Appendix D:
Algorithmic Details

This appendix describes the algorithmic details of the constraint-understanding system. The constraint-understanding system was introduced in Chapter 4, which presented the architecture of the system, focusing on its external functionality. The key functionality of the system is the extraction of higher-level constraints (HLCs) from EXPRESS [34] models. The HLCs embody an understanding of the abstract semantics of the constraints on those models. Chapters 5 and 6 followed on to describe a frame system,[1] in which EXPRESS models, HLCs and cases can be represented. A case-based reasoning system for extracting HLCs from models using cases is built on top of the frame system, and its algorithmic details are the subject of this appendix.

Section D.1 describes how the extraction of HLCs is controlled at the top level, including the issue of where on a model to look for HLCs, how to submit HLCs as retrieval prompts for the extraction of further HLCs, the issue of case library organisation and the related issue of case retrieval. There are two steps to using the retrieved cases to extract HLCs:

- matching the 'problem' parts of cases to the augmented model[2] (Section D.2);
- actually extracting the HLCs suggested by the 'solution' parts of those cases that successfully match (Section D.3).

Section D.2 describes the matching of cases to the model (and to HLCs already extracted). The matching process has two purposes:

- to decide whether or not to extract the HLC suggested by each case;
- to determine the correspondences between the case's components and appropriate parts of the model.

---

1. The frame system was described only in terms of its external functionality. Its algorithmic details are suppressed because, as intricate and complicated as they are, they are not really important to this thesis.
2. The term *augmented model* refers to the frame representation of the input model plus whatever HLCs have been extracted so far.

Section D.3 describes the extraction (instantiation onto the model) of HLCs following a successful match (this step makes use of the correspondences found during matching).

D.1 The Top-Level Extraction Algorithm

This section describes the top-level algorithm for controlling the extraction of HLCs from a given model. This algorithm navigates the frames representing the input EXPRESS model, using those frames from which HLCs could arise (as listed in Section D.1.1) as retrieval prompts to search the case library. The retrieved cases' 'problem' parts are matched to the prompt frame (as will be described in Section D.2), and for each case that matches successfully, the HLC from the 'solution' part of the case is extracted from the input model by copying it, substituting corresponding references to the input model which were kept track of during the matching (Section D.3 will describe this step).

It should be noted that the current constraint-understanding system takes a *bottom-up* approach to extraction (c.f. Section 2.2.2). That is, it starts from the input EXPRESS model, extracts whatever HLCs it can from that (e.g. bounds), and can then extract further HLCs (e.g. a range). A *top-down* approach would hypothesise that a particular HLC could be extracted from the input model, and would instantiate it (e.g. the range) and any lower-level HLCs needed to extract it (i.e. two bounds), until it could verify directly against the input model that it is valid to extract the HLC that was hypothesised to begin with. In a manner analogous to rule-based systems (Section 6.3 of Rich and Knight [64]), the bottom-up approach is *forward chaining* from the 'problem' parts of cases to the 'solution' parts, whereas the top-down approach is *backward chaining* from the 'solution' parts of cases to the 'problem' parts.

A top-down approach to extraction is not viable for the current constraint-understanding system, because it just extracts HLCs from one model in isolation for the sake of it, without having a goal to drive it. However, future research will investigate top-down extraction in the context of understanding the comparative constraint semantics of two models (c.f. Section 1.2.3), as discussed in Appendix G, Section G.3.

This section is divided into three subsections:

- Section D.1.1 describes how the input EXPRESS model is navigated, submitting known sources of HLCs to the case library as retrieval prompts;
- Section D.1.2 describes how HLCs already extracted from the model are submitted to the case library as retrieval prompts in order to extract further HLCs;
- Section D.1.3 describes how the case library handles these retrieval prompts, matching them to the 'problem' parts of cases and extracting the HLCs suggested by the 'solution' parts of matching cases.

D.1.1 Navigating the Input EXPRESS Model

The top-level extraction algorithm iterates over the entities in the input EXPRESS model. For each entity, it submits each of the following potential sources of HLCs to the case library as a retrieval prompt (i.e. it submits frames from the model to match to the `problem` slots of cases):

- The entity itself (an `entity_type` frame), which is matched to cases which have entity types as the top-level frames in their `problem` slots. An `entity_type` frame (see Appendix A, Section A.1) holds information as to whether or not the entity is abstract and information about its subtypes and supertypes, as well as the following information.[3]
- Each explicit attribute (i.e. the frames from the `explicit_attributes` slot). The attribute declarations in an entity imply constraints about optionality, types, and any

3. The current system does *not* submit supertype expressions (i.e. the `SUPERTYPE` clause) and supertypes (i.e. the `SUBTYPE OF` clause) as separate retrieval prompts, as the cases which deal with them use the entity itself as the top-level frame in their 'problem' parts. For example, cases which deal with supertype expressions often need to encode whether the entity is an abstract supertype or not, as entities which are abstract supertypes are constrained not to be instantiated without their subtypes.

constraints attached to those types.

