = a                   *)
(*      attribute = a                        *)
(*      tipe = t                             *)
(*                                           *)
(* NON_NULL                                  *)
(*      extracted_from = a                   *)
(*      attribute = a                        *)
(*********************************************)

INSTANCE extraction_case#1;
  language = express;
  description =
  "A case for extracting type_hlc & non_null from a mandatory attribute";
  problem = explicit_attribute#1;
  solution = {type_hlc#1,non_null#1};
END_INSTANCE;

(***********)
(* Problem *)
(***********)
```

```
INSTANCE explicit_attribute#1;
  is_optional = false;
  tipe = ?type;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE type_hlc#1;
  extracted_from = {explicit_attribute#1};
  attribute = explicit_attribute#1;
  tipe = ?type;
END_INSTANCE;

INSTANCE non_null#1;
  extracted_from = {explicit_attribute#1};
  attribute = explicit_attribute#1;
END_INSTANCE;
```

E.1.16 non_null from EXISTS

```
(***************************************)
(* non_null-exists.filc 2000:11:09 16:38 *)
(***************************************)
(* ENTITY e;                           *)
(*   a : OPTIONAL t;                   *)
(* WHERE                               *)
(*   EXISTS(a);                        *)
(* END_ENTITY;                         *)
(***************************************)
(* NON_NULL                            *)
(*      attribute = a                  *)
(*      extracted_from = EXISTS(a)     *)
(*      covers = EXISTS(a)             *)
(***************************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting non_null from EXISTS";
  problem = function_call#1;
  solution = non_null#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE function_call#1;
  function_called = built_in_function#1;
  actual_parameters = [?attribute];
END_INSTANCE;

INSTANCE built_in_function#1;
  name = "EXISTS";
END_INSTANCE;

(************)
(* Solution *)
(************)
```

```
INSTANCE non_null#1;
  attribute = ?attribute;
  extracted_from = {function_call#1};
  covers = function_call#1;
END_INSTANCE;
```

E.1.17 optional_xor_empty from SET [0 : ?]

```
(***********************************************)
(* optional_xor_empty-empty.filc 1999:11:18 16:15 *)
(***********************************************)
(* a : AGGREGATION [0 : ?] OF t;              *)
(***********************************************)
(* OPTIONAL_XOR_EMPTY_AGGREGATION             *)
(*      attribute = a                         *)
(***********************************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting optional_xor_empty from [0:?]";
  problem = explicit_attribute#1;
  solution = optional_xor_empty_aggregation#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE explicit_attribute#1;
  tipe = aggregation_type#1;
  is_optional = false;
END_INSTANCE;

INSTANCE aggregation_type#1;
  lower_bound = integer_literal#1;
END_INSTANCE;

INSTANCE integer_literal#1;
  literal_value = 0;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE optional_xor_empty_aggregation#1;
  attribute = explicit_attribute#1;
  extracted_from = {explicit_attribute#1};
END_INSTANCE;
```

E.1.18 optional_xor_empty from OPTIONAL SET [1 : ?]

```
(**************************************************)
(* optional_xor_empty-optional.filc 1999:11:18 16:16 *)
(**************************************************)
(* a : OPTIONAL AGGREGATION [1 : ?] OF t;          *)
(**************************************************)
(* OPTIONAL_XOR_EMPTY_AGGREGATION                  *)
(*      attribute = a                              *)
(**************************************************)
```

```
INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting optional_xor_empty from OPTIONAL [1:?]";
  problem = explicit_attribute#1;
  solution = optional_xor_empty_aggregation#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE explicit_attribute#1;
  tipe = aggregation_type#1;
  is_optional = true;
END_INSTANCE;

INSTANCE aggregation_type#1;
  lower_bound = integer_literal#1;
END_INSTANCE;

INSTANCE integer_literal#1;
  literal_value = 1;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE optional_xor_empty_aggregation#1;
  attribute = explicit_attribute#1;
  extracted_from = {explicit_attribute#1};
END_INSTANCE;
```

E.1.19 one_to_one relationship

```
(************************************)
(* one-to-one.filc  1999:11:18 16:13 *)
(************************************)
(* ENTITY man;                      *)
(*   wife : woman;                  *)
(* END_ENTITY;                      *)
(*                                  *)
(* ENTITY woman;                    *)
(* INVERSE                          *)
(*   husband : man FOR wife;        *)
(* END_ENTITY;                      *)
(************************************)
(* ONE_TO_ONE_RELATIONSHIP          *)
(*      extracted_from = {man,woman} *)
(*      source_entity = man         *)
(*      source_attribute = wife     *)
(*      target_entity = woman       *)
(*      target_attribute = husband  *)
(************************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting one_to_one_relationship from two entities";
  problem = inverse_attribute#1;
  solution = one_to_one_relationship#1;
END_INSTANCE;
```

```
(***********)
(* Problem *)
(***********)

INSTANCE entity_type#1;
  explicit_attributes = {explicit_attribute#1};
END_INSTANCE;

INSTANCE explicit_attribute#1;
  tipe = entity_type#2;
  is_optional = false;
END_INSTANCE;

INSTANCE entity_type#2;
  inverse_attributes = {inverse_attribute#1};
END_INSTANCE;

INSTANCE inverse_attribute#1;
  tipe = entity_type#1;
  for_attribute = explicit_attribute#1;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE one_to_one_relationship#1;
  extracted_from = {entity_type#1,entity_type#2};
  source_entity = entity_type#1;
  source_attribute = explicit_attribute#1;
  target_entity = entity_type#2;
  target_attribute = inverse_attribute#1;
END_INSTANCE;
```

E.1.20 one_to_many relationship

```
(**********************************************************)
(* one-to-many.filc                      1999:11:18 16:13 *)
(**********************************************************)
(* ENTITY room;                                            *)
(*   guardian_instances : SET [0 : 8] OF guardian_instance; *)
(* END_ENTITY;                                             *)
(*                                                         *)
(* ENTITY guardian_instance;                               *)
(* INVERSE                                                 *)
(*   the_room : room FOR guardian_instances;               *)
(* END_ENTITY;                                             *)
(**********************************************************)
(* ONE_TO_MANY_RELATIONSHIP                                *)
(*      extracted_from = {room,guardian_instance}          *)
(*      source_entity = room                               *)
(*      source_attribute = guardian_instances              *)
(*      target_entity = guardian_instance                  *)
(*      target_attribute = the_room                        *)
(**********************************************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting one_to_many_relationship from two entities";
  problem = inverse_attribute#1;
  solution = one_to_many_relationship#1;
```

```
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE entity_type#1;
  explicit_attributes = {explicit_attribute#1};
END_INSTANCE;

INSTANCE explicit_attribute#1;
  tipe = aggregation_type#1;
  is_optional = false;
END_INSTANCE;

INSTANCE aggregation_type#1;
  element_type = entity_type#2;
END_INSTANCE;

INSTANCE entity_type#2;
  inverse_attributes = {inverse_attribute#1};
END_INSTANCE;

INSTANCE inverse_attribute#1;
  tipe = entity_type#1;
  for_attribute = explicit_attribute#1;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE one_to_many_relationship#1;
  extracted_from = {entity_type#1,entity_type#2};
  source_entity = entity_type#1;
  source_attribute = explicit_attribute#1;
  target_entity = entity_type#2;
  target_attribute = inverse_attribute#1;
END_INSTANCE;
```

E.1.21 one_to_many (many-to-one) relationship

```
(**********************************************************)
(* many-to-one.filc                      1999:11:18 16:09 *)
(**********************************************************)
(* ENTITY guardian_instance;                              *)
(*   class : guardian_class;                              *)
(* END_ENTITY;                                            *)
(*                                                        *)
(* ENTITY guardian_class;                                 *)
(* INVERSE                                                *)
(*   instances : SET [0 : ?] OF guardian_instance FOR class; *)
(* END_ENTITY;                                            *)
(**********************************************************)
(* ONE_TO_MANY_RELATIONSHIP                               *)
(*      extracted_from = {guardian_instance,guardian_class}  *)
(*      source_entity = guardian_class                    *)
(*      source_attribute = instances                      *)
(*      target_entity = guardian_instance                 *)
(*      target_attribute = class                          *)
(**********************************************************)
```

```
INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting many_to_one relationship from two entities";
  problem = inverse_attribute#1;
  solution = one_to_many_relationship#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE entity_type#1;
  explicit_attributes = {explicit_attribute#1};
END_INSTANCE;

INSTANCE explicit_attribute#1;
  tipe = entity_type#2;
  is_optional = false;
END_INSTANCE;

INSTANCE entity_type#2;
  inverse_attributes = {inverse_attribute#1};
END_INSTANCE;

INSTANCE inverse_attribute#1;
  tipe = aggregation_type#1;
  for_attribute = explicit_attribute#1;
END_INSTANCE;

INSTANCE aggregation_type#1;
  element_type = entity_type#1;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE one_to_many_relationship#1;
  extracted_from = {entity_type#1,entity_type#2};
  source_entity = entity_type#2;
  source_attribute = inverse_attribute#1;
  target_entity = entity_type#1;
  target_attribute = explicit_attribute#1;
END_INSTANCE;
```

E.1.22 many_to_many relationship

```
(**************************************************)
(* many-to-many.filc              1999:11:18 16:08 *)
(**************************************************)
(* ENTITY student;                                *)
(*   courses : SET [1 : ?] OF course;             *)
(* END_ENTITY;                                    *)
(*                                                *)
(* ENTITY course;                                 *)
(* INVERSE                                        *)
(*   students : SET [0 : ?] OF student FOR courses; *)
(* END_ENTITY;                                    *)
(**************************************************)
(* MANY_TO_MANY_RELATIONSHIP                      *)
(*      extracted_from = {student,course}         *)
(*      source_entity = student                   *)
```

```
(*      source_attribute = courses                 *)
(*      target_entity = course                     *)
(*      target_attribute = students                *)
(*************************************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting many_to_many relationship from two entities";
  problem = inverse_attribute#1;
  solution = many_to_many_relationship#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE entity_type#1;
  explicit_attributes = {explicit_attribute#1};
END_INSTANCE;

INSTANCE explicit_attribute#1;
  tipe = aggregation_type#1;
  is_optional = false;
END_INSTANCE;

INSTANCE aggregation_type#1;
  element_type = entity_type#2;
END_INSTANCE;

INSTANCE entity_type#2;
  inverse_attributes = {inverse_attribute#1};
END_INSTANCE;

INSTANCE inverse_attribute#1;
  tipe = aggregation_type#2;
  for_attribute = explicit_attribute#1;
END_INSTANCE;

INSTANCE aggregation_type#2;
  element_type = entity_type#1;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE many_to_many_relationship#1;
  extracted_from = {entity_type#1,entity_type#2};
  source_entity = entity_type#1;
  source_attribute = explicit_attribute#1;
  target_entity = entity_type#2;
  target_attribute = inverse_attribute#1;
END_INSTANCE;
```

E.1.23 for_all

```
(**********************************)
(* for_all.filc    2000:11:09 16:00 *)
(**********************************)
(* EQUATION                       *)
(*      QUERY (x <* s | p(x)) = s   *)
(**********************************)
```

```
(* FOR_ALL                          *)
(*      bound_variable = x          *)
(*      aggregation = s             *)
(*      proposition = p             *)
(*      extracted_from = {EQUATION} *)
(*      covers = EQUATION           *)
(***********************************)

INSTANCE extraction_case#1;
  language = express;
  description = "A case for extracting for_all from QUERY(x <* s | p(x)) = s";
  problem = equation#1;
  solution = for_all#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE equation#1;
  equal_expressions = {query_expression#1, ?s};
END_INSTANCE;

