Chapter 3:
Case-Based Reasoning

Since this thesis takes a case-based reasoning approach to understanding the constraints on information models, this chapter provides an overview of the field of case-based reasoning in order to justify its use in this application.

Case-based reasoning (CBR) is the enterprise of solving new problems by analogy with old ones, rather than solving every problem from first principles as traditional problem solvers do. Kolodner [39] defines CBR as remembering previous similar situations to help solve a new problem. Riesbeck and Schank [65] define CBR as solving new problems by adapting solutions that were used to solve old problems. Figure 3-1 captures the essence of CBR diagrammatically.

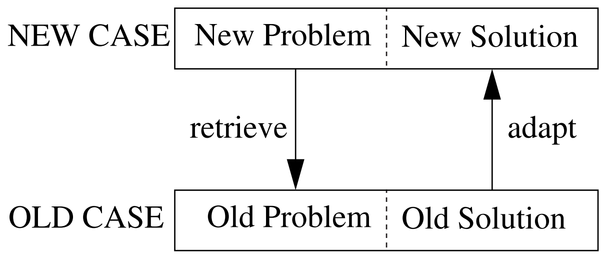

Figure 3-1. Case-based reasoning.

Previous experiences that are useful for future reasoning are stored in memory as *cases*, which can be recalled as appropriate when the reasoner is faced with a new problem. Section 3.1 explains in more detail what cases are (both in general and in some specific problem domains), and how they can be represented.

Section 3.2 addresses the problem of *case retrieval*, which concerns recalling from memory an appropriate case (or cases) which is (or are) useful for solving a new problem. The best case (i.e. the one which is most similar to the new situation[1] in the right respects) often

1. This thesis uses various synonymous terms for the problem situation that a case-based reasoner is faced with: {*new/current*} *problem*, {*new/current*} *situation* and *retrieval prompt* can be read interchangeably.

matches the new situation only partially, which means that there will be differences between the old problem and the new problem.

Section 3.3 explains how the retrieved case can be used to solve the new problem. The solution to the old problem recorded in the case suggests how to solve the new problem, but the old solution often has to be *adapted* to compensate for the differences between the old and new problems. Section 3.3 will discuss adaptation, and also the matter of evaluating solutions and repairing them if they are faulty.

Once a case-based reasoner has solved a problem, it may store that problem-solving episode as a case in memory so that it will be available to help solve similar problems in the future. Storing a new case in memory is sometimes referred to as *memory update* (Section 1.4.6 of Kolodner [39]). This issue will be discussed in Section 3.2, as it is strongly related to retrieval.

Section 3.4 will discuss the advantages and disadvantages of CBR, and will conclude with a justification of the suitability of CBR for code generation and understanding in general, and for the task of understanding constraints in particular.

3.1 Case Representation

This section considers what cases are (Section 3.1.1), and how they can be represented (Section 3.1.2).

3.1.1 What is a Case?

Section 1.2 of Kolodner [39] defines a *case* as "a contextualized [sic] piece of knowledge representing an experience that teaches a lesson fundamental to achieving the goals of the reasoner".

Essentially, a case is the association of a situation with some additional knowledge that can be inferred from that situation; for example, a case might associate a problem with its solution, or a situation with its outcome. Thus, like a rule (Chapter 7 of Winston [86]), a case may be thought of as having an *antecedent* part and a *consequent* part, such that if the former is recognised in a new situation, the latter can be inferred or applied.

Problem:	Create a recipe for chicken and peas. Use stir-fry as the cooking method. Achieve a sweet ‘n’ sour taste.
Solution:	1. Bone the chicken. 2. Shell the peas. 3. Stir-fry the chicken and the peas together using a wok.

Figure 3-2. Example of a case (adapted from Price [56]).

