Appendix G:
Future Work

This appendix discusses the ways in which the research presented in this thesis could be continued.

Section G.1 discusses how the constraint-understanding system could be improved and extended. Section G.2 discusses how the case-based reasoning in particular could be improved.

Since the understanding of constraints discussed in this thesis is geared towards a comparative understanding of the constraints on two models, Section G.3 discusses the actual comparison of the higher-level constraints (HLCs) extracted from two models, which will be a central issue in any system which needs to understand their comparative semantics. The section also discusses some heuristics for guiding the extraction of HLCs in the context of two models for which a comparative understanding of their constraint semantics is being sought.

Section G.4 then discusses how the constraint-understanding system could be used as a building block for systems that need to understand the comparative semantics of constraints on two information models. Transferring constraints for schema-to-schema mapping (see Section 1.3.2) is the particular focus, but there are other applications besides that.

G.1 Extending the Constraint-Understanding System

The constraint-understanding system described in this thesis could be extended and improved in a number of ways, which are discussed in this section.

The system can be extended with more HLCs, according to what is needed to cover the range of models that will be input to the system, expanding the ontology of HLCs presented in Appendix B. New seed cases will be added to the case library to suggest how to extract the new HLCs from appropriate fragments of EXPRESS [34].

The system also needs to be extended to cover more of EXPRESS, as the implementation does not currently support every construct. A list of the significant omissions follows:

- Functions[1] and procedures;
- Global rules (the `RULE` construct);
- Complete reference resolution for identifiers. ExJava objects (see Appendix E, Section E.3) represent these simply as strings, but it is desirable for the frame system to convert these to hard references. The current system achieves this for simple identifiers (attributes of the current entity), but it cannot resolve qualified attributes such as `father.age` in Figure 1-2.[2]

Extending the system with another EXPRESS construct involves writing a frame class for that construct in the EXPRESS frame class model (Appendix A, Section A.1), and writing a corresponding Java class to convert objects of the corresponding ExJava class to frames of the new frame class. However, functions, procedures and rules will require special consideration in terms of understanding the constraints they imply, because they feature imperative constructs which have so far been ignored in developing the constraint-understanding system. How to understand the constraint-like features of imperative code is a major research topic in its own right, and could be tackled by considering some of the techniques for understanding imperative code referred to in Chapter 2.

Another idea is to have an intermediate canonical form between ExJava objects and HLCs. The current system converts ExJava objects to frames conforming to the EXPRESS metamodel in Appendix A, Section A.1, and then proceeds to extract HLCs from those. These EXPRESS frames are very faithful to the corresponding ExJava classes, which means that they are very syntactically biased, and not as flexible as a more semantically

1. Function *calls* are currently supported, but not function *definitions*.
2. In this example, it would successfully resolve `father` (and substitute a reference to the corresponding `explicit_attribute` frame), but leave `age` unresolved (this is represented as an instance of the frame class `unresolved_reference`, which was introduced just for this purpose).

oriented representation would be:

- For example, the system currently represents $(x + (y + z))$ as a binary operation whose kind is addition, whose first expression is x and whose second expression is another binary operation. Instead, it could convert the ExJava object directly to an 'addition' frame whose addends are the bag[3] $\{x, y, z\}$ - this would abstract the semantics away from the syntax, and make it much easier to see the semantic equivalence between $(x + (y + z))$ and $((z + x) + y)$.
- Another example is representing both $(a > b)$ and $(b < a)$ as a comparator which is exclusive, whose 'greater' operand is a and whose 'lesser' operand is b. This would drastically reduce the number of cases needed to extract the 'bound' HLC (see Appendix B, Section B.2). Moreover, the set of cases needed for doing so would collectively emphasise the semantic issue of which operand is the variable and which is the limit, rather than the more syntactic issues of which way round the bound is written, whether it is inclusive or exclusive, and whether it is discrete or continuous, since these three could be represented explicitly in 'comparator' frames and trivially transferred to the 'bound' frames (e.g. via variable unification). The current coverage of extracting the 'bound' HLC from relational operations, for which the current system needs sixteen cases, could be thus achieved with only two cases: one for extracting a lower bound when the 'greater' operand is an explicit attribute and the 'lesser' operand is a literal, and one for extracting an upper bound when the 'lesser' operand is an explicit attribute and the 'greater' operand is a literal. (As discussed in Appendix B, Section B.2, the system needs more cases for interpreting which operand is the limit of a bound and which is the variable being constrained, when the operands are anything other than an attribute and a literal.)