INSTANCE query_expression#1;
  variable = ?x;
  aggregation_expression = ?s;
  logical_expression = ?p;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE for_all#1;
  bound_variable = ?x;
  aggregation = ?s;
  proposition = ?p;
  extracted_from = {equation#1};
  covers = equation#1;
END_INSTANCE;
```

E.1.24 there_exists

```
(*****************************************)
(* there_exists.filc      2000:11:09 17:00 *)
(*****************************************)
(* COMPARATOR                              *)
(*      SIZEOF(QUERY (x <* s | p(x))) >= 1 *)
(*****************************************)
(* THERE_EXISTS                            *)
(*      bound_variable = x                 *)
(*      aggregation = s                    *)
(*      proposition = p                    *)
(*      extracted_from = {COMPARATOR}      *)
(*      covers = COMPARATOR                *)
(*****************************************)

INSTANCE extraction_case#1;
  language = express;
  description =
  "A case for extracting there_exists from SIZEOF(QUERY(x<*s|p(x)))>=1";
  problem = comparator#1;
```

```
  solution = there_exists#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE comparator#1;
  greater = function_call#1;
  lesser = integer_literal#1;
  inclusive = inclusive_or_exclusive(inclusive);
END_INSTANCE;

INSTANCE function_call#1;
  function_called = built_in_function#1;
  actual_parameters = [query_expression#1];
END_INSTANCE;

INSTANCE built_in_function#1;
  name = "SIZEOF";
END_INSTANCE;

INSTANCE query_expression#1;
  variable = ?x;
  aggregation_expression = ?s;
  logical_expression = ?p;
END_INSTANCE;

INSTANCE integer_literal#1;
  literal_value = 1;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE there_exists#1;
  bound_variable = ?x;
  aggregation = ?s;
  proposition = ?p;
  extracted_from = {comparator#1};
  covers = comparator#1;
END_INSTANCE;
```

E.1.25 domain_rules_hlc from an entity

```
(*************************************************)
(* domain_rules_hlc.filc          2000:12:05 10:47 *)
(*************************************************)
(* ENTITY e;                                      *)
(* WHERE                                          *)
(*     dr                                         *)
(* END_ENTITY;                                    *)
(*************************************************)
(* DOMAIN_RULES_HLC                               *)
(*     extracted_from = {e}                       *)
(*     entity = e                                 *)
(*     domain_rules = dr                          *)
(*************************************************)

INSTANCE extraction_case#1;
  language = express;
```

```
  description="A case for extracting a domain_rules_hlc from an entity";
  problem = entity_type#1;
  solution = {domain_rules_hlc#1};
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE entity_type#1;
  domain_rules = ?dr;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE domain_rules_hlc#1;
  extracted_from = {entity_type#1};
  the_entity = entity_type#1;
  domain_rules = ?dr;
END_INSTANCE;
```

E.1.26 domain_rules_hlc propagated from a defined type to an entity

```
(**************************************************)
(* propagate_from_defined_type.filc 2000:11:30 14:24 *)
(**************************************************)
(* ENTITY e;                                        *)
(*   a : dt;                                        *)
(* END_ENTITY;                                      *)
(*                                                  *)
(* TYPE dt = ut;                                    *)
(* WHERE dr;                                        *)
(* END_TYPE;                                        *)
(**************************************************)
(* TYPE_HLC                                         *)
(*      extracted_from = {dt}                       *)
(*      attribute = a                               *)
(*      tipe = ut                                   *)
(**************************************************)
(* DOMAIN_RULES_HLC                                 *)
(*      extracted_from = {dt}                       *)
(*      entity = e                                  *)
(*      substitute_for_self = a                     *)
(*      domain_rules = dr                           *)
(**************************************************)

INSTANCE extraction_case#1;
  language = express;
  description =
  "A case for propagating domain rules from defined types to entities";
  problem = explicit_attribute#1;
  solution = {type_hlc#1, domain_rules_hlc#1};
END_INSTANCE;

(***********)
(* Problem *)
(***********)
```

```
INSTANCE explicit_attribute#1;
  tipe = defined_type#1;
  the_entity = entity_type#1;
END_INSTANCE;

INSTANCE entity_type#1;
END_INSTANCE;

INSTANCE defined_type#1;
  tipe = ?ut;
  domain_rules = ?dr;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE type_hlc#1;
  extracted_from = {defined_type#1};
  attribute = explicit_attribute#1;
  tipe = ?ut;
END_INSTANCE;

INSTANCE domain_rules_hlc#1;
  extracted_from = {defined_type#1};
  the_entity = entity_type#1;
  substitute_for_self = explicit_attribute#1;
  domain_rules = ?dr;
END_INSTANCE;
```

E.2 Navigation Cases

Navigation cases are discussed in Section 6.3.1.

E.2.1 Navigating a model

```
(*********************************************)
(* model.filc               1999:11:18 15:53 *)
(*********************************************)
(* Meta case for navigating an EXPRESS model. *)
(*********************************************)

INSTANCE navigation_case#1;
  language = express;
  description = "A case for navigating an EXPRESS model";
  problem = model_frame#1;
  solution = submit#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE model_frame#1;
  schemata = ?s;
END_INSTANCE;

(************)
```

```
(* Solution *)
(************)

INSTANCE submit#1;
  frames = {?s};
END_INSTANCE;
```

E.2.2 Navigating a schema

```
(**************************************)
(* schema.filc       1999:11:18 15:53 *)
(**************************************)
(* Meta case for navigating a schema. *)
(**************************************)

INSTANCE navigation_case#1;
  language = express;
  description = "A case for navigating a schema";
  problem = schema_frame#1;
  solution = submit#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE schema_frame#1;
  entities = ?e;
  defined_types = ?t;
  rules = ?r;
  functions = ?f;
  procedures = ?p;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE submit#1;
  frames = {?e,?t,?r,?f,?p};
END_INSTANCE;
```

E.2.3 Navigating a defined type

```
(********************************************)
(* defined_type.filc       2000:06:13 17:07 *)
(********************************************)
(* Meta case for navigating a defined type. *)
(********************************************)

INSTANCE navigation_case#1;
  language = express;
  description = "A case for navigating a defined type";
  problem = defined_type#1;
  solution = submit#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)
```

```
INSTANCE defined_type#1;
  domain_rules = ?w;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE submit#1;
  frames = {?w};
END_INSTANCE;
```

E.2.4 Navigating an entity

```
(*************************************)
(* entity.filc       2000:06:14 16:23 *)
(*************************************)
(* Meta case for navigating an entity. *)
(*************************************)

INSTANCE navigation_case#1;
  language = express;
  description = "A case for navigating an entity";
  problem = entity_type#1;
  solution = submit#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE entity_type#1;
  supertype_of = ?se;
  subtype_of = ?s;
  explicit_attributes = ?e;
  explicit_attribute_redeclarations = ?ear;
  derived_attributes = ?d;
  derived_attribute_redeclarations = ?dar;
  inverse_attributes = ?i;
  inverse_attribute_redeclarations = ?iar;
  unique_rules = ?u;
  domain_rules = ?w;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE submit#1;
  frames = {?se,?s,?e,?ear,?d,?dar,?i,?iar,?u,?w};
END_INSTANCE;
```

E.2.5 Navigating a domain rule

```
(*****************************************)
(* domain_rule.filc      1999:11:18 15:51 *)
(*****************************************)
(* Meta case for navigating a domain rule. *)
(*****************************************)
```

```
INSTANCE navigation_case#1;
  language = express;
  description = "A case for navigating a domain rule";
  problem = domain_rule#1;
  solution = submit#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE domain_rule#1;
  expression = ?e;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE submit#1;
  frames = {?e};
END_INSTANCE;
```

E.2.6 Navigating a conjunction

```
(*****************************************)
(* conjunction.filc       1999:11:18 15:51 *)
(*****************************************)
(* Meta case for navigating a conjunction. *)
(*****************************************)

INSTANCE navigation_case#1;
  language = express;
  description = "A case for navigating a conjunction";
  problem = conjunction#1;
  solution = submit#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE conjunction#1;
  conjuncts = ?c;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE submit#1;
  frames = {?c};
END_INSTANCE;
```

E.2.7 Navigating an equation

```
(***************************************)
(* equation.filc       2000:06:13 16:16 *)
(***************************************)
```

```
(* Meta case for navigating an equation. *)
(*****************************************)

INSTANCE navigation_case#1;
  language = express;
  description = "A case for navigating an equation";
  problem = equation#1;
  solution = submit#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE equation#1;
  equal_expressions = ?e;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE submit#1;
  frames = {?e};
END_INSTANCE;
```

E.2.8 Navigating a negation

```
(****************************************)
(* negation.filc      2000:02:02 16:19 *)
(****************************************)
(* Meta case for navigating a negation. *)
(****************************************)

INSTANCE navigation_case#1;
  language = express;
  description = "A case for navigating a negation";
  problem = negation#1;
  solution = submit#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE negation#1;
  operand = ?o;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE submit#1;
  frames = {?o};
END_INSTANCE;
```

E.2.9 Navigating a QUERY

```
(************************************************)
```

```
(* query.filc                 2000:06:12 17:47 *)
(**********************************************)
(* Meta case for navigating a query expression. *)
(**********************************************)

INSTANCE navigation_case#1;
  language = express;
  description = "A case for navigating a query expression";
  problem = query_expression#1;
  solution = submit#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE query_expression#1;
  logical_expression = ?e;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE submit#1;
  frames = {?e};
END_INSTANCE;
```

E.2.10 Navigating a user-defined function

```
(***************************************************)
(* function.filc                    2000:06:13 17:14 *)
(***************************************************)
(* Meta case for navigating a user-defined function. *)
(***************************************************)

INSTANCE navigation_case#1;
  language = express;
  description = "A case for navigating a user-defined function";
  problem = user_defined_function#1;
  solution = submit#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE user_defined_function#1;
  body = ?b;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE submit#1;
  frames = {?b};
END_INSTANCE;
```

E.2.11 Navigating a return statement

```
(**********************************************)
(* return.filc                2000:06:13 17:15 *)
(**********************************************)
(* Meta case for navigating a return statement. *)
(**********************************************)

INSTANCE navigation_case#1;
  language = express;
  description = "A case for navigating a return statement";
  problem = return_statement#1;
  solution = submit#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE return_statement#1;
  return_value = ?r;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE submit#1;
  frames = {?r};
END_INSTANCE;
```

E.3 Semantic-Equivalence/Strength Cases

Semantic equivalence/strength cases are discussed in Section 5.3.

E.3.1 Positive comparators

```
(*************************************************************************)
(* semantic-comparator-positive.filc                    2001:04:26 10:47 *)
(*************************************************************************)
(* A case for assessing the semantic equivalence of two comparators for   *)
(* which the value of the limit positively affects the semantic strength. *)
(*************************************************************************)

INSTANCE semantic_case#1;
  language = express;
  description =
  "A case for assessing the semantic equivalence of two comparators";
  problem = comparator#1;
  solution = semantic_match_prescription#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE comparator#1;
  greater = expression#1;
  lesser = expression#2;
```

```
END_INSTANCE;

INSTANCE expression#1;
  is_variable = true;
END_INSTANCE;

INSTANCE expression#2;
  is_variable = false;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE semantic_match_prescription#1;
  necessary_slots = {"greater", "inclusive"};
  semantic_strength_slots = {semantic_strength_slot#1};
END_INSTANCE;

INSTANCE semantic_strength_slot#1;
  slot_name = "lesser";
  influence = semantic_strength_influence(positive);
END_INSTANCE;
```

E.3.2 Negative comparators

```
(***************************************************************************)
(* semantic-comparator-negative.filc                     2000:05:05 11:31 *)
(***************************************************************************)
(* A case for assessing the semantic equivalence of two comparators for   *)
(* which the value of the limit negatively affects the semantic strength. *)
(***************************************************************************)