- Each derived attribute (i.e. the frames from the `derived_attributes` slot).
- Each inverse attribute (i.e. the frames from the `inverse_attributes` slot).
- Each uniqueness rule (i.e. the frames from the `unique_rules` slot).
- The expression from each domain rule (i.e. the `expression` slot filler from each frame in the `domain_rules` slot). The domain rule frames themselves are not used as retrieval prompts, as a domain rule merely consists of an expression and a label (which is of no use to the constraint-understanding system[4]).

The constraint-understanding system iterates only over the entities in the input model. It does not need to iterate over defined types and functions, because they can be accessed from attributes and expressions (e.g. in domain rules) respectively.[5]

```
ENTITY name [[ABSTRACT] SUPERTYPE [OF (supertype expression)]]
            [SUBTYPE OF (supertypes)];
     [explicit attributes]
[DERIVE
     derived attributes]
[INVERSE
     inverse attributes]
[UNIQUE
     uniqueness rules]
[WHERE
     domain rules]
END_ENTITY;
```

Figure D-1. An entity template for visualising sources of constraints.

To help visualise this process, Figure D-1 shows a template for an entity definition, with the syntactic sources of constraints highlighted in bold italics (square brackets denote optional constructs). Each of these sources, as well as the entity frame itself, is presented to the case

4. The constraint-understanding system currently does not attempt to exploit the meanings of rule labels, identifiers and comments, as natural language analysis is beyond the scope of this project (there is potential for future work here - see Appendix G, Section G.3.1).
5. It *would* need to iterate over global rules, since they are not referenced from within entities, but global rules are not currently supported.

library as a retrieval prompt.

As well as submitting frames from the model itself to the case library as retrieval prompts, the constraint-understanding system also has to use any HLCs that it has already extracted as retrieval prompts, in order to extract further HLCs from those. The use of HLCs as retrieval prompts is discussed in the next subsection.

D.1.2 Navigating HLCs Already Extracted

The constraint-understanding system needs to submit the HLCs that it has already extracted as further retrieval prompts to the case library, because there might be more HLCs that can be extracted from these HLCs. For example, we saw in Section 6.3 that a 'range' HLC is extracted from two bounds and a conjunction. Another example is extracting adders or multipliers from equations, as discussed in Appendix B, Section B.3.

The issue this raises is how to control the extraction of HLCs from the HLCs already extracted. It is a matter of when to attempt to extract further HLCs, and in what order to do so. There are two obvious ways to control extraction of HLCs from HLCs:

- *Depth first*: Whenever an HLC is extracted, immediately try to extract further HLCs (using it as a retrieval prompt). The difficulty with this approach is that if more than one HLC is needed to extract another HLC, it will not be possible (at least not immediately) to do so. Consider the example of extracting a range from two bounds. Using the depth-first approach, the system would try to extract the range after extracting the first bound, and would fail because it had not yet extracted the second bound (though it could try to extract it top-down).
- *Breadth first*: Start by extracting all the HLCs that can be extracted from the input model alone (as described in Section D.1.1), adding each HLC to a queue as it is extracted. Then submit each HLC in the queue (in first-in-first-out order) as a retrieval prompt to try to extract further HLCs. Add any HLCs thus extracted to the queue, and continue

until all the HLCs in the queue have been processed. This is the approach that has been adopted for the current implementation of the constraint-understanding system, because it is maximally effective at extracting all the HLCs that can be extracted from a model, and does not suffer from the difficulty that depth-first extraction does. Breadth-first extraction is not as psychologically plausible as depth-first extraction because it lacks attention focus (humans would tend to finish thinking about one constraint before moving on to another, and breadth-first extraction is not as apt to make opportunistic inferential jumps as depth-first extraction could be). However, since extraction is complete in the current constraint-understanding system, breadth-first extraction is just as effective as depth-first extraction.

Although the current system is required to extract all HLCs that it can extract from a given model, this luxurious requirement could be relaxed for future versions in which the role of constraint understanding is to support a specific task (such as transferring constraints for schema-to-schema mapping - see Section 1.3.2). When HLCs are not simply being extracted for the sake of it, it will be unnecessary (and expensive) to extract all HLCs from a model - ideally, only those that are needed to support the task should be extracted (see Appendix G, Section G.3.3). In the context of such a task, extraction could be guided by heuristic knowledge rather than extracting all HLCs indiscriminately. This would give rise to more intelligent extraction strategies than the pure depth-first and breadth-first approaches described above. It is analogous to the way artificial intelligence systems use heuristic knowledge to guide search, in algorithms such as best-first search (Section 2.1.4 of Pratt [54]) or hill climbing (Chapter 4 of Winston [86]), as opposed to the blind methods: depth-first and breadth-first search.