Another extension to the constraint-understanding system would be to allow constraints to 'propagate' from types to entities through those entities' attributes, and from function

3. A *bag* is appropriate for addition because the order of elements is not significant (addition is commutative), but duplicate elements *do* count (addition is not idempotent).

definitions through function calls, etc. For example, Figure B-8 in Appendix B defines `t` as a defined type of underlying type `INTEGER`, constrained by `{3 <= SELF <= 6}`. It would be useful if the system could understand an attribute "`a : t;`" on an entity as though it had been written "`a : INTEGER; WHERE {3 <= a <= 6};`". It would enable the disappointing shortcoming mentioned in Section 7.2 - that the system fails to extract any bounds and ranges from real models because they are expressed on defined types rather than entities - to be overcome (without having to write more cases to extract bounds from domain rules in defined types).

As mentioned in Appendix B, Section B.2, FIL could be extended to allow slot fillers to be arbitrary expressions rather than just literals. This would enable cases such as extracting an *inclusive* bound from ($a > 0$) to be written (e.g. in order to see the semantic equivalence between ($a > 0$) and ($a >= 1$) when a is an integer), and Section B.2's example of extracting an 'exactly one' HLC from a discrete range. This could be achieved if FIL allowed arbitrary expressions in slots in the 'solution' parts of cases, as in "`(?x + 1)`" or "`enumerate_range(?lower, ?upper)`". These would be evaluated when the case's solutions were instantiated, using the bindings recorded when the 'problem' part was matched to the input (e.g. `enumerate_range(3,6)` would evaluate to `{3,4,5,6}`).

FIL could be made a more powerful case language in other ways:

- When instantiating a solution, it might be convenient to edit prompt frames that have been matched to frames in the 'problem' part of the case (e.g. to add extra slots to record information in the input model when HLCs are extracted). Cases are represented as a network of frames, where the frames in the transitive closure of the `problem` slot constitute the 'problem' part and the frames in the transitive closure of the `solution` slot constitute the 'solution' part - although this is elegantly declarative, it means that instantiating the solution can only entail creating new frames, not modifying frames that were matched by the 'problem' part except indirectly through inverse slots (which is why inverse slots were implemented; see Section 5.3.5).

- As mentioned in Appendix D, Section D.2.3, aggregation slot fillers in the 'problem' parts of cases might need to override their default matching policy. FIL could be extended with a special syntax for this, e.g. "`slot == filler`".

EXPRESS frames and HLC frames could be tagged with extra slots to record more of the information that is needed for the tasks which use the constraint-understanding system. In particular, it would be useful to see what lower-level constraints are *covered* by an HLC. For example, if the constraint-understanding system extracted a range HLC from `((age > 0) AND (age < 130))`, then the range covers that expression. If it only extracted a bound from `(age > 0)`, and did not extract a bound from `(age < 130)`, then `(age > 0)` would be covered by an HLC but `(age < 130)` would be uncovered. A system for transferring constraints for schema-to-schema mapping (see Sections 1.3.2 or G.4.1), could use this information to consider constraints at the highest uncovered level for transference. In the previous example, it would consider the bound extracted from `(age > 0)` for transference rather than `(age > 0)` itself, but it would have to consider `(age < 130)` directly for transference. Covering lower-level constraints with HLCs would help the system to decide whether it is necessary to transfer a constraint from the target model to the source model, and would assist it in transforming the constraint in order to express it on the source model. Great care would have to be taken to ensure that the system had a precise understanding of what semantics of a constraint on a model an HLC extracted from that model does and does not cover!

G.2 Improving the CBR in the Constraint-Understanding System

The constraint-understanding system could also be extended by improving the case-based reasoning (CBR) used in the system in the following ways:

- Making retrieval more efficient (Section G.2.1);
- Partial matching and adaptation (other than having variables in cases) (Section G.2.2);

- Chunking smaller inferences into larger cases (Section G.2.3);
- Retaining cases derived from the system's own experience (Section G.2.4).

G.2.1 Improving Case Retrieval

The constraint-understanding system was originally based on linear case library organisation, and hence linear case retrieval. This scheme does have the desirable property of not missing any relevant cases (as deemed by matching algorithms), but it is obviously inefficient and unintelligent, and slows the system down as the case library grows. Retrieval has already been improved by using shallow indexing, as described in Section D.1.3 of Appendix D, but that only cuts down the retrieval of cases on one dimension (the frame class of the top-level prompt frame). This certainly prunes out a lot of cases that would have been retrieved using full linear retrieval, but there is still room for improvement.

As explained in Section D.1.3 of Appendix D, retrieval is more tightly constrained for the task of extracting HLCs than it is for many tasks to which CBR is applied, so it would be inappropriate to apply the retrieval algorithms described in Section 3.2.2.2 without some refinement to ensure that they do not miss relevant cases.

This line of research will involve careful consideration of how cases should be indexed in this domain such that they will always be retrieved when appropriate, but need not be retrieved when they are definitely inappropriate.