INSTANCE semantic_case#1;
  language = express;
  description =
  "A case for assessing the semantic equivalence of two comparators";
  problem = comparator#1;
  solution = semantic_match_prescription#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE comparator#1;
  greater = expression#1;
  lesser = expression#2;
END_INSTANCE;

INSTANCE expression#1;
  is_variable = false;
END_INSTANCE;

INSTANCE expression#2;
  is_variable = true;
END_INSTANCE;

(************)
(* Solution *)
(************)
```

```
INSTANCE semantic_match_prescription#1;
  necessary_slots = {"lesser", "inclusive"};
  semantic_strength_slots = {semantic_strength_slot#1};
END_INSTANCE;

INSTANCE semantic_strength_slot#1;
  slot_name = "greater";
  influence = semantic_strength_influence(negative);
END_INSTANCE;
```

E.3.3 Conjunctions

```
(*******************************************************************)
(* semantic-conjunction.filc                    2000:12:13 16:32 *)
(*******************************************************************)
(* A case for assessing the semantic equivalence of two conjunctions. *)
(*******************************************************************)

INSTANCE semantic_case#1;
  language = express;
  description =
  "A case for assessing the semantic equivalence of two conjunctions";
  problem = conjunction#1;
  solution = semantic_match_prescription#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE conjunction#1;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE semantic_match_prescription#1;
  necessary_slots = {};
  semantic_strength_slots = {semantic_strength_slot#1};
END_INSTANCE;

INSTANCE semantic_strength_slot#1;
  slot_name = "conjuncts";
  influence = semantic_strength_influence(positive);
END_INSTANCE;
```

E.3.4 Disjunctions

```
(*******************************************************************)
(* semantic-disjunction.filc                    2000:12:13 16:33 *)
(*******************************************************************)
(* A case for assessing the semantic equivalence of two disjunctions. *)
(*******************************************************************)

INSTANCE semantic_case#1;
  language = express;
  description =
  "A case for assessing the semantic equivalence of two disjunctions";
  problem = inclusive_disjunction#1;
```

```
  solution = semantic_match_prescription#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE inclusive_disjunction#1;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE semantic_match_prescription#1;
  necessary_slots = {};
  semantic_strength_slots = {semantic_strength_slot#1};
END_INSTANCE;

INSTANCE semantic_strength_slot#1;
  slot_name = "disjuncts";
  influence = semantic_strength_influence(negative);
END_INSTANCE;
```

E.3.5 Lower bounds of unordered aggregation-types

```
(**************************************************************)
(* semantic-uagg-lower.filc                2000:08:08 14:33 *)
(**************************************************************)
(* A case for assessing the semantic equivalence of two     *)
(* unordered aggregation types based on their lower bounds. *)
(**************************************************************)

INSTANCE semantic_case#1;
  language = express;
  description =
  "A case for assessing the sem.eq. of two unordered aggregation types";
  problem = unordered_aggregation_type#1;
  solution = semantic_match_prescription#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE unordered_aggregation_type#1;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE semantic_match_prescription#1;
  necessary_slots = {};
  semantic_strength_slots = {semantic_strength_slot#1};
END_INSTANCE;

INSTANCE semantic_strength_slot#1;
  slot_name = "lower_bound";
  influence = semantic_strength_influence(positive);
END_INSTANCE;
```

E.3.6 Upper bounds of unordered aggregation-types

```
(************************************************************)
(* semantic-uagg-upper.filc                2000:08:08 14:33 *)
(************************************************************)
(* A case for assessing the semantic equivalence of two     *)
(* unordered aggregation types based on their upper bounds. *)
(************************************************************)

INSTANCE semantic_case#1;
  language = express;
  description =
  "A case for assessing the sem.eq. of two unordered aggregation types";
  problem = unordered_aggregation_type#1;
  solution = semantic_match_prescription#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE unordered_aggregation_type#1;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE semantic_match_prescription#1;
  necessary_slots = {};
  semantic_strength_slots = {semantic_strength_slot#1};
END_INSTANCE;

INSTANCE semantic_strength_slot#1;
  slot_name = "upper_bound";
  influence = semantic_strength_influence(negative);
END_INSTANCE;
```

E.3.7 Lower bounds of list-types

```
(*******************************************************************)
(* semantic-list-lower.filc                       2000:08:08 14:33 *)
(*******************************************************************)
(* A case for assessing the semantic equivalence of two list types *)
(* based on their lower bounds.                                    *)
(*******************************************************************)

INSTANCE semantic_case#1;
  language = express;
  description =
  "A case for assessing the semantic equivalence of two list types";
  problem = list_type#1;
  solution = semantic_match_prescription#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE list_type#1;
END_INSTANCE;
```

```
(************)
(* Solution *)
(************)

INSTANCE semantic_match_prescription#1;
  necessary_slots = {};
  semantic_strength_slots = {semantic_strength_slot#1};
END_INSTANCE;

INSTANCE semantic_strength_slot#1;
  slot_name = "lower_bound";
  influence = semantic_strength_influence(positive);
END_INSTANCE;
```

E.3.8 Upper bounds of list-types

```
(*****************************************************************)
(* semantic-list-upper.filc                       2000:08:08 14:33 *)
(*****************************************************************)
(* A case for assessing the semantic equivalence of two list types *)
(* based on their upper bounds.                                    *)
(*****************************************************************)

INSTANCE semantic_case#1;
  language = express;
  description =
  "A case for assessing the semantic equivalence of two list types";
  problem = list_type#1;
  solution = semantic_match_prescription#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE list_type#1;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE semantic_match_prescription#1;
  necessary_slots = {};
  semantic_strength_slots = {semantic_strength_slot#1};
END_INSTANCE;

INSTANCE semantic_strength_slot#1;
  slot_name = "upper_bound";
  influence = semantic_strength_influence(negative);
END_INSTANCE;
```

E.3.9 UNIQUE rules

```
(********************************************************************)
(* semantic-unique.filc                              2001:09:18 10:35 *)
(********************************************************************)
(* A case for assessing the semantic equivalence of two UNIQUE rules. *)
(********************************************************************)
```

```
INSTANCE semantic_case#1;
  language = express;
  description =
  "A case for assessing the semantic equivalence of two UNIQUE rules";
  problem = unique_rule#1;
  solution = semantic_match_prescription#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE unique_rule#1;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE semantic_match_prescription#1;
  necessary_slots = {};
  semantic_strength_slots = {semantic_strength_slot#1};
END_INSTANCE;

INSTANCE semantic_strength_slot#1;
  slot_name = "attributes";
  influence = semantic_strength_influence(negative);
END_INSTANCE;
```

E.3.10 choice_set HLCs

```
(**************************************************************************)
(* semantic-choice_set.filc                             2002:05:08 18:50 *)
(**************************************************************************)
(* A case for assessing the semantic equivalence of two choice_set HLCs. *)
(**************************************************************************)

INSTANCE semantic_case#1;
  language = express;
  description =
  "A case for assessing the semantic equivalence of two choice_sets";
  problem = choice_set#1;
  solution = semantic_match_prescription#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE choice_set#1;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE semantic_match_prescription#1;
  necessary_slots = {"constrained_entity"};
  semantic_strength_slots = {semantic_strength_slot#1,
                             semantic_strength_slot#2};
END_INSTANCE;
```

```
INSTANCE semantic_strength_slot#1;
  slot_name = "is_optional";
  influence = semantic_strength_influence(negative);
END_INSTANCE;

INSTANCE semantic_strength_slot#2;
  slot_name = "choice_set";
  influence = semantic_strength_influence(negative);
END_INSTANCE;
```

E.4 Subexpression Cases

Subexpression cases are discussed in Section 4.3.

E.4.1 Conjunction

```
(***************************************************)
(* subexpression-conjunction.filc   2000:04:07 16:03 *)
(***************************************************)
(* When matching a frame f to a conjunction frame c, *)
(* match f to all the conjuncts of c.               *)
(***************************************************)

INSTANCE subexpression_case#1;
  language = express;
  description = "A case for walking the conjuncts of a conjunction";
  problem = conjunction#1;
  solution = submit#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE conjunction#1;
  conjuncts = ?c;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE submit#1;
  frames = {?c};
END_INSTANCE;
```

E.4.2 Disjunction

```
(***************************************************)
(* subexpression-disjunction.filc   2000:04:07 16:03 *)
(***************************************************)
(* When matching a frame f to a disjunction frame d, *)
(* match f to all the disjuncts of d.               *)
(***************************************************)

INSTANCE subexpression_case#1;
  language = express;
```

```
  description = "A case for walking the disjuncts of a disjunction";
  problem = disjunction#1;
  solution = submit#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE disjunction#1;
  disjuncts = ?d;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE submit#1;
  frames = {?d};
END_INSTANCE;
```

E.5 Pretty-Print Cases

Pretty-print cases are used to substitute textual descriptions for frame-references in the CCUS output. For example, `explicit_attribute#293` would appear as "`person.age`", `integer_literal#97` would appear as "`0`", `comparator#8` would appear as "`(age >= 0)`", and `bound#9` would appear as "`a discrete, inclusive, lower bound of 0 on person.age`". The pretty-print case used for the last example is shown below.

```
(**********************************)
(* pp-bound.filc  2002:05:11 19:56 *)
(**********************************)
(* BOUND                          *)
(*      kind = ?kind              *)
(*      variable = ?variable      *)
(*      limit = ?limit            *)
(*      inclusive = ?inclusive    *)
(*      discrete = ?discrete      *)
(**********************************)
(* "a ?discrete, ?inclusive, ?kind *)
(*  bound of ?limit on ?variable"  *)
(**********************************)

INSTANCE pretty_print_case#1;
language = express;
description = "A case for pretty-printing a bound HLC";
problem = bound#1;
solution = pretty_print#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE bound#1;
```

```
kind = ?kind;
variable = ?variable;
limit = ?limit;
inclusive = ?inclusive;
discrete = ?discrete;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE pretty_print#1;
text = ["a ",
        ?discrete,
        ", ",
        ?inclusive,
        ", ",
        ?kind,
        " bound of ",
        ?limit,
        " on ",
        ?variable];
END_INSTANCE;
```

There are sixteen pretty-print cases for code-level frame-classes, and thirteen for HLC-classes, which can all be found on the Author's website [Bro].

Appendix F

Experimental Results

This appendix presents the results of a set of experiments devised to exercise all major aspects of the functionality of the Comparative Code-Understanding System (CCUS).

Each pair of models is run with various features enabled and disabled, to show how they affect the capabilities (inferring correspondences and reporting differences) and performance (comparison effort in particular) of the CCUS. Each experiment runs the CCUS first with all features enabled, then with one feature at a time disabled (so that the number of CCUS runs is linear in the number of features, rather than exponential, since there are 2^n possible ways to enable or disable n features). The defaults are:

- Thesauri (Section 4.2.4), subexpression cases (Section 4.3), semantic equivalence/strength (SE/S) cases (Section 5.3) and the extraction of higher-level constraints (HLCs) (Section 6.3) are enabled by default.

- The default remembrance heuristic is unselective remembrance (Section 7.2.2), since this compares the most HLCs and hence infers the most correspondences.

- Extraction prompts are submitted separately (Section 6.3.1) by default. It was considered to use submitting extraction prompts together as the default because this extracts more HLCs, hence a greater number of abstract correspondences (Section 6.3.3) may be inferred in some circumstances. However, this slight increase in capability is offset by the extraction of many extraneous HLCs (Section 6.3.1) and, consequently, a quadratic explosion in the number of HLC-comparisons.

Thus, for each pair of models, the CCUS is run in each of the following modes:

- E: all features enabled;
- -T: thesauri disabled;
- -U: subexpression cases disabled;
- -S: SE/S cases disabled;
- -H: extraction of HLCs disabled;
- -R: no remembrance (Section 7.2.1);
- M: the monogamy remembrance heuristic (Section 7.2.3);
- X: submitting extraction prompts together (Section 6.3.1).

For each pair of models, a table is compiled, showing, for each of the above modes, the following measurements:

- the number of HLCs extracted from the first model;
- the number of HLCs extracted from the second model;
- the total number of comparisons performed (measured as slot-filler comparisons including frames and aggregation elements) – a measure of comparison effort (fewer comparisons mean greater efficiency);
- the number of HLC-comparisons performed (a subset of the total number of comparisons, in which the fillers being compared are HLC frames);
- the total number of correspondences inferred (not including retracted plausible correspondences) – a measure of effectiveness;
- the number of HLC-correspondences inferred (a subset of the total number of correspondences inferred);
- the number of differences found (when two frames have non-matching values for the same slot) – another important part of the HLC CCUS's effectiveness along with inferring correspondences;
- the number of frame-class-thesaurus-lookups (with thesauri enabled, each failure of frame-class identicality (Section 4.2.2) incurs a thesaurus-lookup);

- the number of name-thesaurus-lookups (with thesauri enabled, each failure of name identicality (Section 4.2.3) incurs a thesaurus-lookup).

The following measurements are given once for each pair of models:

- the number of frames in the frame-network representing each model (a measure of size);
- the number of HLCs extracted from each model unselectively by the stand-alone CUS [Bro99] (a basis for comparison for selective extraction – Section 7.1).

The following frame-class thesaurus is used for all experiments:

