Chapter 6: The Frames in the Constraint-Understanding System

This chapter discusses how EXPRESS [34] models, higher-level constraint (HLCs) and cases are represented as frames. The frame system, a substratum of the constraint-understanding system, was described in the previous chapter. By means of a running example, this chapter also provides insight into how the constraint-understanding system works - i.e. how it extracts HLCs from EXPRESS models using cases - though the algorithmic details are consigned to Appendix D.

Sections 6.1, 6.2 and 6.3 describe the frame representations of EXPRESS models, HLCs and cases respectively. Section 6.4 discusses the process of identifying the kinds of HLCs that are useful to support a comparative understanding constraints (c.f. Section 1.2.3) - and hence cases to extract those HLCs - by analysing EXPRESS code examples.

6.1 Frames to Represent EXPRESS Models

This section discusses the representation of EXPRESS models as frames. These kinds of frames are constructed after an EXPRESS model is parsed. The network of frames representing an EXPRESS model is effectively an *abstract syntax tree* (Section 4.3 of Louden [44]).

As mentioned in Section 5.2, the frame classes of frames which represent EXPRESS models are defined in an EXPRESS metamodel. This metamodel is included in Appendix A, Section A.1. A model of EXPRESS in EXPRESS had, in fact, already been written (Hazan [33]). However, Hazan's EXPRESS metamodel is a long-winded, conceptual model, and was not available to the Author until he had already written the one used here. The model used here is comparatively short and driven by the purpose of representing the abstract syntax of EXPRESS models - different styles of modelling are appropriate for different purposes. It is not a complete model of EXPRESS, but could easily be extended to cover additional constructs as needed.

The conversion of an EXPRESS model to a collection of frames which conforms to the EXPRESS metamodel in Section A.1 of Appendix A is quite mundane, so this thesis suppresses the details. To give an impression of the conversion, Figure 6-1 shows an EXPRESS model (as might be input to the constraint-understanding system), and Figure 6-2 shows as an FIL file the collection of frames that would be generated after parsing the model in Figure 6-1.[1]

```
SCHEMA people_model;
     TYPE gender = ENUMERATION OF (male, female);
     END_TYPE;

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

Figure 6-1. An EXPRESS model to be parsed into frames.

This section has discussed the frame representation of EXPRESS models. The next section will describe how EXPRESS model frame networks are augmented with HLC frames.

1. FIL is just a convenient notation for the frames that would be generated internally, although for the benefit of humans, an FIL file can literally be generated as well (as discussed in Appendix E, Section E.5).

```
INSTANCE defined_type#1;
     name = "gender";
     tipe = enumeration_type(["male","female"]);
          (* 'type' is a reserved word in EXPRESS,
          hence the deliberate misspelling 'tipe' *)
END_INSTANCE;

INSTANCE entity_type#1;
     name = "person";
     is_abstract = false;
     explicit_attributes = {explicit_attribute#1,
                            explicit_attribute#2,
                            explicit_attribute#3}
     domain_rules = {domain_rule#1};
END_INSTANCE;

INSTANCE explicit_attribute#1;
     name = "name";
     tipe = simple_type(string_type);
     is_optional = false;
     the_entity = entity_type#1; (* set indirectly *)
END_INSTANCE;

INSTANCE explicit_attribute#2;
     name = "age";
     tipe = simple_type(integer_type);
     is_optional = false;
     the_entity = entity_type#1; (* set indirectly *)
END_INSTANCE;

INSTANCE explicit_attribute#3;
     name = "sex";
     tipe = defined_type#1;
     is_optional = true;
     the_entity = entity_type#1; (* set indirectly *)
END_INSTANCE;

INSTANCE domain_rule#1;
     label = "age_must_be_sensible";
     expression = binary_operation#1;
     the_type = entity_type#1; (* set indirectly *)
END_INSTANCE;
```

Figure 6-2 (a). Frame representation of the model in Figure 6-1.