A key issue in any application of CBR is what a case is in the domain of that particular application. Here are some examples:

- CHEF (Section 2.1 of Kolodner [39]) is a case-based planner in the domain of recipe creation. A case is a recipe. Figure 3-2 shows a simplified version of a typical case from CHEF, adapted from one of CASPIAN’s examples - CASPIAN is a CBR shell which can be downloaded from the Internet (Price [56]).
- JULIA (Section 2.3 of Kolodner [39]) is a case-based designer in the domain of meal planning. A case is a meal (the ‘solution’) which was served under a particular set of circumstances (the ‘problem’).
- CASEY (Section 2.2 of Kolodner [39]) is a case-based diagnosis system. A case associates a patient and his or her symptoms (the ‘problem’) with a diagnosis of his or her disorders, together with a causal explanation (the ‘solution’).

Unlike rules, cases tend to be large chunks of knowledge tied to a specific context, which are likely to match new situations only partially. Whereas rules are small pieces of knowledge whose antecedents must match a new situation exactly and whose consequents can be applied directly, the antecedents of cases can be matched *partially* to the new situation and their consequents must be adapted to account for the differences between the case and the new situation (Section 3.4.1 of Kolodner [39]).

Cases tend to be *experiential* in nature: they come from situations that the reasoner has experienced in the past, for example when it solves a problem and stores that problem and its solution as a case in memory. Rules, on the other hand, tend to represent general facts about a domain that humans proclaim to be true (e.g. if *?x* is a bird and *?x* is not abnormal then *?x* can fly). However, not all cases have to be derived from the reasoner's own experience (for example, a CBR system is often seeded with an initial set of cases, coded by hand), and some cases are rule-like in nature (i.e. they are relatively small pieces of normative, general knowledge). Taking such a broad view of cases means that all knowledge can be represented in a unified framework.

A case must teach a lesson fundamental to the goals of the reasoner. That is to say, it must have the potential to help the reasoner solve a problem that it would not be able to solve (as efficiently, as competently, or at all) if it did not have that case. This implies that a case is only worth retaining if it contributes to the reasoner's capabilities over and above what they are without that case, and that a reasoner should have at its disposal a collection of cases which cover the range of problems that it is likely to encounter.

Cases can support a variety of reasoning tasks. The most obvious one is to suggest solutions to problems, but cases can also suggest how to adapt an old solution to fit a new situation (Section 12.3.1 of Kolodner [39]), and predict what might happen in a new situation based on what happened in similar situations in the past (Section 13.3 of Kolodner [39]). The latter means that cases can be useful for evaluating solutions and pointing out failings so that they can be fixed, and for anticipating what could go wrong even before the reasoner attempts to solve a problem, thereby enabling mistakes that were made in the past to be avoided.

The next subsection follows up this discussion of what cases are with the more pragmatic issue of how cases can be represented as data in a computer.

3.1.2 What's in a Case?

A practical realisation of CBR must address the content of cases and what formalism to use to represent them.

As mentioned in Section 3.1.1, a case is essentially an association between an antecedent and a consequent. In the case of cases that suggest solutions to problems, this means that a case must at least contain a description of the *problem* to be solved (in the context of the old situation in which it was set), and the *solution* to the problem (as it was solved in the old situation). This enables the reasoner to retrieve cases by matching their ‘problem’ parts to the new problem situation, and to use the ‘solution’ part of the selected case to propose a solution to the new problem.

A problem-solving case that consists of a problem and its solution allows the reasoner to propose a solution to a similar new problem, but without knowing how good the solution is. Such a system is doomed to repeat its mistakes time and again, but if the case is also tagged with the *outcome* of solving the problem as the case suggests, the reasoner can learn to avoid its mistakes and repeat only its successes, which enables it to become more competent as well as more efficient over time (Section 5.1 of Kolodner [39]).

A case that suggests an adaptation to a solution must record the difference between problem situations that needs to be adapted for, and the change that must be made to a solution to adapt for that difference. A case that warns of the potential for failure must record the features of the situation which predict that failure, and what the failure actually is (Section

6.2.3 of Kolodner [39]).