```
{entitytype, definedtype, enumerationsymbol,}
```

The following name-thesaurus is used for each experiment unless otherwise stated:

```
{male, man,}
{female, woman,}
```

The detailed CCUS outputs and frame-dumps are on the Author's website [Bro].

Sections F.1 to F.12 pertain to artificial models. Pairs of 'real' models are compared in Section F.13 (comparing the EDIF and CFI DR models) and in Section F.14 (comparing the Gerber and SVG models). Section F.13 presents a more detailed interpretation of the results than the other experiments. Section F.15 summarises the conclusions of the experiments.

F.1 Optional/Mandatory Attribute

The models in Figure F.1 are a trivial example of almost identical models with a single difference, to exercise the basic CCUS frame-comparison algorithm (Section 3.3).

- First model: represented by 8 frames, the CUS extracts 5 HLCs;
- Second model: represented by 8 frames, the CUS extracts 6 HLCs.

`SCHEMA test;`
`  ENTITY person;`
`    name : STRING;`
`    age : OPTIONAL INTEGER;`
`  WHERE`
`    age >= 0;`
`  END_ENTITY;`
`END_SCHEMA;`
	`SCHEMA test;`
	`  ENTITY person;`
	`    name : STRING;`
	`    age : INTEGER;`
	`  WHERE`
	`    age >= 0;`
	`  END_ENTITY;`
	`END_SCHEMA;`

Figure F.1: Two EXPRESS models with optional and mandatory attributes.

Table F.1 shows the results of running the CCUS on these models in the eight modes explained at the start of this appendix. In all modes, it reports one difference: that `age` is optional in the first model but mandatory in the second.

The HLCs extracted are a `type_hlc` on `age` in the first model, and a `type_hlc` and `non_null` on `age` in the second model. All these HLCs are extraneous because there are no abstract correspondences to be inferred – the semantics of the constraints can be compared totally at the code level in this example. The HLCs were extracted because the CCUS encountered a difference in the `is_optional` slot for the two `age` attributes, so it submitted the `age` attributes as extraction prompts, extracting all the HLCs it could from those attributes.

The nine correspondences inferred in all but Mode -H are:

- between the models;
- between the schemata;
- between the entities;
- between the `name` attributes;
- between the `age` attributes;
- between the domain rules;
- between the top-level expressions of the domain rules (`age >= 0`);
- between the integer literals (`0`);
- between the `type_hlc`s.

Mode	E	-T	-U	-S	-H	-R	M	X
HLCs extracted from first model	1	1	1	1	0	1	1	1
HLCs extracted from second model	2	2	2	2	0	2	2	2
Comparisons	41	41	41	41	37	41	41	41
HLC-comparisons	1	1	1	1	0	1	1	1
Correspondences	9	9	9	9	8	9	9	9
HLC-correspondences	1	1	1	1	0	1	1	1
Differences	1	1	1	1	1	1	1	1
FC-thesaurus lookups	0	0	0	0	0	0	0	0
Name-thesaurus lookups	0	0	0	0	0	0	0	0

Table F.1: Experimental results for the models in Section F.1.

Whether the thesauri, subexpression cases and SE/S cases are enabled, and whether extraction prompts are submitted separately or together, make no difference for this simple example (except for the cost of loading the thesauri at the start, and a bit of unnecessary case-retrieval effort).

Not extracting HLCs at all (Mode -H) reduces the comparison effort since there are no HLCs to be compared, and hence infers one fewer correspondence (namely the one between the two `type_hlc`s).

F.2 Aligning an Expression with a Subexpression

The models in Figure F.2 are an example of comparing an expression in one model with a corresponding subexpression in the other model.

- First model: represented by 8 frames, the CUS extracts 6 HLCs;
- Second model: represented by 11 frames, the CUS extracts 8 HLCs.

Table F.2 shows the results for comparing these models. In most of the CCUS runs, the three HLCs extracted from the first model (first row of Table F.2) are a `bound`, plus a `type_hlc` and `non_null` on `age`, which is submitted as an extraction prompt when `age >= 0` is compared unsuccessfully with `age < 130` in the second model (submitting `age` as an extraction prompt is needless, and could be avoided by representing an attribute-reference as a separate object rather than plugging in a reference to the attribute itself). The three HLCs extracted

<pre>SCHEMA test;
  ENTITY person;
    name : STRING;
    age : INTEGER;
  WHERE
    age >= 0;
  END_ENTITY;
END_SCHEMA;</pre>
	<pre>SCHEMA test;
	  ENTITY person;
	    name : STRING;
	    age : INTEGER;
	  WHERE
	    (age >= 0) AND (age < 130);
	  END_ENTITY;
	END_SCHEMA;</pre>

Figure F.2: Aligning an expression in one model with a subexpression in another.

Mode	E	-T	-U	-S	-H	-R	M	X
HLCs extracted from first model	3	3	1	3	0	3	3	3
HLCs extracted from second model	3	3	3	3	0	3	3	3
Comparisons	54	54	39	54	43	54	54	54
HLC-comparisons	3	3	3	3	0	3	3	3
Correspondences	8	8	8	8	7	8	8	8
HLC-correspondences	1	1	1	1	0	1	1	1
Differences	4	4	3	4	3	4	4	4
FC-thesaurus lookups	2	0	1	2	2	2	2	2
Name-thesaurus lookups	0	0	0	0	0	0	0	0

Table F.2: Experimental results for the models in Section F.2.

from the second model are two `bound`s and a `range` when the top-level expressions fail to correspond.

Disabling subexpression cases (Mode -U) causes the `type_hlc` and `non_null`s not to be extracted, since they are extracted in the course of comparing `age >= 0` with the comparators in the other model. Not surprisingly, the disabling of subexpression cases substantially reduces the comparison effort, but at the expense of missing the correspondence between `age >= 0` and `age >= 0`. However, when the two `bound`s are compared and inferred to correspond, the HLC CCUS infers an abstract correspondence between `age >= 0` and `age >= 0` anyway (cf. Section 6.3.3).

F.3 Semantic Equivalence

The models in Figure F.3 are a simple example of two semantically equivalent but syntactically different expressions.

`SCHEMA test;`
`  ENTITY person;`
`    name : STRING;`
`    age : INTEGER;`
`  WHERE`
`    age >= 0;`
`  END_ENTITY;`
`END_SCHEMA;`
	`SCHEMA test;`
	`  ENTITY person;`
	`    name : STRING;`
	`    age : INTEGER;`
	`  WHERE`
	`    0 <= age;`
	`  END_ENTITY;`
	`END_SCHEMA;`

Figure F.3: Two models with semantically equivalent domain rules.

Mode	E	-T	-U	-S	-H	-R	M	X
HLCs extracted from first model	0	0	0	1	0	0	0	0
HLCs extracted from second model	0	0	0	1	0	0	0	0
Comparisons	37	37	37	46	37	37	37	37
HLC-comparisons	0	0	0	1	0	0	0	0
Correspondences	8	8	8	7	8	8	8	8
HLC-correspondences	0	0	0	1	0	0	0	0
Differences	1	1	1	3	1	1	1	1
FC-thesaurus lookups	0	0	0	0	0	0	0	0
Name-thesaurus lookups	0	0	0	0	0	0	0	0

Table F.3: Experimental results for the models in Section F.3.

- First model: represented by 8 frames, the CUS extracts 6 HLCs;
- Second model: represented by 8 frames, the CUS extracts 6 HLCs.

Table F.3 shows the results for comparing these models. Only the disabling of SE/S cases (Mode -S) makes a difference here: without them, `age >= 0` and `0 <= age` fail to correspond because they differ in the slot `which_way_round`.

In the other modes, the CCUS resolves the difference using the SE/S case for comparators (Appendix E, Section E.3.1), which specifies (by omission) that `which_way_round` is an unnecessary slot for two comparators to be semantically equivalent. There is therefore no need for the CCUS to attempt subsequently to resolve the difference by extracting and comparing HLCs.

F.4 Semantic Strength

The models in Figure F.4 are a simple example of one expression (`age >= 18`) which is semantically stronger than another (`age >= 0`).

```
SCHEMA test;
  ENTITY person;
    name : STRING;
    age : INTEGER;
  WHERE
    age >= 0;
  END_ENTITY;
END_SCHEMA;
```