```
INSTANCE binary_operation#1;
     operator = and_operator;
     first_expression = binary_operation#2;
     second_expression = binary_operation#3;
END_INSTANCE;

INSTANCE binary_operation#2;
     operator = greater_than_or_equals;
     first_expression = explicit_attribute#2;
     second_expression = literal_expression#1;
END_INSTANCE;

INSTANCE literal_expression#1;
     literal_value = integer_literal(0);
END_INSTANCE;

INSTANCE binary_operation#3;
     operator = less_than;
     first_expression = explicit_attribute#2;
     second_expression = literal_expression#2;
END_INSTANCE;

INSTANCE literal_expression#2;
     literal_value = integer_literal(130);
END_INSTANCE;
```

Figure 6-2 (b). Frame representation of the model in Figure 6-1 (continued).

6.2 Frames to Represent Higher-Level Constraints

This section discusses the representation of HLCs as frames, which are extracted from the EXPRESS model frames discussed in the previous section.

HLCs were introduced in Section 4.1: they represent abstract characterisations of the implicit semantics of constraints. In this thesis, extracting HLCs from models is the realisation of understanding the constraints on those models.

In Section 4.1, HLCs were expressed informally, in natural language. What is needed for a computer implementation of understanding constraints is a way of representing HLCs formally. This section introduces a frame representation of HLCs.

Like the EXPRESS frame classes, frame classes for HLCs are defined in an EXPRESS model. Section A.2 of Appendix A contains the frame class model for HLCs.

For example, consider extracting from the model in Figure 6-1 the following HLCs (a proper subset of the HLCs that the current implementation of the constraint-understanding system is actually capable of extracting from the model in Figure 6-1):

- A discrete, inclusive, lower bound of zero on `age`;
- A discrete, exclusive, upper bound of one hundred and thirty on `age`;
- A numeric range constraint on `age`, whose bounds are the above two HLCs.

Figure 6-3 shows the HLC frames with which the EXPRESS frames in Figure 6-2 would be augmented. Note that Figure 6-3 refers to some frames from Figure 6-2, linking the HLCs to the frame representation of the model from which they are extracted.

As with the EXPRESS frames in Figure 6-2, the HLC frames in Figure 6-3 are fairly self-explanatory. However, a few slots are worthy of comment:

- `variable`: This slot holds a reference to the variable constrained by a bound or range. For example, `bound#1` in Figure 6-3 has a filler of `explicit_attribute#1` (i.e. "`age`" - see Figure 6-2) for the slot `variable`.
- `extracted_from`: This slot holds the set of frames from which an HLC was extracted (which may be EXPRESS frames, or other HLC frames). For example, `bound#1` in Figure 6-3 was extracted from `binary_operation#2` (i.e. "`age >= 0`" - see Figure 6-2). In fact, all HLCs have the `extracted_from` slot - they are all extracted from some other frames.

```
(**************************************************)
(* A discrete, inclusive, lower bound of 0 on age *)
(**************************************************)

INSTANCE bound#1;
     kind = lower;
     limit = 0;
     variable = explicit_attribute#2;
     discrete = true;
     inclusive = true;
     extracted_from = {binary_operation#2};
     conjunct_of = conjunction#2;
END_INSTANCE;

(****************************************************)
(* A discrete, exclusive, upper bound of 130 on age *)
(****************************************************)

INSTANCE bound#2;
     kind = upper;
     limit = 130;
     variable = explicit_attribute#2;
     discrete = true;
     inclusive = false;
     extracted_from = {binary_operation#3};
     conjunct_of = conjunction#2;
END_INSTANCE;

(*************************************)
(* A numeric range constraint on age,  *)
(* whose bounds are the above two HLCs *)
(*************************************)

INSTANCE range#1;
     lower_bound = bound#1;
     upper_bound = bound#2;
     variable = explicit_attribute#2;
     extracted_from = {bound#1,bound#2};
     conjunct_of = conjunction#2;
END_INSTANCE;
```

Figure 6-3. Some HLC frames which would be extracted from the model in Figure 6-1.

The other slot which warrants a mention at this point is `conjunct_of`. This is an optional slot for all HLCs, which is filled (if at all) with a frame of frame class `conjunction`. The frame class `conjunction` is a special kind of HLC whose purpose is to glue other HLCs together.

For example, the `conjunct_of` slots of all three HLC frames in Figure 6-3 refer to the frame `conjunction#2`. This frame is defined in Figure 6-4 together with another of its kind, `conjunction#1`.