The last two subsections have described how the constraint-understanding system navigates the input model and the HLCs it already extracted from it, submitting retrieval prompts to the case library. The next subsection describes how these retrieval prompts are used to retrieve cases, match them and extract the HLCs suggested by the cases onto the model following successful matches.

D.1.3 Case Retrieval and Top-Level Control of Matching and Extraction

As mentioned in the two previous subsections, the constraint-understanding system navigates the input EXPRESS model and the HLCs already extracted from it, presenting syntax frames to the case library that are potential sources of HLCs. It then proceeds to present the HLCs already extracted to the case library as further prompt frames. This subsection describes the retrieval of cases from the case library, followed by the top-level control of the matching of these prompt frames to the retrieved cases and the extraction of HLCs as suggested by those cases that match.

The nature of extraction is that all cases which match the input model should be used to extract HLCs.[6] This contrasts with the usual case-based reasoning philosophy of selecting the best case to solve a problem. It means that retrieval is required to return all potentially relevant cases. Effectively, the input model constitutes a series of small, local problems rather than one large one, because HLCs are extracted from relatively small, local fragments of the model. Therefore, many of the cases in the library are independent rather than competing to solve the same problem. Here, it is not a matter of *how well* cases match the augmented model, but *whether* they match it or not.

Linear case library organisation and retrieval (see Section 3.2.2.1) would be suitable in that linear retrieval is guaranteed to find all matching cases. However, linear retrieval is particularly inefficient as it calls for all cases in the library to be matched rather than just those cases that are likely to be appropriate. The larger the case library, the longer each retrieval takes. Furthermore, the larger the input model, the longer the program takes to run because there are many more sources of HLCs for large models than for small models, and

6. Although, as discussed in the previous section, the luxury of extracting all HLCs that can be extracted from a model could be compromised for tasks that *use* an understanding of constraints, as opposed to constraint understanding in its own right.

a retrieval is needed for each.[7] A more efficient implementation would prune out, before the matching stage, cases that could not possibly be relevant to the given retrieval prompt, provided that no potentially relevant cases are missed.

The retrieval algorithms referred to in Section 3.2.2.2 gain efficiency at the cost of being prone to missing some relevant cases, which makes them inappropriate here unless they only discriminate on features that only prune out cases from the search that are definitely irrelevant.[8] For example, indexing each case by the frame class of the frame in its `problem` slot (for bottom-up extraction)[9] will not miss any potentially relevant cases because the matching algorithm (described in Section D.2) only considers cases whose frames in the `problem` slot are of the same frame class as the prompt frame.

So, an organisation and retrieval scheme based on *shallow indexing* (see Section 3.2.2.3) has been adopted. The case library is organised as a hash table,[10] mapping frame classes to the cases which have frames of those frame classes in their `problem` slots:

- The *keys* for this hash table are frame classes;
- Each *value* in this hash table is a list of cases, all of whose `problem` slots are filled with a frame of the same frame class. Furthermore, all cases whose `problem` slots are filled with a frame of a particular frame class are in the same list.

To illustrate the above, Figure D-2 shows how the case library would be organised if it contained just the three cases for extracting ‘exactly one’ (Appendix C, Section C.4):

---

7. This may sound as though it could lead to gigantic combinatorial explosion. However, because understanding each constraint is a relatively small, localised problem, combinatorial explosion has not proven to be a major problem in practice.
8. This restriction could be loosened for a constraint-understanding system that is not required to extract all the HLCs it could from a given model. This could make the standard case retrieval algorithms appropriate after all.
9. Or by the frame class of the frame in its `solution` slot (for top-down extraction).
10. A *hash table* (Chapter 5 of Weiss [85]) is an efficient implementation of a set of associated pairs of data - the *value* for a given *key* can be looked up in constant average time.

- The cases for extracting 'exactly one' from enumeration and select types (Appendix C, Sections C.4.1 and C.4.2) both have an `explicit_attribute` frame in their `problem` slots. These are Case #1 and Case #2 in Figure D-2.
- The case for extracting 'exactly one' from a `ONEOF` supertype expression (Appendix C, Section C.4.3) has an `entity_type` frame in its problem slot. This is Case #3 in Figure D-2.