```
SCHEMA test;
  ENTITY person;
    name : STRING;
    age : INTEGER;
  WHERE
    age >= 18;
  END_ENTITY;
END_SCHEMA;
```

Figure F.4: The second domain rule is stronger than the first.

Mode	E	-T	-U	-S	-H	-R	M	X
HLCs extracted from first model	0	0	0	1	0	0	0	0
HLCs extracted from second model	0	0	0	1	0	0	0	0
Comparisons	34	34	34	34	34	34	34	34
HLC-comparisons	0	0	0	2	0	0	0	0
Correspondences	7	7	7	5	7	7	7	7
HLC-correspondences	0	0	0	0	0	0	0	0
Differences	2	2	2	5	2	2	2	2
FC-thesaurus lookups	0	0	0	0	0	0	0	0
Name-thesaurus lookups	0	0	0	0	0	0	0	0

Table F.4: Experimental results for the models in Section F.4.

- First model: represented by 8 frames, the CUS extracts 6 HLCs;
- Second model: represented by 8 frames, the CUS extracts 6 HLCs.

Table F.4 shows the results for comparing these models. Again, the only mode which makes a difference is disabling SE/S cases.

F.5 Mandatory Attribute/EXISTS

The models in Figure F.5 are an example of two ways of expressing the constraint that an attribute must have a value other than `indeterminate`, which can be shown to be equivalent by extracting and comparing `non_null` HLCs (Appendix A, Section A.4).

- First model: represented by 7 frames, the CUS extracts 5 HLCs;
- Second model: represented by 7 frames, the CUS extracts 5 HLCs.

`SCHEMA test;`
`  ENTITY person;`
`    name : STRING;`
`    age : OPTIONAL INTEGER;`
`  WHERE`
`    EXISTS(age);`
`  END_ENTITY;`
`END_SCHEMA;`
	`SCHEMA test;`
	`  ENTITY person;`
	`    name : STRING;`
	`    age : INTEGER;`
	`  END_ENTITY;`
	`END_SCHEMA;`

Figure F.5: Two ways of expressing that `age` must have a non-null value.

Mode	E	-T	-U	-S	-H	-R	M	X
HLCs extracted from first model	2	2	2	2	0	2	2	5
HLCs extracted from second model	2	2	2	2	0	2	2	5
Comparisons	27	27	27	27	21	25	27	47
HLC-comparisons	2	2	2	2	0	1	2	12
Correspondences	7	7	7	7	5	6	7	9
HLC-correspondences	2	2	2	2	0	1	2	4
Differences	2	2	2	2	2	2	2	7
FC-thesaurus lookups	0	0	0	0	0	0	0	0
Name-thesaurus lookups	0	0	0	0	0	0	0	4

Table F.5: Experimental results for the models in Section F.5.

Table F.5 shows the results for comparing these models. The HLC CCUS extracts a `non_null` HLC from each model and infers them to correspond – thus it has seen that `age` must be non-null in both models. It also extracts extraneous `type_hlc`s from both models (and infers them to correspond).

The differences that the choice of remembrance heuristic makes are discussed in Section 7.2.4, which includes a cut-down version of this experiment. The more selective remembrance heuristics (no remembrance is more selective than monogamy, and monogamy is more selective than unselective remembrance) reduce the number of comparisons, and can hence reduce the number of correspondences inferred.

Submitting extraction prompts together (Mode X – see Section 6.3.1) makes a big difference in this example. The number of comparisons explode quadratically – particularly in the number of HLCs (of a given frame-class) extracted, where the sets of HLCs to be compared can get excessively long. This is because, in Mode X, three further extraneous HLCs are extracted from each model (a `non_null` from `name`, a `type_hlc` from `name`, and a `domain_rules_hlc` from `person`). The `non_null`s are inferred to correspond, as are the `type_hlc`s, hence the two extra, extraneous correspondences.

F.6 Range Constraints

The models in Figure F.6 illustrate two ways of expressing a range constraint: as a conjunction of comparators and as an interval expression, which can be shown to be equivalent by extracting and comparing `range` HLCs (Appendix A, Section A.2).

- First model: represented by 11 frames, the CUS extracts 8 HLCs;
- Second model: represented by 9 frames, the CUS extracts 8 HLCs.

Table F.6 shows the results for comparing these models. In this example, the choice of remembrance heuristic does not affect the correspondences that the HLC CCUS is able to infer, because the HLCs (two `bounds` and a `range` from each model) are extracted in corresponding bouts of extraction (cf. Section 7.2), therefore there is nothing to be remembered.

<pre>SCHEMA test;
  ENTITY person;
    name : STRING;
    age : INTEGER;
  WHERE
    (age >= 0) AND (age < 130);
  END_ENTITY;
END_SCHEMA;</pre>
	<pre>SCHEMA test;
	  ENTITY person;
	    name : STRING;
	    age : INTEGER;
	  WHERE
	    {0 <= age < 130};
	  END_ENTITY;
	END_SCHEMA;</pre>

Figure F.6: Two ways of expressing a range constraint.

Mode	E	-T	-U	-S	-H	-R	M	X
HLCs extracted from first model	3	3	3	3	0	3	3	3
HLCs extracted from second model	3	3	3	3	0	3	3	3
Comparisons	59	59	57	59	26	59	59	59
HLC-comparisons	7	7	7	7	0	7	7	7
Correspondences	12	12	12	12	5	12	12	12
HLC-correspondences	3	3	3	3	0	3	3	3
Differences	3	3	3	3	2	3	3	3
FC-thesaurus lookups	3	0	1	3	3	3	3	3
Name-thesaurus lookups	0	0	0	0	0	0	0	0

Table F.6: Experimental results for the models in Section F.6.

SCHEMA test;
  ENTITY person;
    middle_names : LIST [0 : ?] OF STRING;
  END_ENTITY;
END_SCHEMA;
	SCHEMA test;
	  ENTITY person;
	    middle_names : OPTIONAL LIST [1 : ?] OF STRING;
	  END_ENTITY;
	END_SCHEMA;

Figure F.7: Two ways of modelling zero or more items.

Mode	E	-T	-U	-S	-H	-R	M	X
HLCs extracted from first model	3	3	3	3	0	3	3	3
HLCs extracted from second model	2	2	2	2	0	2	2	2
Comparisons	29	29	29	23	24	29	29	29
HLC-comparisons	2	2	2	3	0	2	2	2
Correspondences	8	8	8	5	6	8	8	8
HLC-correspondences	2	2	2	1	0	2	2	2
Differences	3	3	3	5	3	3	3	3
FC-thesaurus lookups	0	0	0	0	0	0	0	0
Name-thesaurus lookups	0	0	0	0	0	0	0	0

Table F.7: Experimental results for the models in Section F.7.

F.7 Mandatory One-or-More/Optional Zero-or-More

The models in Figure F.7 illustrate two ways of expressing that a list can have zero-or-more elements (Appendix A, Section A.4).

- First model: represented by 7 frames, the CUS extracts 4 HLCs;
- Second model: represented by 7 frames, the CUS extracts 3 HLCs.

Table F.7 shows the results for comparing these models.

F.8 Gender by Enumeration or Subtypes

The models in Figure F.8 illustrate two ways of expressing the constraint that a person must have exactly one of two genders (Appendix A, Section A.1). It infers this by extracting a `choice_set` HLC from each model and comparing them.

- First model: represented by 9 frames, the CUS extracts 10 HLCs;
- Second model: represented by 8 frames, the CUS extracts 8 HLCs.

<pre>SCHEMA test;
  ENTITY person;
    name : STRING;
    age : INTEGER;
    gender : gender_type;
  END_ENTITY;
 	
  TYPE gender_type = ENUMERATION OF (man, woman);
  END_TYPE;
END_SCHEMA;</pre>
	<pre>SCHEMA test;
	  ENTITY person ABSTRACT SUPERTYPE OF (ONEOF (man, woman));
	    name : STRING;
	    age : INTEGER;
	  END_ENTITY;
	
	  ENTITY man SUBTYPE OF (person);
	  END_ENTITY;
	
	  ENTITY woman SUBTYPE OF (person);
	  END_ENTITY;
	END_SCHEMA;</pre>

Figure F.8: A person must have exactly one of two genders.

Mode	E	-T	-U	-S	-H	-R	M	X
HLCs extracted from first model	10	10	10	10	0	10	10	10
HLCs extracted from second model	8	8	8	8	0	8	8	8
Comparisons	99	97	99	100	26	79	91	99
HLC-comparisons	33	34	33	34	0	17	25	33
Correspondences	14	11	14	13	5	14	14	14
HLC-correspondences	7	6	7	6	0	7	7	7
Differences	26	20	26	26	5	22	26	26
FC-thesaurus lookups	4	0	4	4	0	4	4	4
Name-thesaurus lookups	20	0	20	20	4	16	20	20

Table F.8: Experimental results for the models in Section F.8.

Table F.8 shows the results for comparing these models. The frame-class-thesaurus and the name-thesaurus come into play in this example: the CCUS uses both thesauri to sanction correspondence between `male` and `man` and between `female` and `woman`. The frame-class-thesaurus avoids vetoing correspondence between an enumeration symbol and an entity, and the name-thesaurus sanctions correspondence between them because their names are declared as synonyms. The number of thesaurus-lookups is a function of the number of comparisons, since every failure of the frame-class identicality and name identicality correspondence heuristics incurs a lookup in the appropriate thesaurus.

In this experiment, disabling SE/S cases (Mode -S) actually **increases** the comparison effort, because when the HLC CCUS fails to resolve a difference using SE/S cases, it attempts to do so by extracting and comparing HLCs.

F.9 Propagation of Constraints

The models in Figure F.9 illustrate the need to propagate a domain rule from a defined type (`age_type`) to an entity (`person` in the second model) through an attribute (`age`) in order to see that it is equivalent to a domain rule in the first model (Section 6.4).

- First model: represented by 8 frames, the CUS extracts 6 HLCs;
- Second model: represented by 10 frames, the CUS extracts 8 HLCs.

Table F.9 shows the results for comparing these models. It only works when submitting extraction prompts together. This is because `age` is of type `INTEGER` (represented by the enumeration symbol `integer_type`) in the first model, but of the defined type `age_type` (represented as a `defined_type` frame) in the second model.

- Submitting extraction prompts together (Section 6.3.1), the HLC CCUS submits both attributes and hence extracts a `domain_rule_hlc` from the second attribute (using the extraction case in Appendix E, Section E.1.26).
- Submitting extraction prompts separately, the HLC CCUS submits the `defined_type` frame instead of the second attribute, and hence misses extracting the `domain_rule_hlc`.

`SCHEMA propagation_of_constraints;`
`  ENTITY person;`
`    age : INTEGER;`
`  WHERE`
`    {0 <= age < 130};`
`  END_ENTITY;`
`END_SCHEMA;`
	`SCHEMA propagation_of_constraints;`
	`  ENTITY person;`
	`    age : age_type;`
	`  END_ENTITY;`
	
	`  TYPE age_type = INTEGER;`
	`  WHERE`
	`    {0 <= SELF < 130};`
	`  END_TYPE;`
	`END_SCHEMA;`

Figure F.9: Propagating a domain rule from a defined type to an entity.

Mode	E	-T	-U	-S	-H	-R	M	X
HLCs extracted from first model	5	5	5	5	0	5	5	6
HLCs extracted from second model	3	3	3	3	0	3	3	8
Comparisons	36	36	36	36	15	15	36	74
HLC-comparisons	11	11	11	11	0	0	11	18
Correspondences	4	4	4	4	4	4	4	11
HLC-correspondences	0	0	0	0	0	0	0	3
Differences	8	8	8	8	3	3	8	10
FC-thesaurus lookups	2	0	2	2	0	0	2	2
Name-thesaurus lookups	0	0	0	0	0	0	0	0

Table F.9: Experimental results for the models in Section F.9.

Mode	E	-T	-U	-S	-H	-R	M	X
HLCs from first model	48	50	48	48	0	48	48	50
HLCs from second model	55	57	55	55	0	55	55	57
Comparisons	2224	2255	2213	2163	344	825	934	2272
HLC-comparisons	1157	1239	1157	1167	0	218	299	1182
Correspondences	94	77	92	82	50	87	82	95
HLC-correspondences	42	36	42	39	0	37	32	43
Differences	614	638	614	621	28	172	209	632
FC-thesaurus lookups	2	0	2	2	2	2	2	2
Name-thesaurus lookups	664	0	664	624	55	223	261	681

Table F.10: Experimental results for the models in Section F.10.

F.10 Course World

The models in this section constitute a larger artificial model, and a revision of that model in which constraints are written in different ways such that the HLC CCUS can assess their semantic equivalence.

- First model (Section F.10.1): represented by 62 frames, the CUS extracts 50 HLCs;
- Second model (Section F.10.2): represented by 66 frames, the CUS extracts 57 HLCs.

The experiment in this section uses the following name-thesaurus:

```
{surname, familyname,}
{firstname, givenname,}
{module, course,}
```

The results in Table F.10 show that the quadratic explosion of correspondences, when the HLC CCUS is enabled, is quite severe even for small models such as these. The results show that monogamy (Mode M) vastly reduces the comparison effort compared with unselective remembrance (the default) – it is much closer to no remembrance (Mode -R) than to unselective remembrance in this regard. The main culprit, however, is the extraction of extraneous HLCs.

F.10.1 The First Model

```
SCHEMA course_world;

  (*******************************************************)
```

```
(* ENTITY person                                         *)
(*********************************************************)
(* Represents a living person, who must be either a man, *)
(* or a woman, but not both.                             *)
(*********************************************************)

