Chapter 1: Introduction

This thesis is about an automated understanding of the constraints on information models.[1] In the context of this thesis, a *constraint* on an information model is any condition which must not be violated[2] by a data repository that conforms to the model. This includes both explicit conditions such as domain rules, and conditions implicit in the structure of the model, such as those implied by an entity's attributes.

Understanding constraints on information models is a special case of understanding code in general, so to set this work in its wider context, Section 1.1 introduces code understanding and its inverse, code generation.

The aim of this research is to develop a method which will enable a computer to extract an understanding of the constraints on a given information model. Section 1.2 provides the background for this task, introducing information modelling and analysing sample models to show what the constraints are.

Section 1.3 introduces schema-to-schema mapping as an underlying motivation for understanding constraints in the particular way that will be presented in this thesis, which is intended to support a comparative understanding of the constraints on two models.

This research takes a case-based reasoning approach to understanding constraints, so case-based reasoning is introduced in Section 1.4.

Section 1.5 establishes the aims and objectives of this research, and Section 1.6 gives an overview of the rest of the thesis.

---

1. Most constraints apply to specific entities within information models, but this thesis will consistently refer to "the constraints on information models" rather than "the constraints on entities in information models".
2. A constraint is *violated* if the condition evaluates to false, and *satisfied* if it evaluates to true. It could also evaluate to unknown, but only false is unacceptable.

1.1 Code Generation and Code Understanding

One of the ultimate dreams of computer science is *automatic programming*, the ability of computers to program themselves. Automatic programming could mean the generation of an executable program from a very-high-level, application-oriented task specification (as mentioned in Rich and Shrobe [60]). It could also entail modifying existing programs, transforming them, or translating them from one programming language to another.

The inverse of automatic programming (code *generation*) is code *understanding*, in which knowledge is extracted from existing code. For code understanding, this knowledge is usually at a higher level of abstraction than the code itself;[3] it captures the implicit semantics of the code.

The motivation for code understanding is typically to enable a human or a computer to make changes to the code or to reimplement it in another form. For example, software reuse requires the code that is being reused to be understood (Geppert and Dittrich [29]), as does translation of code from one language to another (Quilici [58]). Code understanding thus goes hand-in-hand with code generation.

Code understanding and generation will be further discussed in Chapter 2.

It is important to note that code understanding and generation are not confined to program code, but can be applied to code written in any formal language. In particular, these ideas can be applied to information models, and it is to this that we now turn.

3. A compiler could be said to understand code and model it at a lower level of abstraction, but this thesis uses the term *code understanding* in the sense of modelling code at a higher level of abstraction.

1.2 Understanding Constraints on Information Models

This section provides the background and motivation for the task of understanding the constraints on information models. The section starts with an introduction to information modelling, and investigates some sample models with a particular emphasis on constraints. This is followed by a discussion in Section 1.2.2 of the role of constraints and the need to understand them. Section 1.2.3 discusses the need for a *comparative* understanding of the constraints on two models.

1.2.1 Information Modelling

An *information model* is a formal description of the concepts in a particular domain of discourse. It specifies the entities in the domain, the relationships between those entities, and the constraints on those entities and relationships (Section 1.1 of Kasadha [38]).

An *entity* represents a real-world concept in the domain. An entity is a class of objects (such as people or university courses) which satisfy certain constraints, as will be discussed shortly. An *instance* of an entity represents a particular object which satisfies the constraints on that entity. Every entity instance in a data repository that conforms to an information model should[4] satisfy the constraints which the model specifies on that entity. Furthermore, all instances in the repository should collectively satisfy any global constraints.

All the information models referred to in this thesis are written in the EXPRESS language [34], an international standard (ISO 10303-11) which was developed originally for the exchange of product model data. An overview of EXPRESS can be found in Section 2.2 of Kasadha [38].

---

4. An implementation might not be forced to check constraints, so constraints *should* be satisfied rather than they *must* be satisfied.

Note that an EXPRESS model is just a symbolic representation of real-world concepts. The model itself is not concerned with how the symbols are interpreted in the real world, although models are usually written so that humans will interpret them in a meaningful way (e.g. through the use of meaningful names). This thesis is concerned with analysing given models, not with how religiously those models adhere to the principles of 'good' information modelling. With that caveat aside, let us proceed.