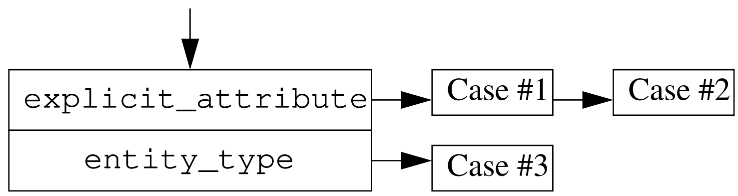

Figure D-2. Shallow indexing (hash table) case library organisation.

The case retrieval algorithm that the case library uses hence looks up the frame class of the prompt frame in the hash table, which yields a list of cases. Each of these cases is said to have been *retrieved*, and is passed on to the matcher. So the matcher has a set of cases to match, but it is a small subset of all the cases in the case library, whereas in pure linear retrieval it would be *all* the cases in the case library.

The case library's memory update algorithm (for adding cases to the library) takes as input a case (already constructed, with its `problem` and `solution` slots filled) and inserts it in the appropriate list, which it looks up in the hash table by the frame class of the new case's `problem` slot (as for retrieval). The memory update algorithm is only invoked when seed cases are manually added to the case library, as mentioned in Section 4.2 - the constraint-understanding system does not retain any cases of its own yet (but it could in the future - see Appendix G, Section G.2).

This section has described the top-level algorithm for extracting HLCs from an input EXPRESS model. Section D.1.1 described how this algorithm navigates the input model, submitting potential sources of HLCs to the case library as retrieval prompts. It also submits the HLCs already extracted as retrieval prompts to extract further HLCs, as discussed in Section D.1.2.

When a retrieval prompt is submitted to the case library, the case library retrieves a set of cases which have the potential to suggest HLCs to extract from the input model. Section D.1.3 described how the case library is organised, and its algorithms for case retrieval and memory update (although memory update is never called during the extraction process, its description is included here for completeness).

The case matcher is then called for each case returned by the case library's retrieval algorithm, to match that case's 'problem' part to the current retrieval prompt. This will be described in detail in the next section. If the match succeeds, then the HLC that the case suggests is extracted from the model (which involves making a copy of the HLC in the 'solution' part of the case, replacing the references to frames in the 'problem' part of the case with references to the corresponding frames in the augmented model - this will be described in Section D.3).

D.2 The Case Matching Algorithm

Each case retrieved is matched to the current prompt frame from the input EXPRESS model (or it could be an HLC already extracted from the input model, from which further HLCs could be extracted) to determine whether the prompt matches the 'problem' part of the case sufficiently well to extract the HLC suggested by the 'solution' part of the case.

The matching algorithm described here is quite harsh, in that the whole match fails if any part of it fails. As explained earlier, the nature of the task of extracting HLCs is to apply all matching cases, rather than to select a best-matching case (as in more orthodox CBR problems).

An extension to the case matching algorithm would be to allow partial matching (c.f. Section 3.2.3) and adaptation (c.f. Section 3.3) other than the simple reinstantiation that the current system does. The matcher would record the differences between the ‘problem’ part of the case and the retrieval prompt. Then adaptation (performed when the HLC from the ‘solution’ part of a successfully matching case is extracted) would have to compensate for each difference. For example, the relational operator for a bound may be ‘>=’ in the case but ‘>’ in the prompt. In this example, it would have to adapt an inclusive bound (the case’s solution) to be exclusive.

The use of partial matching and adaptation would save space in the case library (through not having to store all the variants of a case explicitly). However, it would make matching more expensive in terms of time - the current system can give up on a match as soon as it encounters a difference, but a partial matching algorithm would have to continue because it might be able to adapt for that difference. Partial matching and adaptation would also introduce the problem of having to choose between cases which are competing to solve a problem, because several partially-matching cases could then be suggesting the same HLC to extract. Partial matching and adaptation are further discussed in Section G.2.2 of Appendix G as an area for future development.

For the match to succeed, every slot in the case frame must match the corresponding slot (the one with the same name) in the prompt frame. Note that the converse is not true - not every slot in the prompt frame has to match the corresponding slot in the case frame. This is because prompt frames come from actual models, and are typically cluttered with slots which are irrelevant to the pattern that the case is trying to recognise. A case encodes a minimal fragment of EXPRESS (or lower-level HLCs) from which it is appropriate to extract the HLC suggested by the case. So only those slots in the prompt which are relevant to the case are considered by the matching process. As mentioned in Section 6.3, a case encodes the fact that a particular slot is irrelevant by omitting it (though it can specify that a particular slot must be null by “`slot = null;`”).

Comparing the fillers of two slots to see if they match can be a case of either:

- comparing two constants (other than frames) for equality;
- comparing a variable in the case to a constant in the prompt; or
- comparing two frames: the matcher is called recursively.