ENTITY person ABSTRACT SUPERTYPE OF (ONEOF (man, woman));
  surname : STRING;
  first_name : STRING;
  middle_names : LIST [0 : ?] OF STRING;
  age : INTEGER;
  role : OPTIONAL university_role;
WHERE
  age_must_be_sensible:
    ((age >= 0) AND (age < 130));
END_ENTITY;

(****************************************)
(* ENTITY man                           *)
(****************************************)
(* Represents a man (a kind of person). *)
(****************************************)

ENTITY man SUBTYPE OF (person);
END_ENTITY;

(******************************************)
(* ENTITY woman                           *)
(******************************************)
(* Represents a woman (a kind of person). *)
(******************************************)

ENTITY woman SUBTYPE OF (person);
END_ENTITY;

(***************************************************************)
(* TYPE university_role                                        *)
(***************************************************************)
(* Represents the role that a person can play in a university: *)
(* either a student, or a member of staff, but not both.       *)
(***************************************************************)

TYPE university_role = SELECT (student, staff);
END_TYPE;

(************************************************************)
(* ENTITY student                                           *)
(************************************************************)
(* Represents a person's role as a student in a university. *)
(************************************************************)

ENTITY student;
  courses_taken : SET [0 : ?] OF student_course_record;
DERIVE
  average_mark : REAL
    := sum_marks(courses_taken) / SIZEOF(courses_taken);
INVERSE
  the_person : person FOR role;
END_ENTITY;

(*****************************************************************)
(* ENTITY student_course_record                                  *)
(*****************************************************************)
```

```
(* Represents the association between a particular student and a *)
(* particular course that the student takes.                     *)
(*****************************************************************)

ENTITY student_course_record;
  course : course;
  exam_mark : REAL;
  lab_mark : REAL;
DERIVE
  total_mark : REAL
    := exam_mark + lab_mark;
INVERSE
  the_student : student FOR courses_taken;
END_ENTITY;

(**********************************************************)
(* FUNCTION sum_marks                                     *)
(**********************************************************)
(* Function to compute the sum of marks of a given set of *)
(* student course records.                                *)
(**********************************************************)

FUNCTION sum_marks (records : SET OF student_course_record) : REAL;
  LOCAL
    i : INTEGER;
    sum : REAL := 0.0;
  END_LOCAL;

  REPEAT i:= 1 TO SIZEOF (records);
    sum := sum + records[i].total_mark;
  END_REPEAT;

  RETURN(sum);
END_FUNCTION;

(*******************************************************************)
(* ENTITY staff                                                    *)
(*******************************************************************)
(* Represents a person's role as a member of staff in a university. *)
(*******************************************************************)

ENTITY staff;
  courses_taught : OPTIONAL SET [1 : ?] OF course;
INVERSE
  the_person : person FOR role;
END_ENTITY;

(**************************************************************)
(* ENTITY course                                              *)
(**************************************************************)
(* Represents a course module in a university.                *)
(* Course codes are of the form 'DDlpn' or 'DDlpns', where:    *)
(* - 'DD' is the department;                                  *)
(* - 'l' is the level (year);                                 *)
(* - 'p' is the programme (stream);                           *)
(* - 'n' is used to give each module a different 'lpn' number; *)
(* - 's' is the semester that the module runs in.             *)
(**************************************************************)

ENTITY course;
  department : STRING(2) FIXED;
  level : INTEGER(*1*);
  programme : INTEGER(*1*);
```

```
    id_number : INTEGER(*1*);
    semester : OPTIONAL INTEGER(*1*);
    title : STRING;
  INVERSE
    the_lecturer : staff FOR courses_taught;
    the_students : SET [0 : ?] OF student_course_record FOR course;
  UNIQUE
    department, level, programme, id_number;
  END_ENTITY;
END_SCHEMA;
```

F.10.2 The Second Model

```
SCHEMA course_world;

  (******************************************************)
  (* ENTITY person                                      *)
  (******************************************************)
  (* Represents a living person, who must be either a man, *)
  (* or a woman, but not both.                          *)
  (******************************************************)

  ENTITY person;
    family_name : STRING;
    given_name : STRING;
    middle_names : OPTIONAL LIST [1 : ?] OF STRING;
    gender : gender_type;
    age : INTEGER;
    role : OPTIONAL university_role;
  WHERE
    age_must_be_sensible:
      ((age < 65) AND (age >= 18));
  END_ENTITY;

  (*******************)
  (* TYPE gender_type *)
  (*******************)
  (* Man or woman.    *)
  (*******************)

  TYPE gender_type = SELECT (man, woman);
  END_TYPE;

  (**************************************)
  (* ENTITY man                         *)
  (**************************************)
  (* Represents a man (a kind of person). *)
  (**************************************)

  ENTITY man;
  END_ENTITY;

  (****************************************)
  (* ENTITY woman                         *)
  (****************************************)
  (* Represents a woman (a kind of person). *)
  (****************************************)

  ENTITY woman;
  END_ENTITY;

  (************************************************************)
```

```
(* TYPE university_role                                     *)
(************************************************************)
(* Represents the role that a person can play in a university: *)
(* either a student, or a member of staff, but not both.    *)
(************************************************************)

TYPE university_role = SELECT (student, staff);
END_TYPE;

(*********************************************************)
(* ENTITY student                                        *)
(*********************************************************)
(* Represents a person's role as a student in a university. *)
(*********************************************************)

ENTITY student;
  courses_taken : SET [0 : ?] OF student_course_record;
  average_mark : REAL;
INVERSE
  the_person : person FOR role;
WHERE
   sum_marks(courses_taken) = average_mark * SIZEOF(courses_taken);
END_ENTITY;

(**************************************************************)
(* ENTITY student_course_record                               *)
(**************************************************************)
(* Represents the association between a particular student and a *)
(* particular course that the student takes.                  *)
(**************************************************************)

ENTITY student_course_record;
  course : module;
  exam_mark : REAL;
  lab_mark : REAL;
  total_mark : REAL;
INVERSE
  the_student : student FOR courses_taken;
WHERE
  total_mark = exam_mark + lab_mark;
END_ENTITY;

(*******************************************************)
(* FUNCTION sum_marks                                  *)
(*******************************************************)
(* Function to compute the sum of marks of a given set of *)
(* student course records.                             *)
(*******************************************************)

FUNCTION sum_marks (records : SET OF student_course_record) : REAL;
  LOCAL
    i : INTEGER;
    sum : REAL := 0.0;
  END_LOCAL;

  REPEAT i:= 1 TO SIZEOF (records);
    sum := sum + records[i].total_mark;
  END_REPEAT;

  RETURN(sum);
END_FUNCTION;

(*****************************************************************)
```

```
  (* ENTITY staff                                                      *)
  (*********************************************************************)
  (* Represents a person's role as a member of staff in a university. *)
  (*********************************************************************)

  ENTITY staff;
    courses_taught : OPTIONAL SET [1 : ?] OF module;
  INVERSE
    the_person : person FOR role;
  END_ENTITY;

  (****************************************************************)
  (* ENTITY module                                                *)
  (****************************************************************)
  (* Represents a course module in a university.                  *)
  (* Course codes are of the form 'DDlpn' or 'DDlpns', where:    *)
  (* - 'DD' is the department;                                    *)
  (* - 'l' is the level (year);                                   *)
  (* - 'p' is the programme (stream);                             *)
  (* - 'n' is used to give each module a different 'lpn' number; *)
  (* - 's' is the semester that the module runs in.               *)
  (****************************************************************)

  ENTITY module;
    department : STRING(2) FIXED;
    level : INTEGER(*1*);
    programme : INTEGER(*1*);
    id_number : INTEGER(*1*);
    semester : OPTIONAL INTEGER(*1*);
    title : STRING;
  INVERSE
    the_lecturer : staff FOR courses_taught;
    the_students : SET [0 : ?] OF student_course_record FOR course;
  UNIQUE
    department, level, programme, id_number, semester;
  END_ENTITY;
END_SCHEMA;
```

F.11 Mr. Jones's Garden

For this section, three revisions of the Mr. Jones's Garden model [STEb] were compared: Simple Solution, Complex Solution 1 and Complex Solution 2. The number of lines of EXPRESS code, frames to represent each model, and HLCs extracted from each model, are as follows:

- Simple Solution: 176 lines, 66 frames, 61 HLCs;
- Complex Solution 1: 205 lines, 78 frames, 65 HLCs;
- Complex Solution 2: 268 lines, 118 frames, 54 HLCs.

First, Simple Solution was compared with Complex Solution 1 (Table F.11). The results show that the comparison effort is very low for these models compared with the Course World models – there are few differences between Simple

Mode	E	-T	-U	-S	-H	-R	M	X
HLCs from first model	0	0	0	0	0	0	0	61
HLCs from second model	4	4	4	4	0	4	4	65
Comparisons	311	311	311	311	311	311	311	2806
HLC-comparisons	0	0	0	0	0	0	0	1386
Correspondences	66	66	66	66	66	66	66	151
HLC-correspondences	0	0	0	0	0	0	0	75
Differences	5	5	5	5	5	5	5	775
FC-thesaurus lookups	4	0	4	4	4	4	4	46
Name-thesaurus lookups	2	0	2	2	2	2	2	742

Table F.11: Experimental results for the simple and first complex solutions to Mr. Jones's Garden.

Mode	E	-T	-U	-S	-H	-R	M	X
HLCs from first model	64	64	64	65	0	65	65	64
HLCs from second model	54	54	54	54	0	54	54	54
Comparisons	2469	2469	2469	2431	293	782	901	2471
HLC-comparisons	1259	1259	1259	1261	0	244	301	1261
Correspondences	105	105	105	97	54	83	86	105
HLC-correspondences	43	43	43	41	0	29	32	43
Differences	688	688	688	693	16	150	178	688
FC-thesaurus lookups	57	0	57	42	2	2	40	57
Name-thesaurus lookups	748	0	748	730	40	180	210	748

Table F.12: Experimental results for the first and second complex solutions to Mr. Jones's Garden.

Solution and Complex Solution 1, and hence few HLCs are extracted (except when submitting extraction prompts together (Mode X) – all the extra HLCs and correspondences in Mode X are extraneous).

Then, Complex Solution 1 was compared with Complex Solution 2 (Table F.12). The results are a very different story in this case: this time there are 16 code-level differences rather than 5, and this causes many HLCs to be extracted and compared, resulting in quadratic explosion (not just in Mode X).

F.12 Mr. Jones's Garden Flattened Up

This section presents a simple example of comparing an EXPRESS model with a transformed version of itself. The Simple Solution to Mr. Jones's Garden [STEb] was compared with a flattened-up transformation of itself. Flattening up entails removing subtypes from a model, moving their attributes and constraints up into the supertype (Section 1.4.2.1). Thus this can be thought of as an extended version of the enumeration/subtypes experiment (Section F.8).

The experiment in this section uses the following name-thesaurus:

```
{plant, flattenedupplant,}
{greenhouseplant, plantgreenhouseplant,}
{outdoorsplant, outdoorsplantplant,}
```

The usual table of measurements can be found on the Author's website [Bro], but is not discussed here as it is of little interest for this particular experiment. This experiment is included for the interesting ways in which the constraints are transformed when flattening up.

The CCUS has no problems in inferring all the right entity-correspondences:

- `garden` corresponds to `garden`;
- `greenhouse` corresponds to `greenhouse`;
- `bed` corresponds to `bed`;
- `plant` corresponds to `flattened_up_plant`;
- `greenhouse_plant` corresponds to the enumeration symbol `plant_greenhouse_plant`;
- `outdoors_plant` corresponds to the enumeration symbol `outdoors_plant_plant`.

Figure F.10 shows `plant` and one of its subtypes, and the corresponding entity `flattened_up_plant`.

- `the_temperatures_must_be_sensible_from_greenhouse_plant` says "if this is a greenhouse-plant then `max_temp >= min_temp` must hold."
- `mandatory_exists__of_greenhouse_plant` says "if this is a greenhouse-plant then `min_temp` and `max_temp` must have non-null values, otherwise they must be null."

