Chapter 5:
The Frame System

This chapter and the next chapter concern the system for understanding constraints on information models which was introduced in Chapter 4. As mentioned there, this understanding of constraints is realised by extracting higher-level constraints (HLCs) from models written in EXPRESS [34]. A case-based reasoning (Chapter 3) approach is taken, in which cases suggest HLCs to extract from particular fragments of EXPRESS code. Chapter 6 will show how EXPRESS models, HLCs and cases are represented in a uniform way as frames, and how cases are used to extract HLCs from EXPRESS models. Before that, however, this chapter introduces the frame system and discusses languages for frame classes (Section 5.2) and frame instances (Section 5.3).

5.1 Introduction to Frames

"A frame is a collection of attributes (usually called slots) and associated values (and possibly constraints on values) that describe some entity in the world" (Section 9.2 of Rich and Knight [64]).

The word *frame* means different things to different people in the artificial intelligence community, but the above quotation is close to the sense used in this thesis. It should be noted that this thesis takes a very pragmatic attitude to frames, the purpose being to represent knowledge in a way that is flexible enough to support reasoning processes that are generic enough not to have to be hardwired for every type of frame in the system.

For the purposes of this thesis, a *frame* can be thought of conceptually as a set of slot-filler (or attribute-value) pairs. Figure 5-1 shows a simple example of a frame.

```
person#1
  name = "Andrew Broad"
  age = 23
```

Figure 5-1. A simple example of a frame.

This frame is an instance of the *frame class* `person`, and has an instance number of `1` which distinguishes it from any other instances of `person`. It has a filler (value) of `"Andrew Broad"` for the slot (attribute) `name`, and a filler of `23` for the slot `age`.

A slot filler in a frame can also be a link (reference) to another frame, as Figure 5-2 illustrates. The slot `department` in the frame `person#2` is filled with a reference to the frame `department#1` (an instance of the frame class `department`).

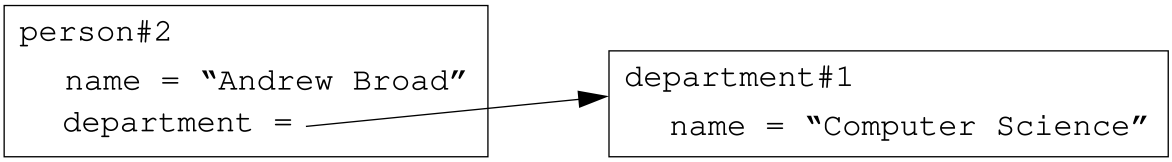

Figure 5-2. A frame linked to another frame.

Frames serve as navigation structures for the constraint-understanding system. The difference between frames and the way data structures are ordinarily written in computer programs is that frame classes and slots, as well as instance frames and slot fillers, are explicitly represented as data, rather than being hardwired into the program code. The system is thus operating in a data-driven, object-centric environment. This environment is suitable for generic algorithms (in particular, for case matching), which can be coded independently of the particular frame classes of the frames, and the slots of the slot fillers, on which those algorithms will operate.

The features of the frame system will be detailed in the following two sections, which discuss languages for representing frame classes and frame instances respectively.

5.2 The Frame Class Language (EXPRESS)

A language is needed for describing the frame classes of which frames are instances, and the slots which frames of a particular frame class should have. For example, if there are instances of `person` (as in Figure 5-1) in the frame system, then the system must be told that there is a frame class `person` which has slots `name` (which should be filled with a string) and `age` (which should be filled with an integer).

It was decided to use EXPRESS itself as the frame class language, since describing frame classes is intuitively similar to information modelling (see Section 1.2.1) and frame classes can naturally be specified in EXPRESS. It is convenient to use EXPRESS as opposed to inventing a separate frame language or using one of the existing ones such as KL-ONE (Woods and Schmolze [88]), KIF (Fikes, Cutkosky et al [25]), or the case representation language CASUEL (Bergmann [5]).

Using EXPRESS as the frame class language, frame classes are defined as entities, and slots are defined as attributes. Figure 5-3 shows a frame class model for the frames in Figures 5-1 and 5-2.

```
SCHEMA frame_class_model;
     ENTITY person;
          name : STRING;
          age : OPTIONAL INTEGER;
          department : OPTIONAL department;
     END_ENTITY;

     ENTITY department;
          name : STRING;
     END_ENTITY;
END_SCHEMA;
```

Figure 5-3. A model for the frame classes in Figures 5-1 and 5-2.