1.2.1.1 A Simple EXPRESS Model

Figure 1-1 shows a simple EXPRESS model, which will be examined with regard to the constraints on it.

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

Figure 1-1. A simple model of people.

The schema[5] `simple_people` in Figure 1-1 has a single entity, `person`, which represents a living person. The entity `person` has two attributes, `name` and `age` (which is intended to be interpreted as the number of years since the person was born). The obvious constraint comes from the domain rule `age_must_be_sensible`, which asserts that the age of a living person must be between zero inclusive and one hundred and thirty exclusive.

5. In EXPRESS, a *schema* is a collection of definitions (such as entity types, defined types and functions), and is the smallest unit that an EXPRESS parser will accept (Section 2.2.2 of Kasadha [38]). An EXPRESS model can consist of multiple schemata, but that distinction is not important for this thesis. In schema-to-schema mapping (Section 1.3), the word *schema* is used to mean *model* in the EXPRESS sense of a set of schemata. This thesis will consistently use the word *model*.

That `name` has to be a string and `age` has to be an integer[6] are also constraints, as is the fact that they are mandatory attributes[7] and must therefore have non-null values. To generalise, the constraint implied by a mandatory attribute $a : t$ of an entity e is that every instance of e must have a value for a which is of type t.

Thus some constraints are imposed by the structure of the model, while others are stated explicitly in rules. Both explicit and implicit constraints can be important to reason about.

1.2.1.2 A More Complicated EXPRESS Model

Figure 1-2 is a more complicated example of an EXPRESS model which illustrates a wider range of the kinds of constraints that may need to be understood.

The schema `complicated_people` in Figure 1-2 has four entities: `person_name`, `person` (which, again, represents a living person), `man` and `woman`. These four entities have relationships between them:

- Each instance of `person` has an instance of `person_name` associated with it, through the attribute `name`.
- Similarly, each instance of `person` has an instance of `man` and an instance of `woman` associated with it, through the attributes `father` and `mother` respectively. The attribute `father` is constrained to have an instance of the entity `man` as its value, and the attribute `mother` is constrained to have an instance of the entity `woman` as its value, though either of these attributes could be null, because they are both optional.[8]