```
ENTITY plant ABSTRACT SUPERTYPE OF (ONEOF (greenhouse_plant, outdoors_plant));
  colour : flower_colour;
...
END_ENTITY;

ENTITY greenhouse_plant SUBTYPE OF (plant);
  min_temp : temperature;
  max_temp : temperature;
WHERE
  the_temperatures_must_be_sensible: max_temp >= min_temp;
END_ENTITY;
```

```
ENTITY flattened_up_plant;
  instance_of : instances_in_flattened_up_plant;
  colour : flower_colour;
  min_temp : OPTIONAL temperature;
  max_temp : OPTIONAL temperature;
...
WHERE
  the_temperatures_must_be_sensible_from_greenhouse_plant:
    ((entities_in_flattened_up_plant.greenhouse_plant
      IN entities_in_flattened_up_plant_instance
         (instance_of, combination_sets_for_flattened_up_plant))
     AND (max_temp >= min_temp))
    XOR NOT (entities_in_flattened_up_plant.greenhouse_plant
        IN entities_in_flattened_up_plant_instance
           (instance_of, combination_sets_for_flattened_up_plant));

  mandatory_exists__of_greenhouse_plant:
    (EXISTS(min_temp) AND EXISTS(max_temp))
    XOR NOT (entities_in_flattened_up_plant.greenhouse_plant
             IN entities_in_flattened_up_plant_instance
                (instance_of, combination_sets_for_flattened_up_plant));
...
END_ENTITY;
```

Figure F.10: `plant` from Mr. Jones's Garden, and flattened up.

- `entities_in_flattened_up_instance` is a function which returns the set of entities from which a given instance is constructed.

- `combination_sets_for_flattened_up_plant` is a derived attribute (not shown) which says that an instance can be a complex instance either of `plant` and `outdoors_plant`, or of `plant` and `greenhouse_plant`.

In order to assess that these constraints are semantically equivalent to the mainly structural constraints in the original model, the CCUS could be extended to extract HLCs from the subtype-attributes in the original model, and from the domain rules in the transformed model.

Flattening the subtype hierarchies is only one side of the flatten-up transformation. Any references to the removed subtype-entities (e.g. `greenhouse_plant`) in other entities must be replaced with references to the flattened-up entity (`flattened_up_plant`), adding a constraint to say that the plants must be greenhouse-plants. To illustrate this, Figure F.11 shows the entity `greenhouse` in the original and flattened-up versions of the model.

To infer that the reference to `flattened_up_plant`, together with the domain rule `attribute_reference_check_for_holds_plants`, is semantically equivalent to the reference to `greenhouse_plant`, the CCUS could be extended with a new HLC-class `subtype_reference`. HLCs of this class could be extracted from the reference to `greenhouse_plant` (because it is a subtype) and from a domain rule of the form of `attribute_reference_check_for_holds_plants`. Since the domain rule calls `entities_in_flattened_up_plant_instance`, the CCUS would have to be extended to understand user-defined functions. This would entail more sophisticated code understanding than is currently implemented.

F.13 EDIF 3 0 0 and CFI DR

For this section, the Electronic Design Interchange Format (EDIF) version 3 0 0 Level 0 information model [LK93] was compared with the CAD Framework Initiative Design Representation (CFI DR) information model [CAD92]. EDIF is a neutral format for exchanging design-information among CAD/CAM tools. CFI DR is limited to electrical connectivity, which is just one of the views covered by EDIF. The CFI DR model semantically overlaps with the part of the EDIF 3 0 0 model that models the connectivity of electrical circuits.

```
ENTITY greenhouse;
  temp : temperature;
  holds_plants : SET [1 : ?] OF greenhouse_plant;
WHERE
  the_plants_can_survive_the_temperature:
    QUERY (p <* holds_plants | {p.min_temp <= temp <= p.max_temp})
    = holds_plants;
END_ENTITY;
```

```
ENTITY greenhouse;
  temp : temperature;
  holds_plants : SET [1 : ?] OF flattened_up_plant;
WHERE
  the_plants_can_survive_the_temperature:
    QUERY (p <* holds_plants | {p.min_temp <= temp <= p.max_temp})
    = holds_plants;

  attribute_reference_check_for_holds_plants:
    SIZEOF (QUERY (inst <* holds_plants |
            (EXISTS(inst) AND NOT
             (entities_in_flattened_up_plant.greenhouse_plant
              IN entities_in_flattened_up_plant_instance
                 (inst.instance_of,
                  inst.combination_sets_for_flattened_up_plant)))))
    = 0;
END_ENTITY;
```

Figure F.11: `greenhouse` from Mr. Jones's Garden, and flattened up.

The number of lines of EXPRESS code, frames to represent each model, and HLCs extracted from each model, are as follows:

- EDIF 3 0 0 model: 8355 lines, 6103 frames, 659 HLCs;
- CFI DR model: 963 lines, 530 frames, 61 HLCs.

The CCUS suffers gigantic quadratic explosion due to the abundance of HLCs of the following frame-classes: `type_hlc`, `domain_rules_hlc` and `non_null`. Thus it was decided to run this experiment with these three HLC-classes disabled (i.e. with the extraction cases for extracting them – Appendix E, Sections E.1.14, E.1.15, E.1.16, E.1.25 and E.1.26 – removed from the case library).

The experiment in this section uses the following name-thesaurus:[1]

```
{hierarchymodel, cfidrinstconcepts, cfidrcellconcepts, cfidrportconcepts,
 cfidrportinstconcepts,}
{instance, cfidrinst,}
{instanceports, portinsts,}
{cell, cfidrcell,}
{containinglibrary, owner, instantiatedcluster,}
{viewgroups, views,}
{masterport, cfidrportscalar,}
{inputmasterport, inputinstanceport, cfidrinput,}
{outputmasterport, outputinstanceport, cfidroutput,}
{bidirectionalmasterport, bidirectionalinstanceport, cfidrio,}
{unspecifieddirectionmasterport, unspecifieddirectioninstanceport,
 cfidrundefinedportdirection,}
{masterportbundle, cfidrportbundle,}
{memberports, ports,}
{instanceport, cfidrportinstscalar,}
{instanceportbundle, cfidrportinstbundle,}

{librarymodel, cfidrlibconcepts,}
{library, cfidrlib,}

{logicalconnectivitymodel, cfidrnetconcepts,}
{signalgroup, cfidrnetbundle,}
{members, nets,}
{signal, cfidrnetscalar,}
```

Table F.13 shows the results of comparing the EDIF 3 0 0 and CFI DR models:

- Disabling the thesauri (Mode -T) severely dents the inference of correspondences for such a thesaurus-driven example. However, the use of thesauri does not increase the comparison effort very significantly.
- Subexpression cases (disabled in Mode -U) do not sanction any extra correspondences in this example – the need for them in real models seems to be very uncommon. They just result in a little extra code-level comparison effort.

[1] Some of the correspondences in this section come from [WWK94] (cf. Section 4.2.4).

Mode	E	-T	-U	-S	-H	-R	M	X
HLCs 1st model	659	659	659	659	0	659	659	659
HLCs 2nd model	61	63	61	61	0	59	59	61
Comparisons	109117	106146	97804	92838	1952	8108	69432	109117
HLC-comparisons	14667	13540	14667	14647	0	655	5005	14667
Correspondences	136	52	136	109	34	52	81	136
HLC-corrs.	26	8	26	25	0	8	8	26
Differences	12311	12058	11924	11622	188	753	6665	12311
FC-t. lookups	52735	0	42977	42485	4	2581	40915	52735
Name-t. lookups	41449	0	41029	34168	1478	4698	16430	41449

Table F.13: Experimental results for EDIF 3 0 0 and CFI DR.

- SE/S cases (disabled in Mode -S) also cost little in terms of comparison effort. However, they are quite important in real models – particularly in the way that they sanction correspondences between frames which differ only in unnecessary slots (e.g. `age >= 0` and `0 <= age`, which differ in the `which_way_round` slot).

- The HLC CCUS (disabled in Mode -H) is very expensive in terms of comparison effort because of the quadratic explosion of HLC-comparisons – even with three HLC-classes disabled. HLCs are needed to infer abstract correspondences (in particular, between entities in supertype expressions and enumeration symbols – see below), but they do not really come into their own in this experiment. The correspondences inferred with the HLC CCUS enabled include correspondences between extraneous HLCs as well as the useful correspondences.

- In this example, the use of no remembrance (Mode -R) has cut down the comparison effort significantly compared with unselective remembrance (the default), but at the expense of missing a lot of correspondences. This means that delocalised differences are important in comparing the EDIF and CFI DR models (for enumeration/supertype correspondences in particular).

In order to interpret the results more precisely, Table F.14 breaks down the "Correspondences" row from Table F.13 to show the correspondences inferred between frames of each frame-class. Whenever frames of two different frame-classes are inferred to correspond, this is counted for the frame-class in the *first*

Mode	E	-T	-U	-S	-H	-R	M	X
model_frame	1	1	1	1	1	1	1	1
schema_frame	6	0	6	6	6	6	6	6
entity_type	18	0	18	18	9	9	9	18
explicit_attribute	28	8	28	28	6	12	14	28
derived_attribute	24	24	24	24	1	1	24	24
aggregation_type	1	1	1	1	1	1	1	1
set_type	11	4	11	0	2	4	4	11
list_type	2	0	2	0	2	2	2	2
built_in_constant	13	4	13	0	4	6	6	13
function_call	2	0	2	2	0	0	2	2
built_in_function	1	1	1	1	1	1	1	1
formal_parameter	1	1	1	1	1	1	1	1
equation	2	0	2	2	0	0	2	2
choice_set	1	0	1	0	0	0	0	1
optional_xor_empty_aggregation	25	8	25	25	0	8	8	25
Total	136	52	136	109	34	52	81	136

Table F.14: Breakdown of correspondences for EDIF 3 0 0 and CFI DR.

model (EDIF) only. Thus each column in Table F.14 adds up to the total number of correspondences for that CCUS run.

The eighteen `entity_type` correspondences consist of nine correspondences between pairs of entities, and nine correspondences between entities and enumeration symbols which are only inferred in modes that compare HLCs using unselective remembrance. The former nine correspondences are as follows:

- `library` in EDIF corresponds to `cfidrLib` in CFI DR;
- `cell` corresponds to `cfidrCell`;
- `instance` corresponds to `cfidrInst`;
- `master_port` corresponds to `cfidrPortScalar`;
- `master_port_bundle` corresponds to `cfidrPortBundle`;
- `instance_port` corresponds to `cfidrPortInstScalar`;
- `instance_port_bundle` corresponds to `cfidrPortInstBundle`;
- `signal` corresponds to `cfidrNetScalar`;

- `signal_group` corresponds to `cfidrNetBundle`.

The EDIF entity `cluster` roughly corresponds to `cfidrNetlistView`[2] because they represent the instantiatable objects in EDIF and CFI DR respectively. In CFI DR, cells consist of views, whereas in EDIF, cells consist of clusters and view-groups – the clusters being the units of instantiation.

It might not be appropriate, therefore, to infer correspondence between `cluster` and `cfidrNetlistView`, but a more sophisticated CCUS might suggest this as a possible correspondence and ask the user whether to infer it or not. It might suggest this correspondence in one of the following ways:

- it could use *structural similarity* to reason that clusters come between cells and views (cf. Section 4.2.6.1);
- it could use an *ontology* to reason about the relationship between cells, clusters and views (cf. Section 4.2.6.3).

The CCUS is less complete when it comes to inferring correspondences between **attributes** in the EDIF and CFI DR models. One reason for this is that it does not currently propagate attributes from supertypes to subtypes so that they can be compared. For example, it fails to infer the following correspondences between the attributes of the EDIF entity `signal` and of the CFI DR entity `cfidrNetScalar`:

- `joined_instance_member_ports` corresponds to `PortInsts`, which `cfidrNetScalar` inherits from `cfidrNet`;
- `joined_master_ports` corresponds to `Ports`, which `cfidrNetScalar` inherits from `cfidrNet`;
- `properties` corresponds to `Props`, which `cfidrNetScalar` inherits from `cfidrObject`;
- the derived attribute `containing_view` corresponds to `Owner`, which `cfidrNetScalar` inherits from `cfidrNet`. To be able to infer this correspondence, the CCUS would also have to compare the set of explicit attributes with the set of derived attributes for each entity (possibly by representing them as a single set of attributes).

[2]The CCUS deliberately does not infer this correspondence.

Another reason for the lack of completeness of inferring correspondences between attributes is that the CCUS cannot currently infer correspondence between attributes of non-corresponding entities. For example, continuing the comparison of `signal` and `cfidrNetScalar`:

- `name.primary_name.original_name.name` corresponds to `Name` (inherited from `cfidrNamedObject`) (cf. Section 4.2.6.5);

- the inverse attribute `members` of `signal_group` corresponds to `NetBundles` (inherited from `cfidrNet`).

Such inter-entity attribute-correspondences could be inferred in one of the following ways:

- having the mapping specified explicitly as an input to the CCUS (cf. Section 4.1.1);

- using an ontology to reason about the relationship between nets and net-bundles.

It may also be the case that an attribute in one model corresponds to an arbitrary expression in the other model:

- when mapping `cfidrNetScalar` to `signal` in EDIF, `ObjectId` corresponds to `SELF`, and `ObjectType` corresponds to `TYPEOF` [WWK94];

- when mapping `master_port` in EDIF to `cfidrPortScalar`, `unused_internally` corresponds to `NOT EXISTS(NetScalar)`.

Again, the CCUS would probably have to have these mappings specified explicitly as input.

Courtesy of extracting and comparing HLCs, the CCUS can infer abstract correspondences between entities in a `ONEOF` supertype expression in one model and enumeration symbols in the other (Section 6.3.3):

- the EDIF entities `unspecified_direction_master_port` and `unspecified_direction_instance_port` both correspond to the enumeration symbol `CFIDR_UNDEFINED_PORTDIRECTION`;

- `input_master_port` and `input_instance_port` both correspond to `CFIDR_INPUT`;

- `output_master_port` and `output_instance_port` both correspond to `CFIDR_OUTPUT`;
- `bidirectional_master_port` and `bidirectional_instance_port` both correspond to `CFIDR_IO`.

F.14 Gerber and SVG

For this section, the Gerber Conceptual Model [STEa] was compared with the Scalable Vector Graphics (SVG) Model 4.7 (Chapter 4 of [Fan02]). SVG is a language for graphics, whereas Gerber is a language for photoplotting (the drawing of graphics by shining light through apertures).

There are three EXPRESS models of Gerber: the Conceptual Model, the Implementation Model, and the Syntax Model.[3] The Conceptual Model is closest to the SVG Model, and thus was chosen for this experiment.

The number of lines of EXPRESS code, frames to represent each model, and HLCs extracted from each model, are as follows:

- Gerber Conceptual Model: 1071 lines, 674 frames, 90 HLCs;
- SVG Model: 2945 lines, 1933 frames, 135 HLCs.

The CCUS suffers gigantic quadratic explosion due to the abundance of HLCs of the following frame-classes: `type_hlc`, `domain_rules_hlc` and `non_null`. Thus it was decided to run this experiment with these three HLC-classes disabled (i.e. with the extraction cases for extracting them – Appendix E, Sections E.1.14, E.1.15, E.1.16, E.1.25 and E.1.26 – removed from the case library).

The experiment in this section uses the following name-thesaurus:[4]