```
INSTANCE conjunction#2;
     conjuncts = {bound#1,bound#2,range#1}; (* set indirectly *)
     extracted_from = {binary_operation#1};
     conjunct_of = conjunction#1;
END_INSTANCE;

INSTANCE conjunction#1;
     conjuncts = {conjunction#2}; (* set indirectly *)
     extracted_from = {entity_type#1};
END_INSTANCE;
```

Figure 6-4. Conjunction frames.

The frame `conjunction#2` is an HLC which was extracted from `binary_operation#1` (i.e. "`(age >= 0) AND (age < 130)`" - see Figure 6-2). The semantic that a `conjunction` HLC represents is that its conjuncts - which are themselves HLCs - must all be satisfied for the conjunction to be satisfied. The semantic role of 'conjunction' HLCs is further discussed in Appendix B, Section B.5. Of more importance, however, is the 'glue' role of a `conjunction` frame: it provides something on which to hang HLCs that need to be considered together. For example, a 'range' HLC is extracted from two bounds *in conjunction*.

For reasons which will become clear in Section 6.3, `conjunction:conjuncts` is defined to be an inverse slot. So `conjunction#2.conjuncts`[2] is filled with the three frames (in Figure 6-3) which have a filler of `conjunction#2` for `conjunct_of`.

2. Further to the notation introduced in Section 1.3.1, "*`frame.slot`*" refers to the attribute *`slot`* in the entity *instance `frame`*.

The frame conjunction#2 is itself a conjunct of conjunction#1. The frame conjunction#1 was extracted from entity_type#1 (i.e. the entity “person” in Figure 6-1 - see Figure 6-2).

In fact, every entity in the input EXPRESS model has a conjunction frame extracted from it - this is hardwired into the program (for once compromising the data-driven theology of HLCs). The purpose of an entity’s conjunction frame is to hold together the HLCs extracted from that entity. The reason that the conjunction frame class has to be hardwired into the program (unlike any other HLC frame class) is so that the program can automatically set the conjunct_of slot of any HLCs that are to be hung on a conjunction frame to that conjunction frame (thereby also setting the conjuncts inverse slot of the conjunction frame accordingly).

Every entity in the input model is automatically tagged with a slot higher_level_constraints, which is filled with the conjunction frame extracted from that entity. This was not shown in Figure 6-2, in order to limit the concern of Section 6.1 to EXPRESS frames, and because the higher_level_constraints slot is only set *after* parsing, during the HLC-extraction phase. Figure 6-5 shows what entity_type#1 from Figure 6-2 looks like after extraction has taken place.

```
INSTANCE entity_type#1;
     name = ”person”;
     is_abstract = false;
     explicit_attributes = {explicit_attribute#1,
                            explicit_attribute#2,
                            explicit_attribute#3}
     domain_rules = {domain_rule#1};
     higher_level_constraints = conjunction#1;
END_INSTANCE;
```

Figure 6-5. An entity frame, tagged with a conjunction frame to hold the HLCs extracted from that entity.

This section has discussed how the HLCs extracted from the input EXPRESS model are represented as frames (with no regard to the cases which are used to extract them). Section 6.4 will discuss the process of identifying the kinds of HLCs which are useful for understanding the comparative semantics of the constraints on two models, as would be needed in schema-to-schema mapping for comparing constraints and transferring them from the target model to the source model (see Section 1.3.2). Before that, it remains to cover the third broad category of frames in the constraint-understanding system: frames to represent cases.

6.3 Frames to Represent Cases

This section discusses the representation of cases (c.f. Section 3.1) as frames. It also provides some insight into how cases are used to extract HLCs from EXPRESS models.

As mentioned in Section 4.2, a *case* for the purpose of extracting HLCs associates a fragment of EXPRESS code (the 'problem' part) with an HLC which represents an abstract characterisation of the constraint semantics implied by that EXPRESS code fragment (the 'solution' part). A case can also include HLCs in its 'problem' part (e.g. for extracting a range from two bounds).

Once again, a frame class model, written in EXPRESS, is needed for cases. The case-based reasoning frame class model is included as Section A.3 of Appendix A.

As described in algorithmic detail in Appendix D, the process of extracting HLCs consists of matching fragments of the model to the 'problem' parts of cases, and instantiating the 'solution' part of each case that matches. Cases (which are represented as frames of frame class `case_frame`) hence have two slots:

- `problem`: This slot is filled with the top-level frame of a frame network representing the 'problem' part of the case - a fragment of EXPRESS code (or lower-level HLCs) to match to the prompt;[3]
- `solution`: This slot is filled with the top-level frame of a frame network representing the 'solution' part of the case - an HLC[4] to extract from the model.

So, a case consists of a frame of the special frame class `case_frame`. It is time to define precisely what we mean by the 'problem' and 'solution' parts of a case:

- The *'problem' part* of a case consists of all the frames in the transitive closure of frame references in the `problem` slot;
- The *'solution' part* of a case consists of all the frames in the transitive closure of frame references in the `solution` slot.

All these frames are considered part of the case, and will be followed recursively by the matching and extraction algorithms respectively (as described in Appendix D).

Seed cases (those that are initially entered in the case library by humans - in the current constraint-understanding system, that means *all* cases) are conventionally defined in separate FIL files.

Figure 6-6 encodes as a case the experience of extracting a discrete, inclusive, lower bound from an EXPRESS code fragment of the form `(`*`attribute`*` >= `*`limit`*`)`, where *`attribute`* is an explicit attribute (which does not need to be named in the case) of type `INTEGER`, and *`limit`* is an integer literal.

---

3. The *prompt* is a fragment of the input model (in the form of a frame network, one frame of which is submitted to the case library as a retrieval prompt). As described in Section D.1 of Appendix D, the HLC-extraction algorithm navigates the input model, submitting frames from the EXPRESS model which are known sources of HLCs as retrieval prompts.
4. There could be multiple HLCs in the 'solution' part. The system supports it, but in all the cases currently implemented, the 'solution' part consists simply of a single HLC frame (and each HLC is represented by a single frame).