Uniformity within the constraint-understanding system is a major advantage of using EXPRESS itself as the frame language. The frame class definition file can be read in using the same parser as the EXPRESS models on which the system is to understand the constraints. The fact that EXPRESS is a standard language makes the constraint-understanding system more open to integration with other systems (although the fact that

EXPRESS is still evolving could cause some difficulties in this regard). Moreover, when it comes to transferring the constraints from a target model to a source model for schema-to-schema mapping and hence automatically generating mapping programs (see Section 1.3), the mapping programs will be able to be tested using frames which are instances of the entities in the input models.

EXPRESS provides the constructs to support almost all the features desired of a frame system (especially the inheritance mechanism implied by subtyping), which justifies pragmatically its use as the frame class language. The only obvious weakness is that EXPRESS does not support default values for attributes. Fortunately, this facility was not essential for the constraint-understanding system - it just limits the suitability of EXPRESS as a universally applicable frame language.

EXPRESS is actually more sophisticated than is necessary to represent frame classes, so the frame system and its frame class models are restricted to a sensible subset of EXPRESS - the frame system would become too complicated for its own good if it had to deal with the full expressive power of EXPRESS! In fact, the use of EXPRESS as a frame class language is not so much *information modelling* (where the emphasis is on communicating the semantics of a domain) as *data modelling* (which is geared more towards the implementation of instances).

When an EXPRESS model is parsed, it is converted into frames which represent the syntactic components of that model (as described in Section 6.1). Since EXPRESS itself is being used as the frame class language, the definition of the EXPRESS frame classes constitutes an *EXPRESS metamodel*: that is, a model of EXPRESS, written in EXPRESS itself.[1] This has the added advantage that frame classes can be represented as frames in their own right. This means that a system which needs to reason about frame classes as well as their instances could navigate frame classes as easily as the 'instance' frames. The current

1. This model is included as Section A.1 of Appendix A.

constraint-understanding system does not do such introspective reasoning itself, but since frame class frames are created anyway when the EXPRESS metamodel is parsed, the hook is there should introspective reasoning be required in the future.

So, EXPRESS is used here as a language for frame *classes*. What is also needed is a language for frame *instances*.

5.3 The Frame Instance Language (FIL)

A language is needed for writing individual frames that are instances of frame classes (as written in EXPRESS). Figures 5-1 and 5-2 are very close to what is required, but a purely textual language is needed so that instance frames can be typed in by humans (in particular, when seeding the case library). This section describes the *Frame Instance Language (FIL)*, which has been invented by the Author for this purpose.

An alternative to inventing FIL would have been to use an existing EXPRESS instance language, namely the ISO 10303-21 Physical File Format (which shall henceforth be referred to as *Part 21*), which is described in Section 2.3 of Kasadha [38]. However, this format was rejected in favour of FIL for cosmetic reasons, the most important of which being that FIL represents slot fillers as an unordered set of attribute-value pairs rather than an ordered list of values without the attribute names. Furthermore, some extensions to the language are needed for a case representation which supports the desired case matching process, as will be discussed here and in Section 6.3.

FIL is designed to resemble the syntax of EXPRESS, for consistency of style. A collection of frames is written in an FIL file (usually, a single seed case is written in a single FIL file), which can be parsed into an internal representation as frames in the computer's memory. An internal frame network can also be written out as an FIL file.

Figure 5-4 shows what the frames in Figures 5-1 and 5-2 look like when written in FIL.

```
INSTANCE person#1;
     name = ″Andrew Broad″;
     age = 23;
END_INSTANCE;

INSTANCE person#2;
     name = ′Andrew Broad′;
     department = department#1;
END_INSTANCE;

INSTANCE department#1;
     name = ″Computer Science″;
END_INSTANCE;
```

Figure 5-4. A sample FIL file.

The noteworthy points about Figure 5-4 are:

- Each frame instance names its frame class (i.e. the entity in the frame class EXPRESS model of which it is an instance), and has an *instance number*. For example, `person#1` means instance number one of the frame class `person`. Instances of different frame classes may have the same instance number, as may instances of the same frame class in different files (`person#1` in one FIL file is not the same frame as `person#1` in another FIL file). When a frame instance is parsed, it is assigned an *internal instance number*, which is not necessarily the same as its external instance number.[2] Every instance of a given frame class must have an internal instance number which is different from the internal instance numbers of all the other instances of that frame class, even if the instances come from different FIL files (unlike external instance numbers). This is because the internal frames usually do come from multiple FIL files (and from other sources, such as when EXPRESS models are parsed, and the frames that the system derives itself), and the frame system does not keep track of where the internal frames

2. Whereas when frames in memory are written out in FIL, their internal instance numbers are simply used as the external instance numbers.

come from.