```
{gerber, svggeometrymodel, svgstructuremodel, svgpresentationmodel,}
{gerberplot, outmostsvgelement,}
{gerberpoint, point, point3d,}
{xcoordinate, x,}
{ycoordinate, y,}
{zcoordinate, z,}
{coordinatevalue,}
{millimeter, svglengthinmillimetre,}
{inch, svglengthininch,}
```

[3] In separate experiments reported on the Author's website [Bro], the Syntax Model was compared with the Implementation Model, and the Implementation Model was compared with the Conceptual Model.

[4] The correspondences in this section come from the Gerber-to-SVG mapping in Chapter 5 of [Fan02].

```
{linearinterpolation, line,}
{fromposition, startpoint,}
{toposition, endpoint, drawtopoint,}
{circularinterpolation, ellipticalarccurve,}
{parabolicinterpolation, quadraticbeziersegment,}
{cubicinterpolation, cubicbeziersegment,}
{throughpoints, controlledparameters,}
```

Table F.15 shows the results of comparing the Gerber and SVG models:

- Disabling the thesauri (Mode -T) severely dents the inference of correspondences for such a thesaurus-driven example. However, the use of thesauri does not increase the comparison effort very significantly.

- SE/S cases (disabled in Mode -S) are not as advantageous for Gerber/SVG as they are for EDIF/CFI DR (Section F.13). Here, the cost in terms of code-level comparisons is more significant, and they only sanction two extra correspondences.

- The HLC CCUS does not make such a big difference for Gerber/SVG either – it only sanctions six abstract correspondences (beyond the 27 correspondences inferred by the code-level CCUS in Mode -H).

- The use of no remembrance (Mode -R) has cut down the comparison effort significantly compared with unselective remembrance (the default), but this time with the loss of only three correspondences. Thus delocalised differences are not as important for Gerber/SVG as they are for EDIF/CFI DR.

- Monogamy (Mode M) really comes into its own in this experiment. It infers as many correspondences as unselective remembrance, with only slightly more comparison effort than no remembrance (which is roughly a third of the comparison effort for unselective remembrance).

The CCUS infers the following entity-correspondences between the Gerber and SVG models:

- `gerber_plot` corresponds to `outMostSvgElement`;

- `coordinate` corresponds to `coordinate`;

- `gerber_point` corresponds to `point`, and also to `Point3D`;

- `linear_interpolation` corresponds to `line`;

Mode	E	-T	-U	-S	-H	-R	M	X
HLCs 1st model	92	92	92	92	0	92	92	92
HLCs 2nd model	135	135	135	135	0	135	135	135
Comparisons	90755	71909	90401	75003	18844	31276	31976	90755
HLC-comparisons	6742	6742	6742	6742	0	1688	1732	6742
Correspondences	33	6	33	31	27	30	33	33
HLC-corrs.	0	0	0	0	0	0	0	0
Differences	3541	3451	3523	3522	86	974	1036	3541
FC-t. lookups	43144	0	42891	31298	2	5212	5648	43144
Name-t. lookups	79343	0	79339	64040	18721	28782	28808	79343

Table F.15: Experimental results for Gerber and SVG.

- `circular_interpolation` corresponds to `ellipticalArcCurve`;
- `instance_port` corresponds to `cfidrPortInstScalar`;
- `parabolic_interpolation` corresponds to `quadraticBezierSegment`;
- `cubic_interpolation` corresponds to `cubicBezierSegment`;
- `symbol` corresponds to `symbol`. The term `symbol` is actually a homonym, for the entities represent different concepts in Gerber and in SVG. In order not to infer this correspondence, the CCUS would have to be extended to veto homonyms (Section 4.2.4).

This experiment also demonstrates needs for the CCUS to be extended in the following ways:

- comparing attributes with attribute-redeclarations (e.g. `image_name` in `gerber_plot`, and `SELF\container.etitle` in `outMostSvgElement`);
- propagating attributes from supertypes to subtypes (e.g. `image_x_justification` in `gerber_plot`, and `viewPort` in `svgElement`);
- inferring correspondence between two attributes, one of which is in an encapsulated entity (continuing with the above two examples, `image_name` actually corresponds to `SELF\container.etitle.eContent`, and `image_x_justification` corresponds to `viewPort.topLeftPoint.x`).

The most feasible way to handle encapsulated entities would be for the CCUS to take as input a (partial) specification of the mapping between the two models (Section 4.1.1). The Gerber-to-SVG mapping (Section 5.3 of [Fan02]) is written in EMM (Section 3.4 of [Fan02]); the CCUS could also take mappings in languages such as EXPRESS-M [Bai95] or EXPRESS-X [Lab96]. The problem is that these languages represent mappings in a very procedural, unidirectional way; to represent them as declarative, bidirectional correspondences is a major challenge.

The CCUS would also have to be extended to handle computational mappings, and one-to-many relationships between attributes in the two models. For example, when mapping an SVG `ellipticalArcCurve` to a Gerber `circular_interpolation`, the attribute `center_point` would be calculated from `startPoint`, `drawToPoint`, `radiusX`, `radiusY` and `xAxisRotation`. Again, the mapping would have to be given as input (unless a very sophisticated CCUS could use an ontology of geometry to infer it).

Finally, although the CCUS can infer correspondences between `through_points` of `parabolic_interpolation` and `controlledParameters` of `quadraticBezierSegment` (courtesy of the name-thesaurus), it is a long way from understanding the mapping as a human can. A parabolic interpolation in Gerber is represented by three points, while a quadratic Bezier segment in SVG is represented by two pairs of points: each an actual point and a control point, where the control points are constrained to be the same – hence three points. Such arbitrary reasoning is well beyond the scope of any foreseeable extension of the CCUS.

F.15 Conclusions

The conclusions from the experiments in this appendix can be summarised as follows:

- Thesauri are very necessary for real models, and do not increase the comparison effort much (e.g. Section F.13).
- Subexpression cases have only proved useful in constructed examples (Section F.2). Disabling them (Mode -U) slightly reduces the comparison effort.
- SE/S cases are much more important – particularly in the way that they sanction correspondence between frames with differences in unnecessary

slots (e.g. Section F.13), although sometimes they are not so advantageous (Section F.14). Disabling them (Mode -S) can even **increase** the comparison effort, because it attempts to resolve the differences by extracting and comparing HLCs instead (Section F.8).

- Extraneous HLCs constitute the biggest performance problem, occurring even for trivial differences (Section F.1). They can lead to a quadratic explosion of HLC-comparisons. Sometimes this quadratic explosion is so severe that some HLC-classes have to be disabled – namely `type_hlc`, `domain_rules_hlc` and `non_null` (Sections F.13 and F.14).

- The HLC CCUS is needed to infer abstract correspondences, but is very expensive in terms of comparison effort. Disabling the HLC CCUS (Mode -H) can be a big handicap, however HLCs have not proved to be all that useful for comparing real models, where thesaurus-driven inference of correspondences is the dominant technique (Sections F.13 and F.14).

- Submitting extraction prompts together (Mode X) is occasionally needed for inferring correspondences (Section F.9), but greatly exacerbates the quadratic explosion, and infers many extraneous correspondences.

- The more selective the remembrance heuristic, the fewer comparisons are performed, but the fewer correspondences are inferred. No remembrance (Mode -R) and monogamy (Mode M) can vastly reduce the comparison effort compared with unselective remembrance (the default) – no remembrance reduces comparison effort more than monogamy does. The pay-off between comparison effort and correspondences inferred depends very much on the particular models being compared.

- Sometimes no remembrance misses a lot of correspondences (Section F.13), sometimes not (Sections F.10 and F.14).

- Monogamy, in its best circumstances, can infer as many correspondences as unselective remembrance, with only slightly more comparisons than no remembrance (Section F.14).