In essence, a case must record the information which will enable the reasoner to use that case to achieve the goal that the case is supposed to support. It is beyond the scope of this subsection to enumerate every piece of information that could be useful to record, but this needs to be analysed when a case representation is designed to support a particular reasoning task.

As pointed out in Section 5.2 of Kolodner [39], there is a variety of formalisms that can be used to represent cases, such as frames, semantic networks and forms. The substantive point is that cases must be represented in a format that works for a given application and holds the information needed for that application, rather than that particular knowledge representation formalisms must always be followed religiously.

This section has addressed the issues of what cases are, what they should consist of and how to represent that knowledge. The next section discusses the retrieval of cases that will help to solve a problem (prior to actually solving the problem, which will be considered in Section 3.3).

3.2 Case Retrieval

The task of *case retrieval*, given a situation in which reasoning goals are to be achieved, is to find a case (or cases) that can help to achieve those reasoning goals. This is separate from the problem of actually using the retrieved case(s) to achieve those goals, which will be addressed in Section 3.3.

The *indexing problem* (Section 1.1 of Kolodner [39]) is the conceptual problem of labelling cases such that they will be retrieved under appropriate circumstances, a problem which can manifest itself in different guises but which any case-based reasoner must, by definition, address. Section 3.2.1 will discuss indexing.

Section 3.2.2 discusses the issue of c*ase library organisation* (how to structure the *case library* - the repository in which cases are stored - to facilitate retrieval which is both effective and efficient) and the inextricably intertwined issue of what *retrieval algorithm* to use to navigate the case library in search of relevant cases.

Section 3.2.3 discusses the matter of matching cases to the current problem situation in order to determine which case(s) to select to use in solving the new problem.

3.2.1 Indexing

Indexes are features of a case that encode the circumstances under which it is appropriate to use the case, so that just those cases which are advantageous in the current problem situation will be retrieved (Chapter 6 of Kolodner [39]).

Indexes have three major roles (Section 10.2 of Kolodner [39]), and are implemented in different ways according to those roles:

- Indexes can act as conceptual labels on cases that describe the situations in which the case is useful.
- Indexes can be used to guide search through the case library (as in Section 3.2.2). Their purpose in this context is to assist the retrieval process in navigating towards the most relevant cases to the reasoner's goals. In this guise, an index is typically implemented as a link from one node to another, the link labelled with an attribute-value pair such that the retriever will traverse the link only when the current situation has that value for that attribute.
- Indexes can be used to designate what to prioritise in matching and ranking[2] (as in

2. *Ranking* (Chapter 9 of Kolodner [39]) is the process of ordering cases by their similarity score with respect to the new situation (so that the best cases can be selected).

Section 3.2.3). Indexes encode the relative importance of the case attributes,[3] so cases that match the current situation on important attributes will attain a higher similarity score than cases that match only on not-so-important attributes. For example, a similarity metric could be implemented as a weighted polynomial in which each term corresponds to an attribute: the multiplicand represents the degree of match for that attribute, and the multiplier represents the importance of that attribute - the multiplier is playing the role of an index.

The organisation of the case library affects the relative emphasis of these three roles. At the two extremes, a purely linear organisation places the indexing problem wholly on the matching process, whereas a purely hierarchical organisation emphasises the role of indexes to guide search. Whatever the balance, the issue of which features of a case determine under what circumstances it will be recalled cannot be avoided.

Good indexes have certain qualities:

- Indexes should be *predictive* (Section 6.1.1 of Kolodner [39]). When indexes are assigned to a case, the range of circumstances under which it would be appropriate to retrieve that case must be anticipated. Cases that suggest solutions to problems should be indexed by the key characteristics of the problem situation that make that solution appropriate (such as the goals in the problem situation that the solution achieves, as in CHEF (Section 2.1 of Kolodner [39])), so that the solution will be proposed in future situations that have those characteristics. Cases that suggest adaptations to solutions should be indexed by the differences in the problem that the adaptation accounts for. Cases that warn of the potential for failure should be indexed by the features that predict the failure. To generalise, cases should be indexed by the combinations of features of the antecedent part that make the consequent part appropriate.
- Indexes should be *distinctive* but not *unique* (Section 2.5.2 of Riesbeck and Schank

3. *Attributes* are also known as the *dimensions*, *slots* or *fields* of cases. An attribute is separate from the particular *values* or *fillers* of that attribute in individual cases.