---

6. `STRING` and `INTEGER` are types which are built into EXPRESS.
7. A *mandatory attribute* is one which is not marked `OPTIONAL`.
8. These attributes *must* be optional, for if not, there would have to be an infinite number of instances of `person` - so they are *not* constrained to have non-null values. Since the model is restricted to living people, a value of null for `father` or `mother` could be interpreted as that parent being dead, or it could simply represent that this information is unknown.

```
SCHEMA complicated_people;
  ENTITY person_name;
    family_name : STRING;
    given_names : LIST [1 : ?] OF STRING;
  INVERSE
    the_person : person FOR name;
  END_ENTITY;

  ENTITY person ABSTRACT SUPERTYPE OF (ONEOF (man, woman));
    name : person_name;
    age : REAL;
    father : OPTIONAL man;
    mother : OPTIONAL woman;
  UNIQUE
    name_must_be_unique: name;
  WHERE
    parents_must_be_older_than_self:
      ((father.age > age) AND (mother.age > age));
  END_ENTITY;

  ENTITY man SUBTYPE OF (person);
  END_ENTITY;

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

Figure 1-2. A more complicated model of people.

There is also a supertype-subtype relationship between `person` and `man`, and between `person` and `woman`. The entities `man` and `woman` inherit the attributes `name`, `age`, `father` and `mother` from their supertype, `person`, and are thus constrained to have values for those attributes. The subtypes `man` and `woman` are also constrained by the local rules `name_must_be_unique` and `parents_must_be_older_than_self` (more on these rules later), which they also inherit from `person`.

In EXPRESS, inheritance can be understood in terms of *complex instances*. Because `man` is declared to be a subtype of `person`, every instance of `man` is tied to an instance of `person`, and these two instances form a so-called *complex instance*. Likewise, every instance of `woman` is tied to an instance of `person` to form a complex instance.

The entity person is declared to be abstract, which means that an instance of person cannot exist on its own (i.e. without being tied to either an instance of man or an instance of woman). The entity person is also declared to be a supertype of ONEOF(man,woman), which constrains each instance of person to be tied to an instance of at most one of the entities man and woman. In other words, every instance of person must be either a man or a woman - it cannot be neither (because of the ABSTRACT) and it cannot be both (because of the ONEOF).

Other noteworthy constraints come from the type of given_names in person_name, a list of strings (the fact that it is a list means that the order of elements is significant and elements can be duplicated). There must be at least one element in this list, but there is no upper bound on the list's cardinality ('?' is the symbol for an indeterminate bound).[9] Each element in the list is constrained to be a string.

The inverse attribute the_person in person_name represents an existence constraint, meaning that for each instance of person_name, there must exist an instance of person for which the value of name is that particular instance of person_name. The INVERSE clause rules out any instances of person_name existing without being associated with a person.

The uniqueness rule name_must_be_unique in person is another constraint, meaning that each instance of person must have a different value of name from all the other instances of person.[10]

9. In EXPRESS, LIST [n : m] constrains a list to have a minimum of n elements and a maximum of m elements. This is also true of sets and bags, but not arrays - ARRAY [n : m] means an array with a lower bound of n and an upper bound of m.
10. This model ignores the possibility that different people could have the same name - a 'better' information model would use a truly unique identifier such as a person's DNA, but that is beside the point of this example.

Finally, the domain rule `parents_must_be_older_than_self` in `person` expresses the constraint that a person's parents must both be older than that person.[11] It follows that there cannot be any circularities in the family tree, since this constraint imposes a total order on any chains of `person` instances linked by `father` or `mother`.

The above discussion is a typical example of the sort of understanding which a system that reasons about the constraints on EXPRESS models might need. The next subsection discusses the role of such constraints and the need to understand them in the sense illustrated above.

1.2.2 The Role of Constraints

The main reason why constraints are needed in information models is to capture the semantics of the models (and in particular, of the entities therein) by expressing the restrictions on those models (and entities). Since this thesis considers constraints to be *any* conditions on a model that are supposed to be satisfied but could be violated, including those implied by the structure of the model, it follows that a totally unconstrained model would sanction instances which could have values of any arbitrary types for any arbitrary attributes! That would defeat the point of information modelling, which is to model the concepts in a domain precisely.