- Each instance specifies each of its slot fillers in the form "*`slot = filler;`*". The order in which slot fillers are specified is not significant. Only slots which are declared in the instance's frame class may be filled. All slots which are specified in the frame class as mandatory attributes should be filled, except in frames which are in the 'problem' part of a case (as will be explained in Section 6.3). Unfilled slots are encoded by omission (e.g. `department` in `person#1`, and `age` in `person#2`).

- String literals are surrounded by either single quotes (e.g. `'Andrew Broad'`) or double quotes (e.g. `"Andrew Broad"`). Single quotes and double quotes are interchangeable (although they must be the same in any particular string literal).

- Where a slot filler is a reference to a frame, it is written in the form "*`frame_class#instance_number`*", as in frame definitions. An example of a frame reference is in the slot filler "`department = department#1`" in `person#2`. The frame referred to must be defined in the same file, if that file is to be accepted by the current FIL parser.[3]

The above style is similar to Part 21 except for the following significant differences:

- Slot fillers in FIL specify the names of slots as well as their fillers. This means that slot fillers can be given in any order, and unfilled slots are simply omitted. This is clearer than Part 21, which specifies attribute values without the names of attributes. Therefore, in Part 21, attribute values must be given in the order that the attributes are declared in the entity definitions in the EXPRESS model to which the instances conform, and null values are denoted by a special symbol (`$`).

3. If the FIL parser allowed a frame referred to in one file to be defined in another file, it would have to force the frame class and instance number of every frame to be jointly unique across all files - a more inconvenient restriction than having to define in the same file all the frames in the transitive closure of a frame's references. Another alternative would be to allow frames defined outside a file to be referred to by qualifying them with a filename, e.g. "`person#1@person.fil`".

- In FIL, frames are named as "*`frame_class#instance_number`*", both in definition and in reference. In Part 21, instances are simply given an instance number for reference, so `person#1`, `person#2` and `department#1` would be referred to as `#1`, `#2` and `#3`. Their entity types are named only in definition.

For example, `person#1` would be defined in Part 21 as:

```
#1 = PERSON('Andrew Broad', 23, $);
```

- which looks less elegant than its definition in Figure 5-4, in the Author's opinion.

The following subsections cover miscellaneous features of FIL which have not been covered in the above discussion.

5.3.1 Aggregation Slot Fillers in FIL

Slot fillers which are aggregations are written as comma-separated lists, enclosed by curly brackets when the slot type is `SET` or `BAG` (e.g. `{1,2,3}`), and by square brackets when the slot type is `LIST` or `ARRAY` (e.g. `[1,2,3]`). The rationale for this convention is just human readability.

5.3.2 Enumeration Slot Fillers in FIL

Enumeration type slot fillers are written simply as the symbols in the enumeration type (e.g. `male` and `female` are the possible fillers for `ENUMERATION OF (male,female)`).

5.3.3 Defined Type Slot Fillers in FIL

Defined type slot fillers can be written in the form "*`type`*`(`*`filler`*`)`", where *`type`* is the name of the defined type and *`filler`* conforms to the underlying type. The type name and parentheses are only obligatory for defined types inside select types. Figure 5-5 shows a frame class model and Figure 5-6 a frame instance file to illustrate this point.

```
SCHEMA defined_types;
     TYPE s = SELECT (t,u);
     END_TYPE;

     TYPE t = ENUMERATION OF (x,y);
     END_TYPE;

     TYPE u = ENUMERATION OF (v,w,x);
     END_TYPE;

     ENTITY z;
          a : s;
          b : t;
     END_ENTITY;
END_SCHEMA;
```

Figure 5-5. A frame class model for Figure 5-6.

```
INSTANCE z#1;
     a = t(x);
     b = x;
END_INSTANCE;
```

Figure 5-6. An FIL file to illustrate defined type instances.

In "`a = t(x)`", the type name and parentheses are needed to tell the frame system that the filler is of underlying type `t`. Writing this slot filler as "`a = x`" would be ambiguous, because it could be either the `x` in `t` or the `x` in `u`. Even "`a = y`" is not allowed, because the rule in FIL is that the type name and parentheses are mandatory for defined types in a select type. However, "`b = x`" *is* legal, because the system just looks up the slot type of `b` in the frame class to determine that the filler is of type `t`. These slot fillers could be written as "`a = s(t(x))`" and "`b = t(x)`", but that would be unnecessarily long-winded.