It may be that retrieval need not be so tightly constrained for tasks which *use* an understanding of constraints (such as transferring constraints for schema-to-schema mapping, for example - see Sections 1.3.2 and G.4.1), but the constraint-understanding system presented in this thesis just extracts HLCs for the sake of it, so there is no basis for justifying it to miss any. The luxury of extracting all HLCs that can be extracted from the model is expensive, and Section G.3.3 will discuss how extraction could be controlled (as opposed to extracting HLCs indiscriminately), driven by the purpose of the task for which the understanding of constraints is required.

Despite the tightly-constrained nature of retrieval, partial matching and hence adaptation nevertheless have a role to play in the understanding of constraints, as discussed in the next subsection.

G.2.2 Partial Matching and Adaptation

Adaptation (together with partial matching) would enable the system to handle sensible variants of its cases without having to store all those variants explicitly, which could lead to combinatorial explosion in the number of cases required (as mentioned in Section 3.2 of Filer [26]). For example, consider extracting 'bound' HLCs from relational operations. To recap from Section B.2 of Appendix B, bounds can vary orthogonally along the following four dimensions:

- Whether it is a lower bound or an upper bound;
- Whether the bound is inclusive or exclusive;
- Whether the bound is discrete or continuous;
- Which way round the inequality is written.

As illustrated in Appendix B, Figure B-7, this requires $2^4 = 16$ cases, each representing a possible combination of values for four variables which each have two possible values. To generalise, if n dimensions of a case are allowed to vary, each of which has up to m possible values, then $O(m^n)$ cases need to be stored[4] - and these are just variants of a single case. Clearly, it would be impractical (for large cases) to store a number of cases which grows exponentially with the number of dimensions that are allowed to vary.

---

4. The exact number of variant cases is given by multiplying together the numbers of possible values for all dimensions that are allowed to vary - m is the maximum number of possible values across all such dimensions.

Adaptation would enable the system to handle all variants of a case without having to store them all explicitly. In principle, it would only be necessary to store one variant of each case, because the system could derive any other variant as needed. It would do so by adapting the solution of the case it uses, based on the differences between the ‘problem’ part of the case and the piece of the input model (or HLCs which it has already extracted from the model) which it has matched that case to.

Once the system has derived a variant of a case by adaptation and used that to extract an HLC, it may then add that variant to the case library as a case in its own right. That case will then be available for extracting HLCs in similar circumstances. This could have been done anyway using the original case from which the variant was derived. However, the variant may be a closer match in some future situations, requiring less adaptation or none at all. Retaining cases derived by adaptation thus saves time - the system becoming more efficient as it gains experience (unfortunately, this efficiency could be diminished by the effort needed to select which case variant to retrieve, which increases as more variants are stored). So the case library does not need to store all possible variants of a case; it will only end up storing those variants which have been useful in the past. It is one of the fundamental tenets of CBR that something is more likely to be useful in the future if it has already been useful in the past.

A case-based approach to adaptation is proposed. Experiences of adapting solutions (to account for differences between the ‘problem’ parts of cases and actual problem situations) could be recorded as *adaptation cases* (c.f. Section 3.1). Each adaptation case consists of:

- the ‘problem’ part: a difference between the ‘problem’ part of an HLC-extraction case and the corresponding part of a model that needs to be adapted for;
- the ‘solution’ part: how to change the solution of the HLC-extraction case based on that difference.

For example, Figure G-1 shows an adaptation case which is sufficient for adapting all discrete bounds to be continuous, depending on the type of the variable (this case could also be used the other way round, to adapt continuous bounds to be discrete). An adaptation case teaches a more focused lesson than an extraction case; it makes clear the connection between a difference in problems and the corresponding difference in solutions (e.g. the case in Figure G-1 makes it clear that the type of the variable affects whether the bound is discrete or continuous). Although adaptation cases are rather rule-like in nature, and would probably be provided by humans rather than acquired through the system's own experience (it would be more efficient just to provide them rather than to teach them), they can still be considered to be experiential knowledge, and it is elegant to represent them in a unified CBR framework.

Problem: "`age : INTEGER`" in old case, "`age : REAL`" in new case.
Solution: Change "discrete" to "continuous" in bound constraint.

Figure G-1. An adaptation case.

This case is sufficient for adapting bounds along the dimension of whether they are discrete or continuous, no matter how many other dimensions are allowed to vary. Whereas if all variants are stored explicitly, there has to be a discrete and a continuous variant for every other way the bound-extracting case can vary. When all variants of a case are stored explicitly, adding a new way in which that case can vary increases the number of variants of a case *multiplicatively*, whereas adaptation cases to cover those variants would simply be *added* to the case library.