Constraints characterise the things that can be valid instances of concepts and the things that cannot, by stating their necessary and sufficient conditions. Constraints are an integral part of the intensional definition[12] of a concept, and need to be specified faithfully to the domain - a model should specify just those constraints which do apply to the real-world concepts that the model represents, no more and no less.

11. The possibility that `father` or `mother` may be null does not cause a problem: the appropriate conjunct in `((father.age > age) AND (mother.age > age))` would evaluate to unknown - the constraint is only violated if at least one of the conjuncts evaluates to false.
12. An *intensional definition* of a concept is a rule for determining whether a given object is an instance of that concept (Section 4.3.4 of Rich and Knight [64]).

Not only is it elegant to describe a domain precisely in a formal language such as EXPRESS, expressing the constraints on a model also facilitates the automatic checking that those constraints are satisfied by (the entity instances in) any data repository which purports to conform to that model. A computer can thus verify that a repository does indeed conform to the model, or else determine in what ways it does not (i.e. which specific constraints are violated). Constraints enforce the integrity of a repository with respect to the model, a topic which has long been studied in the field of databases (Section 21.4 of Elmasri and Navathe [21]).

Another role of constraints is to help retain the semantics of a model when it is transformed, or when mapping from one model to another. This role gives rise to the need for a *comparative* understanding of the constraints on the two models, which will be discussed in Section 1.2.3.

There are various possible motivations for understanding the constraints on information models, for example:

- Generating code to check constraints on implementations derived from an information model;
- Interpreting the meaning of constraints for humans (constraints on realistic information models are often expressed as complicated logical expressions which are not easy for humans to understand);
- Interpreting the meaning of constraints for machines (in order to support particular tasks such as schema-to-schema mapping, as discussed in Section 1.3);
- Making sure that the constraints are sensible (e.g. that they are not impossible for instances to satisfy);
- Comparing the semantics of the constraints on two models, to see where they overlap and what the differences are. This is the motivation for understanding constraints as presented in this thesis, and will be discussed in the next subsection;
- Understanding the constraints on two models may even help to deduce knowledge of the

computational mappings between those models. For example, imagine two models such that temperature is represented in degrees Celsius in one model and degrees Fahrenheit in the other. If both models had range constraints that were known to be equivalent (e.g. range constraints for the temperature of water), then it would be possible to deduce the mapping function, knowing that it is of the form ($\lambda x.\ mx + c$).

1.2.3 Understanding the Comparative Semantics of the Constraints on Two Models

The general aim of this thesis is to extract a higher-level understanding of the constraints on information models: that is, to derive a more abstract view of the constraints that makes explicit their implicit semantics (i.e. semantics that are not directly expressed). In EXPRESS, many constraints have implicit semantics. For example, a `WHERE` clause which says `(0 <= age) AND (age < 130)` *implies* that `age` has an inclusive lower bound of zero and an exclusive upper bound of one hundred and thirty (these are *bound* constraints on the attribute `age`), which together constitute a *numeric range* constraint on `age`.

These explicit statements of the implicit semantics of constraints are referred to in this thesis as *higher-level constraints*, which are the output of the constraint-understanding process. Bound constraints and numeric range constraints are two particular kinds of higher-level constraint.

This thesis considers the *comparative* semantics of the constraints on two models. Two models of the same concept written differently may have the same meaning in terms of how they are constrained (or there may be differences). For example, the models in Figure 1-3 and Figure 1-4 both have the abstract semantics that a person must be either male or female (but not both). The former captures this meaning with an explicit attribute `gender` whose value must be either `male` or `female`, while the latter encodes this information by the entity type of each subtype instance (either `man` or `woman`).

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

  ENTITY person;
    gender : gender_type;
  END_ENTITY;
END_SCHEMA;
```