The idea of using type names and parentheses only when it is necessary to resolve ambiguity comes from Part 21, which even allows underlying type names in select types to be dropped when there is no ambiguity as to *their* underlying types, as determined by some rather complicated rules (mentioned in Section 2.3.2 of Kasadha [38]).

5.3.4 Subtype Instances in FIL

FIL's approach to subtyping is to create an instance of the subtype and inherit the slots from its supertypes,[4] using only single inheritance. This means that the frame class model is not allowed to use the full power of EXPRESS's subtyping, which was not essential for the constraint-understanding system's frame class models anyway. This restriction imposed on subtyping in FIL avoids the need for complex instances (composite instances of more than one entity - see Section 1.2.1.2). Figures 5-7 and 5-8 illustrate subtyping in FIL.

```
SCHEMA subtyping;
     ENTITY person;
          name : STRING;
          nationality : STRING;
     END_ENTITY;

     ENTITY tennis_player SUBTYPE OF (person);
          ranking : INTEGER;
     END_ENTITY;
END_SCHEMA;
```

Figure 5-7. A frame class model for Figure 5-8.

```
INSTANCE tennis_player#1;
     name = ”Monica Seles”;
     nationality = ”United States of America”;
     ranking = 1;
END_INSTANCE;
```

Figure 5-8. An FIL file to illustrate subtyping in FIL.

A more orthodox EXPRESS instance language would have to define `tennis_player#1` as a composite instance of `person` and `tennis_player`, but FIL allows us to avoid complex instances. FIL (and hence also the internal frame system) could be extended to

4. Both its direct supertype and any indirect ancestors that it may have.

deal with complex instances if necessary (e.g. to support multiple inheritance).

Part 21 defines complex instances by grouping together the instances of each entity in the supertype/subtype configuration. A complex instance can then be referenced by the instance number associated with the instance group. For example, the frame in Figure 5-8 would be defined in Part 21 as:

```
#1 = ( PERSON('Monica Seles', 'United States of America')
       TENNIS_PLAYER(1)
     );
```

Complex instances exacerbate the drawback in Part 21 of having to specify attribute values in a particular order due to the omission of attribute names. In the presence of multiple inheritance, the rules for determining the right order are complicated and contentious, as mentioned in Section 2.3.3 of Kasadha [38]. A future version of FIL which supported complex instances would not have these difficulties, because FIL avoids having to constrain the order of slot fillers by including the names of slots in slot fillers.

5.3.5 Inverse Slot Fillers in FIL

All the slots we have seen so far have been explicit attributes in the EXPRESS frame class models. FIL also supports *inverse slots*, i.e. it enables inverse attributes to be filled. Inverse attributes represent existence constraints (as explained in Section 1.2.1.2). They also provide a convenient mechanism for backpointers in FIL.

Figure 5-9 illustrates the idea. It shows a model of students and their supervisors. For the sake of simplicity, supervision is modelled as a one-to-one relationship between students and members of staff.

```
SCHEMA student_supervisor_model;
     ENTITY student;
          name : STRING;
          supervisor : staff_member;
     END_ENTITY;

     ENTITY staff_member;
          name : STRING;
     INVERSE
          supervisee : student FOR supervisor;
     END_ENTITY;
END_SCHEMA;
```

Figure 5-9. A model of students and supervisors.

In FIL, inverse slots are filled automatically, to keep their values coherent with the values of their `FOR` attributes.

Consider Figure 5-10 as an FIL file which will be *input* to the frame system (i.e. parsed into the internal representation of frames).

```
INSTANCE student#1;
     name = "Andrew Broad";
     supervisor = staff_member#1;
END_INSTANCE;

INSTANCE staff_member#1;
     name = "Nick Filer";
END_INSTANCE;
```

Figure 5-10. An FIL file to be input to the frame system.

When the frame system reads the FIL file in Figure 5-10, it will automatically fill the `supervisee` slot of `staff_member#1` with `student#1`. Figure 5-11 shows the frames after this has been done. It is expressed in FIL again, but Figure 5-11 should be thought of as a file which is *output* from the frame system. In fact, the current system literally outputs frames in this way for humans to browse (usually augmented with HTML tags, as described in Appendix E, Section E.5). It indicates with comments those slot fillers which were not given in the input FIL file, but were set automatically because they are inverse slots.

```
INSTANCE student#1;
     name = "Andrew Broad";
     supervisor = staff_member#1;
END_INSTANCE;

INSTANCE staff_member#1;
     name = "Nick Filer";
     supervisee = student#1; (* set indirectly *)
END_INSTANCE;
```

Figure 5-11. The frames in Figure 5-10, after being processed by the frame system.