```
INSTANCE case_frame#10;
     problem = binary_operation#12;
     solution = bound#11;
END_INSTANCE;

(****************)
(* Problem part *)
(****************)

INSTANCE binary_operation#12;
     operator = greater_than_or_equals;
     first_expression = explicit_attribute#12;
     second_expression = literal_expression#11;
END_INSTANCE;

INSTANCE explicit_attribute#12;
     tipe = simple_type(integer_type);
END_INSTANCE;

INSTANCE literal_expression#11;
     literal_value = integer_literal(?limit);
END_INSTANCE;

(*****************)
(* Solution part *)
(*****************)

INSTANCE bound#11;
     kind = lower;
     variable = explicit_attribute#12;
     limit = ?limit;
     inclusive = true;
     discrete = true;
     extracted_from = {binary_operation#12};
END_INSTANCE;
```

Figure 6-6. A case for extracting a discrete, inclusive, lower bound from '>='.

Note the use of a variable (`?limit`) here: this allows reinstantiation to be implemented using unification (c.f. Section 3.3.2), as opposed to replacing constants in the case (which do not contribute to the lesson that the case teaches). Variables were introduced into FIL specifically for this role, and should not be used in frames which are not part of cases. The scope of a variable is the FIL file (i.e. the case) in which it occurs.

Figure 6-6 also illustrates another point about FIL: mandatory attributes may be omitted in frames which are in the 'problem' part of a case. This is because these frames are not part of actual models, but are patterns to match frames in actual models to. Therefore, only those slots that are relevant to matching should be included in these frames. The semantics of filled and unfilled slots in cases, in terms of the behaviour of the matching algorithm (see also Appendix D, Section D.2.3), are as follows:

- Each slot which is filled in the 'problem' part of the case is required to match the corresponding slot filler in the actual EXPRESS fragment to which the case is being matched.
- Slots which are not filled in the 'problem' part of the case are ignored when the case is matched to an actual model. For example, the frame `explicit_attribute#12` in Figure 6-6 does not specify a value for the `name` slot (even though it is a mandatory attribute) because the name of the explicit attribute is not important to the lesson that this case teaches about extracting bound constraints.
- It is also conceivable that a case would need to specify that a slot must be unfilled in the actual model (i.e. the corresponding slot in the model must have a null filler for the match to succeed). Because FIL represents unfilled slots by omission, and the omission of a slot in a case means that the slot will be ignored in matching, FIL has been extended with a means of explicitly specifying a filler of null for a slot. The syntax for this is "`*slot* = null;`".

The frames `binary_operation#12`, `explicit_attribute#12` and `literal_expression#1` constitute the 'problem' part of this case, because they are in the transitive closure of frame references from the `problem` slot of `case_frame#10`. The frame `bound#11` constitutes the 'solution' part.

Note that the numbering convention here is adopted for clarity: the frames in Figure 6-6 are actually called `case_frame#1`, `binary_operation#1`, `explicit_attribute#1`, `literal_expression#1` and `bound#1` in the real FIL file this case is written in.[5] They have been renumbered for the purposes of this chapter in order to avoid confusion with frames of those names in Sections 6.1 and 6.2.[6] Here, the frames in the case are numbered according to their correspondence with the frames in the (augmented) model presented in those sections: `literal_expression#11` corresponds to `literal_expression#1`, `binary_operation#12` corresponds to `binary_operation#2`, etc.

Whenever the ‘problem’ part of the case shown in Figure 6-6 is matched to a fragment of code in an actual model, the variable `?limit` gets bound to whatever is at the corresponding position in the model (any further occurrences of the variable in the ‘problem’ part are required to match whatever the variable was bound to at its first occurrence). If the ‘problem’ part of the case is matched successfully, then the ‘solution’ part is instantiated (for the augmented model) and all occurrences of each bound variable are replaced with whatever in the model that variable was bound to during matching.