[65]). That is, they must distinguish similar cases from each other in important ways, which is conducive to retrieving the most usefully similar[4] cases to a given situation. That indexes are distinctive helps to filter out the myriad of irrelevant cases to a situation; that indexes are not unique will cause cases to be recalled in the range of situations in which they are applicable.

- Indexes should be abstract enough to be generally applicable (Section 6.1.2 of Kolodner [39]), yet concrete enough to be recognisable (Section 6.1.3 of Kolodner [39]). If abstract descriptors are used as indexes to a case, it will be applicable to a wider range of situations, including those whose surface features differ from those of the case but which share the abstract descriptors. Unfortunately, it can be difficult to compute whether such situations do indeed have those descriptors! Such computation is called *elaboration* or *situation assessment* (Chapter 10 of Kolodner [39]), and has to be carried out before retrieval proper (as discussed in the next subsection) can commence.

An issue in CBR is whether to automate the assignment of indexes to cases (Section 7.2 of Kolodner [39]). This is clearly desirable in terms of achieving fully-automated CBR, and can be done using artificial intelligence techniques such as credit/blame assignment (Section 7.6 of Kolodner [39]), but it is a very difficult problem because today's computers are not good at anticipating the range of situations in which a case would be applicable, nor at generalising indexes so that they are applicable to a sensible range of situations without making them so general that they will be retrieved indiscriminately or so abstract that situation assessment becomes computationally infeasible.

This subsection has discussed indexing, which concerns the issue of what features of a case determine under what circumstances it will be appropriate to recall it. The next subsection discusses the actual retrieval of cases from a case library, together with how the case library is organised to support retrieval processes.

4. What is important is not just how similar to the current situation the retrieved cases are, but that they are similar in useful ways - Kolodner calls this *usefully similar* (page 283 of Kolodner [39]).

3.2.2 Case Library Organisation, and Retrieval Algorithms

This subsection discusses the various ways that a case library can be organised, and the corresponding retrieval algorithms. As usual in computer science, the choice of algorithm is inseparable from the choice of data structure. This subsection discusses the relative merits of the different schemes with respect to *effectiveness* (the ability to retrieve from memory the most usefully similar cases to the current problem situation) and *efficiency* (the amount of time and space consumed by the CBR system).

3.2.2.1 Linear Case Library Organisation

The simplest way to organise a case library is as a linear list of cases (Section 8.3 of Kolodner [39]). To retrieve cases, the list is searched sequentially, matching each case to the current problem situation and assigning a similarity score to the case (as discussed in Section 3.2.3). After searching through the entire list, the case (or cases, if more than one is needed) with the highest similarity score is (are) returned. Figure 3-3 illustrates this diagrammatically.

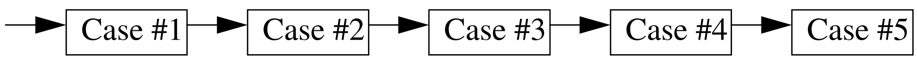

Figure 3-3. Linear case library organisation.

The main advantage of this scheme is that it is maximally effective: it is guaranteed to find the case with the highest similarity score, so retrieval is as good as the matching process.

Another advantage of linear case library organisation is that it is maximally efficient in space consumption, because it has the least overhead of storing organisational structure as well as the cases themselves. It is also maximally efficient in memory update time: a new case can be added to the library in O(1) time just by inserting it at one end of the list.

However, these advantages are offset by one potentially major disadvantage. Retrieval time is O(n) for a case library containing n cases, because every case in the list has to be matched in order to find the best case for the current problem. This time can be unacceptable for case libraries which contain many cases, and some case libraries can contain thousands of cases (Section 9.2.3 of Watson [84]). Furthermore, the retrieval time gets longer as more and more cases are added to the library. This seems neither intelligent nor psychologically plausible, as humans clearly do not consider every experience they have ever had whenever they try to recall a case, and they do not seem to become slower at retrieval as they acquire more knowledge.