As a side effect, the case matching process keeps track of the associations between frames in the case and the corresponding frames in the augmented model, so that when the HLC is extracted later[11] (as will be discussed in Section D.3), the system will be able to substitute frames from the actual input model for the frames they matched in the case. In a similar way, the case matching process also records the bindings of variables in the case as it unifies them with constants in the input model, so that the variables can be replaced with their bindings when the HLC in the 'solution' part is instantiated and extracted.

The obvious way to represent these frame associations and variable bindings would be to use a hash table which maps frames in the case to the frames they correspond to in the augmented model, and another hash table to map variables to their bindings in the augmented model. Hash tables would suffice for storing and looking up frame associations and variable bindings, but this scheme has a limitation. The limitation has to do with *retracting* frame associations and variable bindings. As we shall see later on, it is possible for the matcher to take a 'wrong turn', during which it may record some frame associations and variable bindings. When the matcher realises that it has made a wrong choice, it needs to backtrack to the point at which it made that choice and try an alternative. Any frame associations and variable bindings that were recorded since the matcher made the wrong choice are invalid, and should be retracted when the matcher backtracks. The problem with using simple hash tables is that there is no way for the system to know *which* frame associations and variable bindings stored there are invalid and hence should be retracted.

11. Providing the match succeeds, of course.

As a solution to this problem, a data structure called a *hash table stack* has been implemented. The matcher uses two instances, one for frame associations and one for variable bindings. A *hash table stack* is a stack of hash tables. It has the following methods:

- `push`: This creates a new hash table and pushes it onto the top of the stack. It is called when the matcher reaches a choice point and makes a choice that could later turn out to be wrong (we shall see later what constitutes a choice point).
- `store`: This takes as input a key-value pair, and stores it in the hash table at the top of the stack (the key is either a variable or a frame in the case, and the value is either the binding of that variable or the corresponding frame in the augmented model).
- `lookup`: This takes as input a key, and looks it up in each hash table on the stack in turn until it is found, whereupon the corresponding value is returned (or null if the key is not in any of the hash tables, indicating that the variable is not bound or the frame not associated).
- `pop`: This removes the hash table at the top of the stack. If the matcher has to backtrack, it calls `pop` in order to retract the variable bindings and frame associations that were erroneously recorded while it was going the wrong way.

The backtracking problem discussed above occurs quite often in artificial intelligence, usually in the guise of having to retract beliefs. When an intelligent agent ceases to believe something, it should retract those beliefs, and only those, that derive from the belief that is explicitly being retracted. A simple form of retraction is *chronological backtracking*, where the system retracts all the conclusions it drew since the last choice point when it backtracks. This can be done using a stack-oriented context mechanism (Section 5.1.3 of Forbus and de Kleer [28]), which is equivalent to the scheme above which uses hash table stacks. A more complicated form of retraction is *dependency-directed backtracking* (Section 6.1.4 of Forbus and de Kleer [28]), which records a justification for each belief in terms of the beliefs it was derived from. Only those beliefs whose reason to be believed has been undermined need to be retracted, even if the system came to believe them after a choice point to which it has to backtrack. Chronological backtracking can waste time for systems which need to revise their beliefs, but it is all that is needed for keeping track of frame associations and variable bindings because that does not involve non-chronological dependencies.

The case matching algorithm discussed above will now be presented in object-oriented pseudocode, as a set of instance methods of the class `Case`. The first, `match`, is a public method, and the top-level function for the case matching process. It takes a prompt frame as an input parameter, and returns true if the prompt frame successfully matches the 'problem' part of the current instance of `Case` (i.e. the case that the system is up to in its match-extract cycle for the current prompt frame), and false if it fails to match. The others are private methods which serve as auxiliary functions (they are not visible outside the `Case` class).

The class `Case` has three instance variables:

- `_frameMap`: a hash table stack which associates frames in the case with the frames they correspond to in the prompt;
- `_variableMap`: a hash table stack which associates variables with their bindings;
- `_caseFrame`: an instance of the frame class `case_frame`, representing the top-level frame of this case.

D.2.1 `match`

The method `match` (Figure D-3) is the top-level function for the case matching process. It calls `matchFrames` (see Section D.2.2) to match the frame which fills the `problem` slot of `_caseFrame` to the prompt frame.

```
Case::match (prompt : Frame) : boolean {
     _frameMap:= new HashtableStack;
     _variableMap:= new HashtableStack;

     if matchFrames("problem" slot filler of _caseFrame, prompt) = false
     then {
          _frameMap:= null;
          _variableMap:= null;
          return false;
     }
     else return true;
}
```

Figure D-3. `match`: the top-level case matching algorithm.

The instance variables `_frameMap` and `_variableMap` are initialised to empty hash table stacks, ready for `matchFrames` to record frame associations and variable bindings in. If the match fails, `_frameMap` and `_variableMap` are cleared because they are no longer needed, but if the match succeeds then they are not cleared, because they will be needed for the extraction phase (which is invoked just when `match` returns true), as will be explained in Section D.3.