The process of matching variables to constants described above is known as *unification*, and is widely used in rule-based systems (Section 4.1 of Forbus and de Kleer [28]) and resolution theorem-proving in predicate logic (Section 5.4 of Rich and Knight [64]).

As an example, imagine matching the case in Figure 6-6 to the EXPRESS code fragment “`(age >= 0)`” in Figure 6-1 (`binary_operation#2` in Figure 6-2). The variable `?limit` would get bound to `0` when the matching process encountered its first occurrence.[7] Because this EXPRESS code fragment does indeed match the ‘problem’ part of the case, the ‘solution’ part of the case is instantiated for the model: a new instance of

5. The FIL files for all cases currently implemented are included as Appendix C.
6. Figure 6-6 still constitutes a valid FIL file - any natural numbers will do for instance numbers. The original numbering scheme does not technically cause a problem either - recall that the joint uniqueness of a frame’s frame class and external instance number only applies within the boundaries of an input FIL file.
7. In this case, the first occurrence of `?limit` (in `literal_expression#11`) happens to be its only occurrence in the ‘problem’ part of the case. Any further occurrences of `?limit` in the ‘problem’ part would be matched as though they were the constant 0.

`bound` is created (namely `bound#1` in Figure 6-3), copying the slot fillers from `bound#11`. The variable `?limit` is replaced with its binding, `0`, and the frame references that were matched in the 'problem' part (`binary_operation#12` and `explicit_attribute#12`) are replaced with the frames which they matched in the input EXPRESS model (`binary_operation#2` and `explicit_attribute#2`).

The HLC `bound#2` in Figure 6-3 can be extracted in a similar way, using a case whose 'problem' part matches "`(age < 130)`" (see Appendix C, Section C.2.7).

The HLC `range#1` in Figure 6-3 can be extracted using the case shown in Figure 6-7 (again, the frames have been renumbered to avoid confusion with other frames in this chapter, so that a frame in Figure 6-7 with instance number *n* matches the corresponding frame with instance number (*n* - 20) in the augmented model).

The HLCs `bound#1`, `bound#2` and `conjunction#2` have to be extracted before the case in Figure 6-7 can be used to extract `range#1`, so that they can match `bound#21`, `bound#22` and `conjunction#22` respectively, in the case. Figure 6-8 shows the case that would be used to extract `conjunction#2`.

Note that when `conjunction#2` is first created (by making a copy of `conjunction#32` after successfully matching the case in Figure 6-8), it starts with only one slot (`extracted_from`) being filled, and yet we know from Figure 6-4 that `conjunction#2` ends up having two more slots filled:

- `conjuncts` is an inverse slot which is set indirectly when the `conjunct_of` slots of other HLC frames are filled with `conjunction#2`;
- As explained in Section 6.2, `conjunct_of` is a special slot for HLC frames which is filled automatically (this is hardwired into the program) in order to hang HLCs on a `conjunction` frame. We can now see why the filling of this slot has to be hardwired. A case does not have enough context to fill `conjunct_of` simply by specifying a filler

for it in the 'solution' part. A case cannot 'see' the `conjunction` frame on which the HLC it suggests to extract is to be hung (although in the case of Figure 6-7, it can, because the `conjunction` frame on which to hang the `range` HLC is the same conjunction frame that is matched in the 'problem' part). So the system keeps track of `conjunction` frames as it navigates the model extracting HLCs, and sets the `conjunct_of` slots of the HLCs that it extracts, in order to hang them on the appropriate `conjunction` frame.