This subsection has discussed how the system could be extended to perform adaptation (other than simple reinstantiation through variables in cases). It has focused on the role of adaptation as a means of avoiding having to store all variants of a case explicitly, though it may nevertheless be desirable for a system to retain (store in the case library) some of the cases it derives by adaptation. Section G.2.4 will discuss some heuristics for controlling this automatic retention of cases, after another situation which gives rise to case retention has been discussed in the next subsection.

G.2.3 Storing Networks of Higher-Level Constraints as Cases

Although the nature of the task of extracting HLCs from information models is that it is a series of small, local problems as opposed to one large, global problem, that does not mean that each problem can be solved with a single inference step (e.g. by applying a case which simply extracts a single HLC from a small fragment of EXPRESS). It may be necessary to extract HLCs from HLCs already extracted, or extract HLCs from subexpressions of a complex expression and then combine them to attain an overall understanding of the expression as a whole, resulting in a network of HLCs extracted from the expression. Figure G-2 shows an entity from the AP214 model for automotive design [36] to illustrate this.

```
ENTITY applied_effective_context_assignment
  SUBTYPE OF (effective_context_assignment);

  items : SET [1:?] OF effective_context_item;
WHERE
  ...
  WR2: (SIZEOF (QUERY (i <* SELF.items |
       'AUTOMOTIVE_DESIGN.ORGANIZATION' IN TYPEOF(i))) = 0)
        OR (SELF.role.name = 'concerned organization');

  WR3: (SIZEOF (QUERY (i <* SELF.items |
       'AUTOMOTIVE_DESIGN.PRODUCT_DEFINITION_CONTEXT' IN TYPEOF(i))) = 0)
        OR (SELF.role.name = 'effectivity context');
END_ENTITY;
```

Figure G-2. An entity from the AP214 model [36].

The current version of the constraint-understanding system would attempt to understand the domain rule `WR2` by extracting HLCs from the two disjuncts separately and then combining those to understand the whole expression.[5] In turn, it would try to understand the first disjunct by going inside its subexpressions, extracting HLCs from those and then combining them to understand the whole expression.

The current constraint-understanding system would then proceed to understand the domain rule `WR3` in the same way. However, since the two rules are obviously very similar, it should be more efficient if `WR2` reminded the system of `WR3` so that it could reuse the network of HLCs it extracted there (adapting it to account for the slight differences). This is much more like a true CBR approach: chunking small inferences together so that they can be retrieved as a whole, allowing a similar problem to be solved with one inferential leap next time. The issue it raises is that of storing networks of HLCs which are likely to be useful in the future as cases, so that they can be retrieved under similar circumstances in the future to suggest extracting a network of HLCs in one go. The current constraint-understanding system just retrieves a series of simple cases which suggest one HLC at a time. As discussed in the next subsection, the system would need heuristics to guide it in deciding whether an HLC network is worth retaining as a case, and when to delete retained cases that are not worth keeping any longer.

G.2.4 Case Retention and Deletion Heuristics

The current version of the constraint-understanding system does not retain cases (i.e. store its own HLC-extracting experiences as cases in memory) - the only cases in its case library are those with which it was seeded. However, when adaptation is added, it could retain the new problem and the adapted solution to the old problem as a case in memory. As discussed in the previous subsection, the system could also be extended to retain networks of HLCs as cases.

If the system simply retained every experience it ever had as a case in memory, and kept it there forever, the case library would become ridiculously large (which in turn could exacerbate the utility problem if retrieval time depended on the number of cases in the

5. The current constraint-understanding system has no HLCs to extract from these domain rules except equations (due to the equality operators), but that's beside the point.

library (see Section 3.4.1)). Two issues will thus arise when the system is allowed to modify the case library itself:

- *Case retention*: Is a new experience worth storing as a case in memory?
- *Case deletion*: Are old cases worth keeping?

There is considerable overlap between heuristics for case retention and those for case deletion, as they are both concerned with the issue of whether a case deserves to be in the case library or not (no matter whether it is there already or not). The issue of whether a case deserves to be in the case library depends on two factors (Smyth and Keane [69]):

- *Competence*: a case certainly deserves to be in the case library if it enables the system to solve problems that it would not be able to solve at all without that case;
- *Performance*: a case that allows the system to solve a problem faster than it could without that case deserves to be in the case library more than a case for the same problem which does not offer as much of a speed-up.

The simplest heuristic for case retention or deletion would be simply to ask a human. This might be acceptable for the short term, while the system is still being developed and tested, but not when the system is used in anger (because it is inconvenient and because humans are often wrong).