D.2.2 `matchFrames`

The method `matchFrames` (Figure D-4) is the heart of the case matching process: it matches a frame from the case (`frameFromCase`) to a frame from the prompt (`frameFromPrompt`), which are top-level frames if `matchFrames` was called directly from `match` (see Section D.2.1), and deeper frames if called from `matchFillers` (see Section D.2.3).

```
Case::matchFrames (frameFromCase : Frame, frameFromPrompt : Frame)
                  : boolean
{
     look up frameFromCase in the _frameMap hash table stack;
     if it is there already
     then {
          if it is to frameFromPrompt that frameFromCase is mapped
          then exit, return true;
          else exit, return false;
     }
     store (frameFromCase -> frameFromPrompt) in _frameMap;

     if frameFromCase's frame class does not subsume
        frameFromPrompt's frame class
     then exit, return false;

     for each slot which is filled in frameFromCase
         (except for inverse slots that are filled indirectly)
     {
          if matchFillers(slot's filler in frameFromCase,
                          slot's filler in frameFromPrompt)
               = false
          then exit, return false;
     }

     return true;
}
```

Figure D-4. `matchFrames`:
algorithm to match a frame from the case to a frame from the retrieval prompt.

The method `matchFrames` first checks to see if `frameFromCase` has been matched already in the course of the current invocation of `match`. If it has already been matched to `frameFromPrompt` then `matchFrames` immediately returns true (which avoids recomputation and eliminates the possibility of infinite recursion). If it has already been matched to a frame other than `frameFromPrompt` then `matchFrames` immediately returns false (thereby failing the whole match), for a frame in the case cannot map to multiple frames in the prompt. So `matchFrames` starts by looking up `frameFromCase` in the `_frameMap` hash table stack and, only if it is not there already, stores it there (associating it with `frameFromPrompt`) and then proceeds with the matching.

For `frameFromPrompt` to match `frameFromCase` successfully, the frame class of `frameFromCase` must subsume that of `frameFromPrompt`, i.e. the frame class of `frameFromPrompt` must be the same as, or a subclass of, the frame class of `frameFromCase`.

Furthermore, every slot which is filled[12] in `frameFromCase` must match the corresponding slot in `frameFromPrompt`. The only exceptions to this are inverse slots (see Section 5.3.5) which are filled indirectly, i.e. by virtue of other frames setting their `FOR` attributes rather than being filled explicitly.[13] It is the job of `matchFillers` (see Section D.2.3) to match corresponding slot fillers from the case and prompt frames.

If `frameFromCase` and `frameFromPrompt` come through the above battery of tests, then `matchFrames` returns true (so matching will continue in the caller).

The `matchFrames` algorithm is where the frame system discussed in Chapter 5 really comes into its own - frame classes and slots are first class objects, which can be represented explicitly, independently of the matching algorithm. If standard records and fields were used, they would have to be hardwired into the program, and a specific matching function would have to be written for each record type. With the frame system as the foundation, it has been possible to write a generic matching algorithm, which makes the system much more flexible and extensible - new frame classes (e.g. new kinds of HLC) can be added simply by editing the frame class EXPRESS model (see Appendix A), with no need to edit the program.

---

12. Including slots with a value of null explicitly specified in the case, but not slots which are unfilled by omission.
13. Inverse slots can, however, have fillers explicitly specified in the case, which are considered to fill the inverse slots directly, in which case they *are* required to be matched in the prompt frame.

D.2.3 `matchFillers`

```
Case::matchFillers (valueFromCase : Object, valueFromPrompt : Object)
                   : boolean
{
     if valueFromCase is explicitly specified as null
     then return true if valueFromPrompt is null,
                 false otherwise;

     if valueFromPrompt is null
     then return false;

     if valueFromCase is a variable
     then {
          look it up in the _variableMap hash table stack;
          if it is not there already
          then {
               store (valueFromCase -> valueFromPrompt) in _variableMap;
               return true;
          }
          else return matchFillers(valueFromCaseBinding,valueFromPrompt)
                 where valueFromCaseBinding is whatever valueFromCase
                 is bound to, as just looked up in _variableMap;
     }

     if valueFromPrompt is not type-compatible with valueFromCase
     then return false;

     if valueFromCase and (hence also) valueFromPrompt are frames
     then return matchFrames(valueFromCase,valueFromPrompt);

     if valueFromCase and valueFromPrompt are instances of
          a defined type (the same type for both - return false if not)
     then return matchFillers(underlying value of valueFromCase,
                              underlying value of valueFromPrompt);

     if valueFromCase is an aggregation
     then use an aggregation matching algorithm and return the result;

     return equals(valueFromCase,valueFromPrompt)
          where equals is the equality relation for whatever the type of
          valueFromCase and valueFromPrompt is;
}
```

Figure D-5. `matchFillers`:
algorithm to match a value from the case to a value from the retrieval prompt.