Figure 1-3. Representing gender by the value of an enumeration type.

```
SCHEMA gender_by_subtyping;
  ENTITY person ABSTRACT SUPERTYPE OF (ONEOF (man, woman));
  END_ENTITY;

  ENTITY man SUBTYPE OF (person);
  END_ENTITY;

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

Figure 1-4. Representing gender by subtypes.

It is possible to test for this kind of semantic equivalence by extracting and then comparing the abstract meaning from two different models. For example, the constraint-understanding system presented in this thesis could recognise that `person` in `gender_by_enumeration` must have exactly one of the properties `male` or `female` (which happen to be values of an enumeration type), and that `person` in `gender_by_subtyping` must have exactly one of the properties `man` or `woman` (which happen to be entity types). A system could realise that these two higher-level constraints are equivalent if it knew about the correspondence between `male` and `man`, and between `female` and `woman`. In this thesis, it will be assumed that such structural equivalences are known. It is beyond the scope of the system presented in this thesis to actually compare higher-level constraints on two models - it just extracts higher-level constraints from a single model.

Understanding constraints in this comparative sense has various potential applications, such as ensuring that as much of the semantics of a model as possible can be retained when it is transformed, for example as the model evolves, or when a richly conceptual model is transformed into one that is conducive to more efficient implementation (see for example Pun [57], or the STEPWISE Project [70]). Schema-to-schema mapping, which is introduced in the next section, also requires a comparative understanding of the constraints on two models.

1.3 Schema-to-Schema Mapping

Schema-to-schema mapping (Brown, Filer and Moosa [11]) is the task of transferring entity instances from one data repository to another. The two repositories, which shall be called the *source repository* and the *target repository*, each conform to a model (written in EXPRESS or some other modelling language). These models shall be called the *source model* and the *target model* respectively.

The goal for research on schema-to-schema mapping is: given two models (and a specification of the mapping between them), to generate automatically a program to map instances from one repository to another, as opposed to having humans write the mapping code by hand (c.f. Section 1.1).

The motivation for schema-to-schema mapping is to promote the interoperability of software systems by supporting mappings between heterogeneous repositories. For example, it allows mapping between different versions of the same model as it evolves, and facilitates the integration of computer-aided design tools and data interchange between applications.

The two repositories are stored in different representation formats (as described by their models), which may be incompatible. This problem is referred to as *semantic heterogeneity* (see, for example, Drew, King et al [19]). Schema-to-schema mapping must overcome the problem of semantic heterogeneity in order to ensure that the integrity of the target repository (with respect to the target model) is maintained when instances from the source repository are mapped to the target repository.

In the context of schema-to-schema mapping, the problem of semantic heterogeneity has two aspects:

- the structural differences between the two models;
- the differences in the semantics of the constraints on the two models.

These two aspects are elaborated in the two subsections that follow.

1.3.1 Structural Mapping

The first aspect of the problem of semantic heterogeneity in schema-to-schema mapping is that the source and target models may differ *structurally*, which requires that the instances in the source repository be translated into the format specified by the target model.

Let us adopt the following notation:

- "*`schema`*`::`*`entity`*": The entity type *`entity`* is defined in the schema *`schema`*.
- "*`entity`*`:`*`attribute`*": The entity type *`entity`* has the attribute *`attribute`*.

As an example of structural difference, consider mapping from `complicated_people` (Figure 1-2) to `simple_people` (Figure 1-1). For each instance of `complicated_people::person`, the value of `complicated_people::person:name` would have to be converted from an instance of `complicated_people::person_name` to a string, so that it could be assigned to `simple_people::person:name`. Similarly, the value of `complicated_people::person:age` would have to be converted from a real number to an integer.

It is on the structural aspect of schema-to-schema mapping that prior work has concentrated. For example, Brown, Filer and Moosa [11] propose a set of generic operators that declaratively specify the ways in which schemata, entities and attributes must be transformed when mapping from one model to another; the mapping program can be generated automatically from these operators. Pun [57] presents a framework for deducing mappings between a knowledge base and the data structures of application programs. Lindén, Tirri and Verkamo [42, 43] describe ALCHEMIST, an environment for generating data transformations by establishing the mapping from one context-free grammar to another. However, these works do not address the problem of constraints explained in the next subsection.

1.3.2 The Problem of Constraints in Schema-to-Schema Mapping

The second aspect of the problem of semantic heterogeneity in schema-to-schema mapping is that the source and target models may have different constraints on them - that is to say, an instance from the source repository may violate constraints on the target model even though it does not violate the constraints on the source model.

Consider again the mapping from `complicated_people` (Figure 1-2) to `simple_people` (Figure 1-1). It is possible to have an instance of `complicated_people::person` which satisfies all the relevant constraints on `complicated_people`, yet violates the domain rule `age_must_be_sensible` in `simple_people`, because this constraint is not specified for `complicated_people`. For example, there is nothing in `complicated_people` to rule out a 300-year-old person, but it would violate `age_must_be_sensible` if it were mapped to `simple_people`.

What is missing from previous work on schema-to-schema mapping is a way of automatically detecting the possible ways in which constraints could be violated in cases such as this. Code needs to be planted in the (automatically generated) mapping program to check for violation of constraints when instances from the source repository are mapped to the target repository, raising exceptions if constraints are violated.

A simple way for the mapping program to check for violation of target model constraints would be first to translate an instance from the source repository into the format of the target model without checking the constraints, and then to check the target model's constraints on the new target instance before it is committed to the target repository. If the instance does violate the constraints on the target model, then it will have been a complete waste of time to map it to the format of the target model. This waste of time could be expensive if performing the structural mapping is time-consuming and there are a lot of source instances to be mapped, many of which are going to violate the constraints on the target model. It could get even worse in heterogeneous distributed systems in which source instances have to be sent over a network before they can be mapped to the target repository!

It would be convenient to be able to check source instances *before* attempting to map them to the target repository, to make sure that they are not going to violate any constraints on the target model (see Figure 1-5). To facilitate the generation of code to perform such checking, those constraints on the target model that do not hold on the source model need to be transferred to the source model. Figure 1-6 illustrates this transference of constraints.