An effective heuristic to use would be *reusability*, which would try to predict how useful the case is likely to be in future situations. Unfortunately, the reusability of a case is very difficult to judge. There is no hard-and-fast rule, as that would require prescience. The best that could be hoped for is heuristics which tend to work well in practice. For example, cases which describe more complicated situations (e.g. complex expressions such as those in Figure G-2) are likely only to be useful in a narrow range of future situations than are cases which describe simpler situations. On the other hand, cases which deal with complex situations are likely to provide a greater pay-off when they can be used. Perhaps it would be best to break complex expressions into smaller parts, and index those parts as snippets

(Section 5.4.1 of Kolodner [39]).

An important heuristic is *novelty*, i.e. whether a case contributes knowledge that could not have been inferred easily[6] from the other cases already in the case library. The system should not bother to retain a case if it already has cases in its case library which are similar in the right respects to the new one. Alternatively, it could delete one of these similar cases in favour of the new one if the new one is better.

The most important case deletion heuristic is not to delete any case without which the system would not be able to solve a problem (unless there are lots of obscure cases which are most unlikely ever to be needed). Smyth and Keane [69] call such a case a *pivotal case*. This heuristic is likely to sanction the deletion of cases that the system derived itself but never seed cases (except for seed cases which are simply variants of each other and could thus have been derived from one another using adaptation, or a seed case which has since been surpassed by a better case for solving the same problem).

Another possible heuristic for case deletion is *recency*, i.e. delete cases that have not been used for a long time, unless there is a good reason to keep them (according to the other heuristics). The system could retain cases fairly liberally while it is processing a given model, and delete them when it has finished processing that model. For example, as we saw earlier, it is advantageous to retain the experience of processing rule `WR2` in Figure G-2 when processing the AP214 model, but it might not be worth retaining after the system has finished with AP214.

Another case deletion heuristic is *frequency*, which would consider deleting cases that have not been used often or at all (providing that deleting the case would not rob the system of any capabilities that it relies on that case for).

6. It's a matter of how quickly a problem could be solved without the case in question rather than whether it could be solved at all without the case, since the system has just derived the case itself!

Note that recency and frequency are not relevant heuristics for case retention, as they rely on patterns of access to the case which, of course, do not exist before the case is added to the case library. However, any case retention heuristic is also relevant to case deletion - in fact, cases could be tagged with values from when case retention heuristics were evaluated to decide whether to add them to the case library. These values could then be taken into consideration by case deletion heuristics.

The above discussion implies that there will be metrics for case retention and deletion - numeric values calculated for a case, which is retained or deleted if the metrics exceed certain boundaries (it is likely that the decision would be taken based on a combination of heuristics rather than just one). There is no intrinsically correct answer as to what the values of those boundaries should be - rather, they should be learned from experience. Determining these boundary values could itself be done in a case-based manner.

This section has described how the constraint-understanding system could be enhanced by improving the CBR that it uses. Having focused on the constraint-understanding system itself so far in this appendix, we will consider how it could be extended to be used as a building block for systems which require a comparative understanding of the constraints on two models (in Section G.4). Before that, however, it remains to discuss how the constraints on two models could be compared in order to assess their semantic equivalence (or lack thereof).

G.3 Comparing Constraints on Two Models

This thesis has presented a method for understanding the constraints on information models that is driven by the purpose of understanding the comparative semantics of constraints on two models. The HLCs presented in Appendix B were devised to capture the abstract semantics of constraints so that they can be compared in order to assess the semantic equivalence between the constraints on two models. The actual comparison of HLCs extracted from two models is beyond the scope of this thesis - it requires significant further research, as discussed in this section.

A system that can understand the comparative semantics of constraints on two models will need first to infer correspondences between the HLCs on the two models (Section G.3.1) and then to assess the semantic equivalence between those HLCs that correspond (Section G.3.2).

Understanding constraints comparatively across two models - as opposed to in isolation on one model - gives rise to heuristics for controlling the extraction of HLCs from the two models, heuristics which should be exploited for efficiency and intelligence. These heuristics will be discussed in Section G.3.3.

G.3.1 Inferring Correspondences between the Constraints on Two Models

It is one thing to extract HLCs from two models, but another to be able to see that two HLCs on different models correspond, and then to assess to what extent they are semantically equivalent. For example, consider extracting an 'exactly one' HLC from Figure B-1 (`gender_by_enumeration`) and an 'exactly one' HLC from Figure B-2 (`gender_by_subtyping`), as shown in Figure G-3.

```
INSTANCE exactly_one#1;
     choice_set = ["male", "female"];
     extracted_from = {defined_type#1};
     constrainee = explicit_attribute#1;
END_INSTANCE;

INSTANCE exactly_one#2;
     choice_set = {entity_type#3,entity_type#4};
     extracted_from = {oneof_expression#1};
     constrainee = entity_type#2;
END_INSTANCE;
```

Figure G-3. HLCs extracted from Figures B-1 and B-2.