The method `matchFillers` (Figure D-5) is used to compare a value from the case with a value from the prompt, which are literally slot fillers when it is from `matchFrames` (see Section D.2.2) that `matchFillers` is called. The two values can be of any type (as denoted by `Object`).

The algorithm first checks whether the value from the case is explicitly specified to be null, in which case the value in the prompt frame is required to be null for the match to succeed. In all other circumstances, the value in the prompt must not be null.[14] This behaviour corresponds to the discussion in Section 6.3 of how filled and unfilled slots in cases are treated by the case matcher.

If the case value is a variable then it is unified (see Section 6.3) with the prompt value:

- If the variable has not been bound yet, then the match succeeds unconditionally and the binding is recorded;
- If the variable has already been bound, then the value to which it is bound is matched to the prompt value.[15]

The value from the prompt has to be type-compatible with the value from the case (e.g. they must be of the same type, or the prompt value's type could be a subtype of the case value's type) in order for them to match. This condition is checked first, followed by a case analysis of the type of the two values:

- Frames are matched recursively (by calling `matchFrames`);
- Instances of defined types[16] are matched by comparing their defined types and matching their underlying values;
- Aggregations are matched using aggregation matching algorithms (discussed below);

---

14. If it were acceptable for the slot filler in the prompt to be null, the corresponding slot filler in the case would be omitted and `matchFillers` would not have been called at all.
15. Note that the prompt value cannot be a variable itself, because variables are not allowed in the augmented model (they are only allowed in cases, for the purpose of matching).
16. This refers to defined types in the frame system (see Section 5.3.3).

- If the types of the values are not any of the above, then they are compared using a standard equality relation for their type.

There is more than one way to match an aggregation in the case to an aggregation in the prompt. There are two sensible policies for matching aggregations (it is unclear which is intrinsically correct - it depends on the circumstances):

- **Policy 1:** Every element in the case aggregation must match a corresponding element in the prompt aggregation, but not every element in the prompt aggregation needs to be thus matched. Effectively, each individual element in the case aggregation is treated like a filled slot, which must be matched in the prompt.
- **Policy 2:** Every element in the case aggregation must match a corresponding element in the prompt aggregation, and every element in the prompt aggregation must be matched. Effectively, the aggregation is treated like one slot filler, which must be matched exactly.

The current implementation of the case matcher uses Policy 1 when the case aggregation is *unordered* (i.e. is a set or a bag). The match succeeds just in case *every* element in the case aggregation matches *some* element in the prompt aggregation. This is where the backtracking discussed early in Section D.2 comes in. For each element in the case aggregation, a number of elements in the prompt aggregation may have to be matched before one of them matches. The selection of a prompt element constitutes a choice point, and if that element fails to match the current case element, the matcher has to backtrack to that choice point (retracting any variable bindings and frame associations that were recorded in the meantime) and try another element in the prompt aggregation.[17]

The current case matcher uses Policy 2 for *ordered* aggregations (i.e. lists and arrays), however. Every element in the case aggregation must match an element in the prompt aggregation, *and* vice versa, and the corresponding elements must be in the same order in

17. This is the only situation in the current case matching algorithm where local failure to match does not result in the whole match failing (for the current case), so it is the only situation where backtracking is needed.

the two aggregations. A quick test for failure is whether the two aggregations are of unequal length. (There is no backtracking.)

Policy 1 was chosen for unordered aggregations and yet Policy 2 was chosen for ordered aggregations, because this decision feels intuitively correct for the majority of circumstances. For example, when extracting a range from a conjunction of two bounds (see Figure 6-7), the aggregation to be matched in the case is the set `{bound#21,bound#22}`. Failing to match a superset of this in the prompt would be ridiculous - it does not matter if the conjunction contains more HLCs besides the two bounds. On the other hand, when an ordered aggregation is to be matched, such as a list of parameters for a function call, it *does* seem sensible to fail the match if the list in the prompt has extra elements. However, the argument for using Policy 1 for unordered aggregations and Policy 2 for ordered aggregations may not generalise to all circumstances, so it may be necessary to override these defaults in individual cases - FIL could be extended with a means of specifying whether a particular aggregation slot filler in the 'problem' part of a case should be matched according to Policy 1 or Policy 2.

This section has described the case matching algorithm used by the constraint-understanding system to determine whether a retrieval prompt successfully matches the 'problem' part of a case. If it does, the extraction algorithm described in the next section is called to extract from the model the HLC suggested in the 'solution' part of the case.