```
for each instance in the source repository
    if instance violates the constraints from the target model
          then raise exceptions
          else map instance to the target repository
    fi
rof
```

Figure 1-5. Template for a mapping program with prior checking of target model constraints.

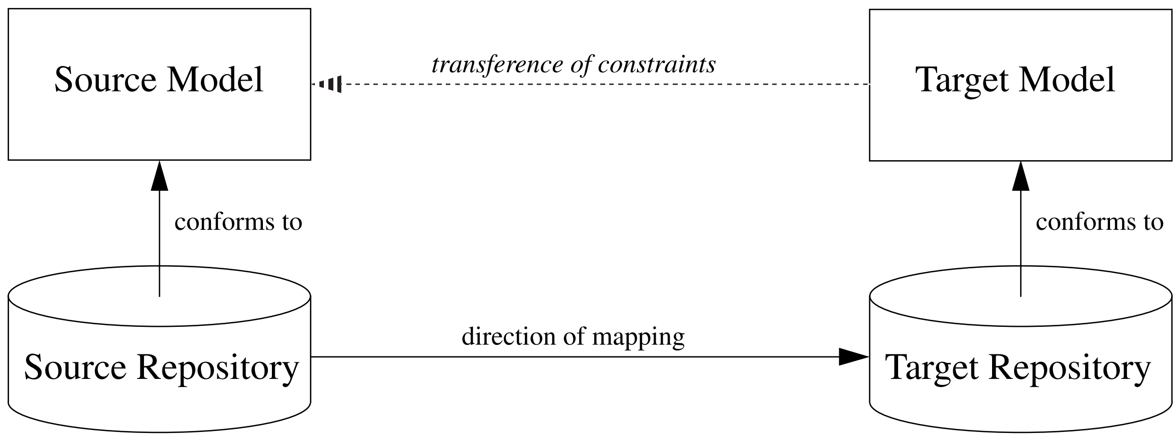

Figure 1-6. Transference of constraints for schema-to-schema mapping.

For example, when generating a program to map from `complicated_people` (Figure 1-2) to `simple_people` (Figure 1-1), the domain rule `age_must_be_sensible` would be transferred from `simple_people` to `complicated_people` (since, as we observed earlier, this constraint is not enforced by `complicated_people`). Figure 1-7 depicts `complicated_people` after `age_must_be_sensible` has been transferred to it. The figure only shows `person` (the entity affected by this transference), and the constraint from the target model is highlighted in bold italics.

Once a source model has had constraints from the target model transferred to it, it shall be referred to as the *transformed source model*.[13]

In this example, the transferred constraint has been realised on the transformed source model by simply copying it across (adapting it slightly for the fact that `age` is a real number in `complicated_people`, whereas it was an integer in `simple_people`).

13. The source model is only *transformed* in the sense that it has extra constraints added to it. It is not transformed structurally, as that would require another schema-to-schema mapping: from the source model to the transformed source model!