The EXPRESS frames referred to in Figure G-3 are assumed to represent the following:

- `entity_type#1`: The entity `person` in Figure B-1 (although this is not actually referred to in Figure G-3).
- `explicit_attribute#1`: The attribute `person:sex` in Figure B-1.
- `defined_type#1`: The defined type `gender` in Figure B-1.
- `entity_type#2`: The entity `person` in Figure B-2.
- `entity_type#3`: The entity `man` in Figure B-2.
- `entity_type#4`: The entity `woman` in Figure B-2.
- `oneof_expression#1`: The expression `ONEOF(man,woman)` in Figure B-2.

A system which can see that `exactly_one#1` and `exactly_one#2` are semantically equivalent first needs to realise that they even correspond. This clearly requires knowledge of the structural and computational mapping[7] between the two models - in this case, that `gender_by_enumeration::person` corresponds to `gender_by_subtyping ::man` when the value for `sex` is `male`, and to `gender_by_subtyping::woman` when the value for `sex` is `female`.

A major issue will thus be how to represent the structural and computational mapping between the two models. There are existing mapping languages which will have to be considered. For example, EXPRESS-M [16] and EXPRESS-X [37] represent mappings in a procedural manner (though at a higher level of abstraction than a conventional programming language would). The KRU operators (Brown, Filer and Moosa [11]) represent mappings in a more declarative manner (with more emphasis on *what* the correspondences between schemata, entities and attributes are, rather than *how* to map between them). Whatever representational formalism is chosen, the key requirement is that it will need to describe the mapping *precisely*, with enough detail to be able to understand fully the correspondences between the two models.

7. The *structural mapping* between two models is concerned with the correspondences between entities and between attributes, whereas the *computational mapping* is concerned with value conversions (e.g. converting between degrees Celsius and degrees Fahrenheit).

A more advanced research issue would be whether the system really needs to be given a specification of the structural mapping at all, since humans can look at Figures B-1 and B-2 and just see the equivalence without any further information. A more intelligent computer system should be able to do the same, by exploiting other information such as names and comments in the models, and using those to draw on knowledge from outside the models themselves. For example, it could be inferred that `male` corresponds to `man`, and `female` to `woman`, by bringing to bear knowledge about the meanings of the words. A rich base of world knowledge would be required to solve this problem flexibly, and it would be impossible to solve the problem in its full generality because meaningless names such as `a`, `b` and `c` could be used. Section 4.4 of Mir [47] discusses name analysis in the context of electronic design, but the principles should be the same for all sorts of code understanding.

It is also possible that a system could exploit its knowledge of the constraints on two models to infer the structural mapping between them. For example, if the system extracted two very similar HLCs from two models, it might hypothesise that they correspond and use that plausible knowledge to infer information about the mapping (rather tentatively).

G.3.2 Assessing Semantic Equivalence between Corresponding Constraints

Having established that two HLCs such as those in Figure G-3 correspond (by drawing on knowledge of the structural mapping, as discussed above), the system would then need to realise to what extent they are semantically equivalent. The two HLCs may be:

- *totally equivalent* (as in Figure G-3);
- *partially equivalent* (see the discussion at the end of Section B.1 of Appendix B);
- not equivalent at all.[8]

8. Which probably means that the two HLCs do not really correspond, or one of them is nonexistent!

How to judge such semantic equivalence is an open research issue. The system could use a theory of *semantic strength* (to deduce when one constraint implies another, as discussed in Appendix B, Section B.1), and/or do it in a case-based manner. The system should have access to the standard axioms of mathematics and Boolean algebra so that it can resort to using those to prove semantic equivalence if necessary (using an intermediate canonical form as mentioned in Section G.1 would avoid the need for axioms concerning commutativity, associativity, idempotence, and other such low-level equivalences).

G.3.3 Heuristics for Controlling the Extraction of Higher-Level Constraints

Understanding the comparative semantics of the constraints on two models (as opposed to merely understanding the semantics of the constraints on one model) could give rise to heuristics for controlling the extraction of HLCs. The current constraint-understanding system extracts HLCs indiscriminately from a given model, extracting every HLC it knows how to extract from the model. However, for tasks such as transferring constraints for schema-to-schema mapping (see Section 1.3.2), it is not necessary to extract every single HLC that could be extracted from each model (and doing so could be very expensive if there are a lot of HLCs and cases). Instead, the extraction of HLCs should be driven by the needs of constraint transference.