```
INSTANCE case_frame#20;
     problem = conjunction#22;
     solution = range#21;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE conjunction#22;
     conjuncts = {bound#21,bound#22};
END_INSTANCE;

INSTANCE bound#21;
     kind = lower;
     variable = ?variable;
END_INSTANCE;

INSTANCE bound#22;
     kind = upper;
     variable = ?variable;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE range#21;
     lower_bound = bound#21;
     upper_bound = bound#22;
     variable = ?variable;
     extracted_from = {bound#21,bound#22};
     conjunct_of = conjunction#22;
END_INSTANCE;
```

Figure 6-7. A case for extracting a range constraint from the conjunction of two bounds.

```
INSTANCE case_frame#30;
     problem = binary_operation#31;
     solution = conjunction#32;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE binary_operation#31;
     operator = and_operator;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE conjunction#32;
     extracted_from = {binary_operation#31};
END_INSTANCE;
```

Figure 6-8. A case for extracting a conjunction from an AND operation.

This section has presented the frame representation of cases. It has also provided some insight into how cases are used to extract HLCs (though the algorithmic details are consigned to Appendix D). The next section discusses the process of identifying the kinds of HLCs (and hence cases to extract those HLCs) that are needed to support a particular constraint-understanding task.

6.4 Identifying Higher-Level Constraints

The important question which will arise when the constraint-understanding system presented in this thesis is used to support a task which requires an understanding of constraints is: what kinds of HLCs are needed to capture the knowledge about constraints which is needed for that task?

For example, the underlying motivation for the research which led to this thesis is to support an understanding of the comparative semantics of constraints on two models (see Section 1.2.3), which would be needed for such tasks as transferring constraints for schema-to-schema mapping (as explained in Section 1.3.2).

For this task, the kinds of HLCs that are needed are those that enable abstract similarities between constraints which are expressed differently on the two models to be seen, so that the semantic equivalences and differences between the constraints on the two models can be assessed. Section 1.2.3 used the example that the models in Figures 1-3 and 1-4 both have the abstract semantics that a person is constrained to be either male or female but not both (i.e. constrained to have exactly one of two properties), even though one model expressed this constraint using an enumeration type, whereas the other model expressed this constraint using a subtype hierarchy.

For a particular constraint-understanding task, the HLCs that are needed must be identified (and written in the HLC frame class model (Appendix A, Section A.2)). The approach taken to identifying HLCs is to analyse sample EXPRESS models by hand, thinking about how the constraints implied by fragments of the models could be characterised abstractly. In particular, identifying the HLCs that are useful to support a comparative understanding of constraints on two models involves analysing different ways of expressing the same constraint (as in Figures 1-3 and 1-4), in order to invent an HLC that captures the abstract similarity.

Cases are a natural by-product of this HLC-identification process, because each experience of inventing an HLC from a fragment of EXPRESS code can be encoded as a case to extract that HLC from such a fragment. The case library can then be seeded with these cases.

Appendix B presents an ontology of HLCs (and cases for extracting them), by analysing a series of worked examples. It is omitted from the main text of this thesis in order to keep the length down, but it is well worth a read at this juncture. This section has distilled the important principles; Appendix B backs them up with examples.

This chapter has described the frames used to represent EXPRESS models, HLCs and cases. It has also provided some insight into how the constraint-understanding system works, particularly in Section 6.3, which showed how cases are used to extract HLCs from EXPRESS models. The actual algorithmic details of doing so are described in Appendix D. Some implementational issues regarding the languages and tools that are used to implement the constraint-understanding system are discussed in Appendix E. Section 6.4 discussed the (manual) process by which HLCs (and hence cases to extract them) are identified, referring to Appendix B for examples. Chapter 7 will conclude this thesis by summarising it, presenting some experimental results of running the constraint-understanding on some ‘real’ EXPRESS models, discussing the major contributions of the thesis, and discussing the potential for future work.