```
SCHEMA complicated_people_transformed;
  ...

  ENTITY person ABSTRACT SUPERTYPE OF (ONEOF (man, woman));
    name : person_name;
    age : REAL;
    father : OPTIONAL man;
    mother : OPTIONAL woman;
  UNIQUE
    name_must_be_unique: name;
  WHERE
    parents_must_be_older_than_self:
      ((father.age > age) AND (mother.age > age));
    age_must_be_sensible:
      ((age >= 0.0) AND (age < 130.0));
  END_ENTITY;

  ...
END_SCHEMA;
```

Figure 1-7. A transformed source model.

In practice, constraints would probably not be transferred by literally transforming an EXPRESS model as shown here for the purposes of illustration. What is required, of course, is for code to check the transferred constraints to be planted in the mapping program.

The process of transferring constraints requires the constraints on the two models to be understood in a way which allows their semantic equivalences and differences to be recognised easily (or even recognised at all, in many cases). Transference of constraints requires an understanding of constraints for two reasons:

- Constraints need to be understood in order to determine whether it is necessary to transfer a particular constraint from the target model to the source model. If the source model already implies that constraint, it would be a waste of time for the generated mapping program to check it for the instances in the source repository.[14] For example, no transference of constraints is necessary when mapping between the models in Figures

14. Assuming that the instances in the source repository do not violate constraints on the source model, which the source repository is supposed to conform to in any case.

1-3 and 1-4 because the constraints are totally semantically equivalent.

- When a constraint is transferred from the target model to the source model, it may be necessary to transform that constraint, due to structural differences between the two models. This transformation may be as simple as renaming some attributes, or it may be more radical. For example, consider mapping from `gender_by_enumeration` (Figure 1-3) to `gender_by_subtyping` (Figure 1-4), and imagine that `gender_by_enumeration::person:gender` were optional. The constraint implied by the fact that `gender_by_subtyping::person` is abstract - namely that a person must be either a man or a woman - would have to be transferred to `gender_by_enumeration`. This would be realised by transforming `gender_by_enumeration` to constrain `person:gender` not to be null.

This thesis aims to investigate an understanding of constraints which would support the task of transferring them from a target model to a source model for schema-to-schema mapping. Although it is beyond the scope of this thesis to actually carry out this transference of constraints, it serves as the underlying motivation which drives the research presented in this thesis on understanding constraints. Transference of constraints requires a comparative understanding of the constraints on the source and target models (as discussed in Section 1.2.3).

Code generation and code understanding (as introduced in Section 1.1) manifest themselves in various guises in the context of schema-to-schema mapping:

- Generating the mapping program is an instance of code generation;
- Transferring constraints from the target model to the source model can also be thought of as code generation (regardless of whether the transferred constraints are realised as EXPRESS code (as in Figure 1-7) or as the code which is planted in the mapping program to check those constraints);
- Understanding constraints on models is an instance of code understanding, since those models are written as code in a formal language such as EXPRESS.

The problem of understanding the constraints implied by fragments of EXPRESS could be tackled using contemporary code understanding techniques (which will be reviewed in Chapter 2). However, this thesis proposes a case-based reasoning approach to understanding constraints (and to understanding code in general). Case-based reasoning is therefore briefly introduced in the next section.

1.4 Case-Based Reasoning

Case-based reasoning (Kolodner [39]) is a problem-solving paradigm in which a new problem is solved by remembering an old problem and adapting its solution to fit the new problem, as opposed to solving all problems from first principles. Old problems and their solutions are stored in memory as *cases*.

This thesis proposes a case-based reasoning approach to code understanding (see Section 1.1) and, in particular, to understanding the constraints implied by EXPRESS models (a particular kind of code). A case-based reasoning approach would be to understand code by analogy with similar old code, extracting higher-level knowledge about the new code by reusing and adapting knowledge about the old code. Each case would associate a fragment of code (the 'problem' part of the case) with some higher-level knowledge about that code (the 'solution' part of the case).

In the case of understanding the constraints on information models, fragments of previously experienced models (the 'problems'), with which the appropriate higher-level constraints (the 'solutions') are associated, are stored in memory as cases so that they can be matched to the input models. If a case matches a fragment of an input model successfully, it suggests a higher-level constraint to extract from the model, which characterises the abstract semantics of that fragment (see Section 1.2.3).

Case-based reasoning could also be applied to code generation (see Section 1.1). For code generation, the higher-level knowledge about a code fragment would constitute the ‘problem’ part of a case, while the actual code would constitute the ‘solution’ part. Code would be generated by retrieving cases whose ‘problem’ parts match the specifications of code to be written, and reusing and adapting their ‘solution’ parts to generate code which implements those specifications.

A case-based reasoning system is generally more flexible than the traditional algorithmic or rule-based approaches to problem solving, because:

- It can learn from experience (by acquiring new cases) and thus increase its efficiency and its problem-solving capabilities over time;