A simple heuristic for controlling extraction in transferring constraints for schema-to-schema mapping would be to:

- Extract HLCs indiscriminately from the target model, as these are the constraints that need to be accounted for on the source model.
- For each HLC on the target model, hypothesise that there is a corresponding HLC on the source model. HLCs would then be extracted from the source model in a top-down manner, instantiating onto the augmented source model progressively lower-level HLCs from the target HLC, until they can be matched against the actual EXPRESS code to determine whether the source model really does have those constraints. This could be

achieved by applying the algorithm of Appendix D in reverse, matching HLCs to the 'solution' parts of cases and extracting lower-level HLCs from the 'problem' parts, and matching the 'problem' parts to the model itself when they are EXPRESS frames rather than HLC frames.

However, extracting HLCs bottom-up from the target model and then top-down from the source model may not be the optimum way to control extraction in all circumstances:

- There is a threat of combinatorial explosion if every single HLC on the target model is matched top-down to the source model. This would certainly be the case if the system had no idea where on the source model to look for the HLC, and had to match the HLC to many fragments of the source model. However, the system should be able to use knowledge of the structural mapping between the two models to focus its search (e.g. if the HLC was extracted from a specific entity on the target, the system would attempt to extract it top-down from the corresponding entity on the source rather than trying all entities on the source).
- It may cause the system to extract unnecessarily high-level HLCs from the target model when lower-level (intermediate) HLCs are all that are needed to see the equivalence. The most extreme example of this would be extracting any HLCs at all from syntactically identical models. It would also be unnecessary to extract an 'exactly one' HLC from a discrete 'range' HLC (as in Appendix B, Section B.2) in order to assess the equivalence between two models which have corresponding 'range' HLCs. The heuristic here is to extract HLCs only to as high a level as is necessary to see the equivalence between the constraints on the two models. Judging how high to go in extracting HLCs from the target is a difficult problem, as it cannot be judged without also taking the source model into account (making some attempt to see if there are equivalent HLCs to those so far extracted from the target model).

Another heuristic is not to bother extracting very obscure or *implicit* HLCs (such as an 'exactly one' HLC from a discrete 'range' HLC) unless it is really necessary, regardless of whether it is from the target model or the source model that those HLCs would be extracted. This seems psychologically plausible, as humans clearly would not consider a discrete 'range' HLC as 'exactly one' unless they had a good reason, such as trying to find a corresponding 'exactly one' for an 'exactly one' on the other model. The idea of implicit HLCs has much in common with Quilici's [58] implied plans (see Section 2.2.2), although HLCs are probably implicit to a lesser or greater degree, rather than either implicit or explicit. The heuristic described in this paragraph means that extraction should not be controlled entirely from the target model, but that a mixture of bottom-up and top-down extraction from both models should be done, taking into account how explicit the HLCs are.

This section has briefly discussed how the constraint-understanding system presented in this thesis could be extended to understand the comparative semantics of the constraints on two models, such as the source and target models for schema-to-schema mapping. It has discussed how the system could realise that equivalent HLCs on the two models do correspond (which requires knowledge of the structural and computational mapping between the two models), and how it could hence assess the extent to which the HLCs are semantically equivalent. The section then proposed some heuristics for controlling the extraction of HLCs from two models - focusing on the source and target models for schema-to-schema mapping - as opposed to extracting HLCs indiscriminately from each model in isolation.

The next section describes how the constraint-understanding system could be used as a building block for other systems - a constraint-transference system for schema-to-schema mapping in particular, but there are also other potential applications.

G.4 Applications of Comparative Understanding of Constraints

This section discusses some possible applications of understanding the comparative semantics of the constraints on two models, as discussed in the previous section. Section G.4.1 starts with the application which motivated the research on understanding constraints in the first place: the transference of constraints for schema-to-schema mapping. Section G.4.2 then discusses other possible applications of understanding the comparative semantics of constraints.

G.4.1 Transferring Constraints for Schema-to-Schema Mapping

To recap from Section 1.3, the task of schema-to-schema mapping is to generate automatically a program to map instances from one data repository to another. A major unsolved problem is that the mapping may violate the constraints on the target model if those constraints do not hold on the source model (see Section 1.3.2). It was therefore proposed to transfer those constraints on the target model which do not hold on the source model to the source model, which would enable the generated mapping program to check those constraints to be checked before executing the mapping itself. This transference of constraints requires a comparative understanding of their semantics, so the constraint-understanding system will be an important building block of the constraint-transference system.

Before a constraint is transferred from the target model to the source model, it needs to be understood, for two reasons:

- It may not be possible simply to copy the constraint to the source model as it is, due to structural differences between the two models. Structural differences may be as simple as different attribute names, or they may be more complicated. For example, temperature may be stored in degrees Celsius in one model but degrees Fahrenheit in the other, requiring the limits of range constraints on temperature to be converted. For another example, one model may have rich inheritance hierarchies, while the other may be flat, which could necessitate more complicated changes to the constraints to account for the structural difference.