Figure 5-9 is a minimal example of a frame class model with inverse slots, and encodes a one-to-one relationship. The idea can be generalised as follows:

- A one-to-many relationship can be encoded by making the inverse attribute an aggregation. For example, each staff member could supervise many students (as in "`INVERSE supervisees : SET [0 : ?] OF student FOR supervisor`").
- A many-to-one relationship can be encoded by making the attribute referred to in the `FOR` clause of the inverse attribute an aggregation. For example, each student could be supervised by many staff members (as in "`supervisors : SET [0 : ?] OF staff_member`").
- A many-to-many relationship can be encoded by combining the above two points, so a student could have multiple supervisors, who could each supervise several students.

FIL can handle all of the above cases.

This thesis omits the details of how the frame system handles inverse slots, suffice it to say that it is necessary to think *procedurally* in order to understand inverse slots (whereas all other aspects of FIL can be understood purely declaratively). The procedural nature of inverse slots makes them rather awkward and inelegant, but they are very useful for automatically setting backpointers, especially in cases (see Section 6.3), where the filling of slots can be data-driven (as opposed to hardwired) to a greater extent than would be possible without inverse slots.

5.3.6 FIL Grammar

The complete grammar of FIL is shown in Figure 5-12, written in Extended Backus-Naur Form (EBNF) (Chapter 4 of Louden [44]). FIL has some features which are peculiar to writing cases, namely variables (identifiers with the prefix '`?`') and explicitly specifying a filler of null for a slot. A discussion of these features is deferred until Section 6.3.

Figure 5-12 uses the following EBNF notation:

- `(`*x*`)*`: zero or more repetitions of *x*;
- `(`*x*`)+`: one or more repetitions of *x*;
- `(`*x*`)?`: zero or one repetitions of *x*;
- *x* `-` *y*, where *x* and *y* are alphanumeric characters: all the characters in the range *x* to *y* inclusive;
- `~`*x*: anything except *x*;
- `'`*x*`'`, where *x* is a non-alphanumeric character: the literal *x* (to avoid confusion with metacharacters);
- `'\''`: the literal `'`.

Like EXPRESS, FIL is case-insensitive, and ignores white space (spaces, tabs, newlines and carriage returns) and comments beginning with '`(*`' and ending with '`*)`'.

```
<start>        ::= (<instance>)*

<instance>     ::= INSTANCE <identifier> ‘#’ <number> ‘;’
                      (<slot>)*
                   END_INSTANCE ‘;’

<slot>         ::= <identifier> ‘=’ <value> ‘;’

<value>        ::= [‘+’ | ‘-’] <number>
                 | <real>
                 | <string>
                 | <logical>
                 | null
                 | ‘?’ <identifier>
                 | <identifier> ‘#’ <number>
                 | <identifier> ‘(’ <value> ‘)’
                 | <identifier>
                 | ‘{’ (<value-list>)? ‘}’
                 | ‘[’ (<value-list>)? ‘]’

<number>       ::= (0 - 9)+

<real>         ::= (‘+’ | ‘-’)? <number> ‘.’ (<number>)?
                      (E (‘+’ | ‘-’)? <number>)?

<string>       ::= (‘\’’ (~’\’’)* ‘\’’)
                 | (‘"’ (~‘"’)* ‘"’)

<logical>      ::= TRUE | FALSE | UNKNOWN

<identifier> ::= (A - Z) (A - Z | 0 - 9 | ‘_’)*

<value-list> ::= <value> (‘,’ <value>)*
```

Figure 5-12. A grammar for FIL.

This section has described FIL, a language for the external representation of frame instances. Its main uses in the constraint-understanding system are for seeding the case library, and also for outputting the frame networks in the computer’s memory for humans to peruse (as discussed in Appendix E, Section E.5). Moreover, it provides a convenient notation for frames, which will be used in the next chapter.

This chapter has introduced the frame system, a substratum of the constraint-understanding system that provides an environment in which EXPRESS models, HLCs and cases can be represented as frames (as described in the next chapter). This frame-based representation is intended to support generic reasoning processes (as explained in Section 5.1) - for case matching in particular.

Frames allow knowledge to be represented in a uniform way, both internally and externally. This chapter has discussed the use of EXPRESS and FIL as textual languages for the external representation of frames classes and frame instances respectively. This chapter has also provided insight into the internal frame system. The discussion in this chapter was independent of the particular frames used in the constraint-understanding (frames to represent EXPRESS models, HLCs and cases) - these will be described in the next chapter.