D.3 The Post-Match Extraction Algorithm

As mentioned earlier, if a prompt successfully matches the 'problem' part of a case, then the HLC in the 'solution' part is extracted. This means making a deep copy of the HLC from the 'solution' part, copying all its slots.

Any variables in the ‘solution’ part are replaced with the values to which they were bound in the course of matching the ‘problem’ part of the case to the input model (the ‘solution’ part of a case should not contain variables which will not be bound during matching).

Any frames in the ‘solution’ part which have been matched to frames in the prompt during matching are replaced with the frames that they are associated with in the prompt. If a frame in the ‘solution’ part is not associated with a frame in the prompt, then it is copied recursively - no references from within the case end up in the copied solution.

The process described above will now be presented in pseudocode. Again, there are three instance methods of the `Case` class, the first of which, `extract`, is a public method.

D.3.1 `extract`

The method `extract` is the top-level function for extracting the HLC suggested by the case. It simply calls `extractFrame` (Section D.3.2) with the top-level frame of the case's `solution` slot.

Note that the precondition of calling `extract` is that `match` has been called and returned true. This not only ensures that it is legitimate to extract the HLC suggested by the case, it also records variable bindings and frame associations in `_variableMap` and `_frameMap` respectively, which are needed to instantiate the solution frame properly.

```
Case::extract () : Frame {
     result:= extractFrame("solution" slot filler of _caseFrame);

     _frameMap:= null;
     _variableMap:= null;

     return result;
}
```

Figure D-6. `extract`: the top-level algorithm for post-match extraction.

At the end, `extract` clears `_frameMap` and `_variableMap`, because that is when the system has finished with them.

D.3.2 extractFrame

The method `extractFrame` (Figure D-7) is an auxiliary function to create a new frame with which to augment the model, by copying the slot fillers from a given frame from the case's 'solution' part.

```
Case::extractFrame (oldSolutionFrame : Frame) : Frame {
     newSolutionFrame= new Frame of the same frame class as
          oldSolutionFrame;

     store (oldSolutionFrame -> newSolutionFrame) in _frameMap;

     for each slot in oldSolutionFrame {
          set slot's filler in newSolutionFrame to the result of
          extractFiller(slot's filler in oldSolutionFrame);
     }

     return newSolutionFrame;
}
```

Figure D-7. `extractFrame`:
algorithm to reinstantiate a frame from the case's solution for the actual model.

The algorithm records in `_frameMap` an association between the frame from the case's solution and the new copy of that frame which is being instantiated for the augmented model. This is to ensure that if the solution consists of more than one frame,[18] and the frames in the solution refer to each other, then only one copy of each frame will be created.

The `extractFiller` method, described in the next subsection, is responsible for processing the slot fillers as they are copied from the old solution frame to the new solution frame, substituting bindings for variables, and substituting frame references in the

18. Not that any of the cases currently implemented have solutions which consist of more than one frame - their solutions are just single HLC frames.

augmented model for frame references in the case.

D.3.3 `extractFiller`

The method `extractFiller` (Figure D-8) processes the slot fillers from the case, replacing frames associated with frames in the augmented model, and substituting bindings for variables. The input value is literally a slot filler only if it is from `extractFrame` that `extractFiller` has been called, as `extractFiller` can also recurse to process values which are not top-level slot fillers.

```
Case::extractFiller (filler : Object) : Object {
     if filler is a frame
     then if it is associated with a frame in the prompt
     (according to _frameMap)
          then return the frame with which it is associated;
          else return extractFrame(filler);

     if filler is a variable
     then if it is bound to a value in the prompt
     (according to _variableMap)
          then return the value to which it is bound;
          else error - unbound variable;

     if filler is an aggregation
     then {
          create a new aggregation;
          for each element in filler {
               insert extractFiller(element) into the new aggregation;
          }
          return the new aggregation;
     }

     return filler;
}
```

Figure D-8. `extractFiller`:
algorithm to process a value from the case's solution for the actual model.

The algorithm does a case analysis of the type of the value to be processed:

- A frame is replaced with the frame that it was associated with during matching, or copied deeply using `extractFrame` if it is not already associated with a frame from the prompt.

- A variable is replaced with the value to which it was bound during the match.
- An aggregation has its elements processed using `extractFiller` recursively, as those elements may also contain frames or variables.
- Any other type of value is simply transferred as it is.

This appendix has described the algorithmic details of the constraint-understanding system - in particular, the mechanisms for matching a retrieval prompt from the model to the 'problem' part of a case, and for extracting the HLC suggested by the 'solution' part of the case onto the model. Appendix E discusses some of the issues of implementing the constraint-understanding system as a computer program. The actual source code is available for browsing on the Author's web site (Broad [8]).