- It may not be necessary to transfer the constraint, as it may already hold on the source, albeit in a different guise (there are several examples of this in Appendix B). It may otherwise be the case that the constraint on the target is semantically weaker (in the sense described in Appendix B, Section B.1) than the corresponding constraint on the source, and thus need not be transferred because the source constraint implies it anyway. Another possibility is that a constraint on the target may be *partially* implied by the source. Consider the target model to have a range constraint `(age >= 0) AND (age < 130)`. If one of its bounds already holds on the source, then the range constraint only needs to be partially transferred. For example, if `(age >= 21)` holds on the source, then only `(age < 130)` needs to be transferred to the source because `(age >= 0)` is implied by `(age >= 21)`.

Avoiding unnecessary transferences of constraints from the target model to the source model (unnecessary in that they are already implied by the source model) is desirable for elegance and efficiency of the generated mapping program.[9] What is really important is transferring all the constraints that *are* necessary to ensure that mappings will not violate the integrity of the target repository. A decision procedure for whether it is unnecessary to transfer a constraint is thus required to be *sound* (i.e. it never says that it is unnecessary to transfer a constraint when it is in fact necessary), but it is merely preferable that it be *complete* (i.e. that it always says that it is unnecessary to transfer a constraint when it really is unnecessary).

A major research issue will be how to actually carry out the transference of a constraint from the target model to the source model. CBR may provide the answer to this as well - recall that an extraction case associates a fragment of EXPRESS code with an HLC that characterises the abstract semantics of that fragment. If the 'solution' part (i.e. the HLC) of such a case were matched to the HLC that is to be realised on the source model, then the

9. As opposed to the efficiency of generating the mapping program, which is a one-time cost anyway, whereas the generated mapping problem would presumably be used many times in practice.

'problem' part of the case (i.e. the EXPRESS code fragment) would suggest a way of realising that HLC as EXPRESS code. This would be a bidirectional form of CBR, where the 'problem' and 'solution' parts of cases could also be used the other way round to suggest solutions to inverse problems.

The above paragraph implies that the transference of constraints from the target model to the source model involves transforming the source model by adding extra constraints to it. This is not literally what is required (although it might well be a useful intermediate step) - what is actually required for schema-to-schema mapping is to plant code into the generated mapping program to check those constraints that were transferred from the target model to the source model on instances in the source repository before they are mapped to the target repository.

G.4.2 Other Applications

Although transferring constraints for schema-to-schema mapping is the major motivation for understanding the comparative semantics of constraints, it is not the only application for which this comparative understanding of constraints could be useful. There are potentially other applications for which a constraint-understanding system could be a useful building block.

One area of potential applications concerns what happens to the constraints on models when they are transformed. For example, a highly conceptual model which is intended to express richly the semantics of a domain may be transformed into a model which is more geared to the efficient processing of instances. The STEPWISE Project [70] concerns such transformations: for example, 'flattening' inheritance hierarchies into flat collections of entities. This should preserve as much of the semantics as possible, so the constraints implied by supertype expressions and other structural constraints are written as explicit rules for the flattened model. This transformation of constraints is hardwired and supposed to be correct by construction, but it would be instructive to see to what extent they are semantically equivalent according to the HLCs that could be extracted from the original model and from the transformed (flattened) model.

Another possible application could be to understand the constraints on models written in different modelling languages. In principle, there is no reason why the method for understanding constraints presented in this thesis could not be generalised to languages other than EXPRESS, such as the Unified Modeling [sic] Language (UML)[10] [59], the Entity-Relationship (ER) model (Chapter 3 of Elmasri and Navathe [21]) or the Extended Entity-Relationship (EER) model (Section 21.1 of Elmasri and Navathe [21]). If the constraint-understanding system had cases to extract the same kinds of HLCs from these languages as it does from EXPRESS, it would be possible to assess the semantic equivalence between models which are not even written in the same language.

A constraint-understanding system could also be used as the building block for a system which checks whether executable code derived from an information model (as in Teganeanu [76]) respects the constraints on that model. This would require understanding the constraints on the model in order to verify that the executable code guarantees that every externally visible state conforms to the model.

This section has considered some potential applications for a comparative understanding of the constraints on different models. Section G.4.1 focused on the transference of constraints for schema-to-schema mapping (which was the main motivation underlying the constraint-understanding method presented in this thesis). Section G.4.2 discussed some other potential applications. These applications are concerned with understanding the constraints on models (or implementations of models) which are derived by transforming original models.

10. “Unified Modeling Language” and “UML” are trademarks of the Object Management Group.