- It can suggest solutions in domains where knowledge is too incomplete to write an algorithm, a set of rules to cover the domain, or a general model.

These points are, in the Author’s estimation, the two most important advantages of case-based reasoning (Section 1.5.2 of Kolodner [39]). The second point is particularly important as a reason why it would be beneficial to take a case-based approach to code understanding and generation, as the issue of what knowledge should be extracted from code is ill-specified, as is the issue of how to generate code which implements a given specification.

Chapter 3 covers some background material on case-based reasoning, and includes further discussion of its advantages (and disadvantages), and a justification of its applicability to code understanding (and code generation) and to understanding the constraints on information models.

1.5 Aims and Objectives

The aim of the research presented in this thesis is to develop a method for understanding the constraints on information models at a high level of abstraction, in order to facilitate certain reasoning tasks, namely those requiring a comparative understanding of the constraints on two models (as discussed in Section 1.2.3), such as transferring constraints for schema-to-schema mapping (as explained in Section 1.3.2).

The method presented in this thesis for understanding constraints on information models applies specifically to information models written in EXPRESS. In principle, it should be possible to generalise the method to any modelling language.

The approach taken is to let the investigation of ways of understanding constraints be driven by what is needed to understand the comparative semantics of the constraints on two models. The kinds of higher-level constraints that are appropriate to extract from models are identified by analysing pairs of models in terms of the semantic equivalences of the constraints. It is important to note that actually comparing the semantics of constraints on two models, and transferring constraints for schema-to-schema mapping, are outside the scope of this thesis - they merely serve as the underlying motivation.

An objective for the system for understanding constraints which will be presented in this thesis is that it should be largely data-driven. The system's knowledge of higher-level constraints and how to extract them from EXPRESS code should be explicitly represented as data as opposed to being hardwired into the program. This data-driven approach promises greater flexibility than burying such knowledge in the program code. It should be easier for humans to understand, and easier to extend (without having to modify or recompile the program).

This thesis aims to investigate a case-based reasoning approach to understanding constraints, as discussed in Section 1.4. A large part of the data-driven nature of the constraint-understanding system comes from the use of case-based reasoning, because it stores problems and their solutions as cases, which are explicitly represented as data rather than hardwired into the program.

An experimental computerised system has been implemented, which extracts higher-level constraints from a given EXPRESS model. In principle, such a system should be able to understand the constraints implied by any constructs in EXPRESS; however, this is a very difficult problem, and the constraint-understanding system presented in this thesis can handle only a restricted subset of EXPRESS.[15]

1.6 Thesis Overview

This chapter has introduced the problem of understanding constraints, discussed the underlying motivation for this research, and established the aims and objectives of this thesis. The rest of the thesis is organised as follows.

- **Chapter 2, "Code Understanding and Generation":** Since this thesis aims to understand the constraints in EXPRESS code, the next step is to look at code understanding in general, and at its inverse, code generation. Chapter 2 reviews the literature in these areas.

- **Chapter 3, "Case-Based Reasoning":** This chapter gives an overview of case-based reasoning, including the steps involved and a discussion of its advantages and disadvantages. Chapter 3 provides background material, laying the premise for the justification of a case-based approach to code generation and code understanding in

15. The system ignores any EXPRESS constructs it cannot handle rather than rejecting the input model, so it should be able to process any legal EXPRESS model.

general, and to understanding constraints on information models in particular.

- **Chapter 4, "Architecture of the System":** This chapter presents the top-level architecture of a case-based reasoning system for understanding the constraints on EXPRESS models, which is the subject of the following chapters.

- **Chapter 5, "The Frame System":** This chapter introduces the underlying frame system upon which the constraint-understanding system is built: frames are a formalism for the internal representation of EXPRESS models, higher-level constraints and cases. Chapter 5 discusses languages for frame classes and instances.

- **Chapter 6, "The Frames in the Constraint-Understanding System":** This chapter shows how EXPRESS models, higher-level constraints and cases are represented as frames. It discusses how cases are used to extract higher-level constraints from EXPRESS models. It also discusses how the kinds of higher-level constraints (and hence cases for extracting them from EXPRESS code) which are useful for the comparative understanding of constraints on two models (c.f. Section 1.2.3) can be identified by analysing a range of worked examples.

- **Chapter 7, "Results and Conclusions":** This chapter concludes the thesis by summarising it, presenting some experimental results of running the constraint-understanding system on 'real' EXPRESS models, and discussing the extent to which the aims and objectives (stated in Section 1.5) have been met. It also discusses the potential for further work, both in terms of how the current constraint-understanding system could be enhanced, and the directions in which this thesis could lead to future research.