Since one of the main advantages of CBR is that it is supposed to be more efficient to solve new problems by analogy with old ones than to solve them from scratch, it would be a shame if it took so long to retrieve cases that this benefit was outweighed, a phenomenon which is known as the *utility problem* (Smyth and Keane [69]).

One way to reduce the time needed to retrieve cases from a linear case library is to search it in parallel (Sections 8.3 and 8.8 of Kolodner [39]). An extreme realisation of parallelism would be to have a processor for every single case in the library, which matches it to the current situation at the same time as all the others. This is impractical for large case libraries containing thousands of cases! It used to be believed (according to Brown and Filer [10]) that massive parallelism was the only way to achieve efficient retrieval for CBR, but the more modern idea is that efficiency can be achieved by structuring the case library more intelligently so that only a sensible subset of potentially relevant cases has to be examined during retrieval. It is time to consider more sophisticated ways of organising case libraries.

3.2.2.2 Hierarchical Case Library Organisation

Some alternative ways to organise a case library are hierarchical in nature. This means that the cases are held in a tree structure in which nodes represent cases or generalisations of cases, and there are links from nodes to more specific nodes. Retrieval processes start at the top of the tree, and follow the links down to retrieve the cases, which are held at the leaf nodes. Figure 3-4 illustrates this diagrammatically (the rectangles represent specific cases, and the ovals represent generalisations of those cases).

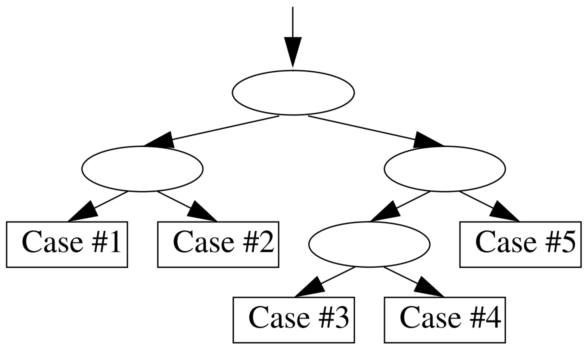

Figure 3-4. Hierarchical case library organisation.

There are different variants of this hierarchical scheme. For example, *shared feature networks* (Section 8.4 of Kolodner [39]) cluster together cases that share common features (i.e. common values for the same attributes) under the same internal node. Retrieval starts at the root node and moves down to the node whose features best match the current situation, continuing in this fashion until a leaf node (i.e. a case) is reached.

Discrimination networks (Section 8.5 of Kolodner [39]) partition cases by one attribute at a time. Effectively, each link is labelled with an attribute-value pair which discriminates the target node from its siblings, which have different values for the same attribute, as specified on their links. Retrieval starts at the root node and moves down the link whose value part best matches the current situation's value along that link's attribute part; retrieval continues in this fashion until a leaf node (i.e. a case) is reached.

The main advantage of hierarchical over linear case library organisation is the efficiency of retrieval. Whereas the time complexity of linear retrieval is O(n), retrieval from a hierarchy should be O(log n), roughly speaking - it depends on the branching factor of the tree (i.e.

how many outgoing links from each node there are) and how evenly balanced it is, but hierarchical schemes should not penalise retrieval time as much as linear schemes do as the case library grows.

Another advantage of hierarchical organisation is that it is conceptually more elegant and psychologically more plausible than a flat, linear organisation. It is intuitively obvious that humans structure their knowledge more richly than a linear list, providing associations between related concepts, economy of storage and speed of recall. The internal nodes of the hierarchy are also significant pieces of knowledge, because they represent generalisations of cases, which can be useful to reason with in a more abstract way than the cases themselves. For example, a person who has had many train journeys does not remember every individual journey, but rather the knowledge acquired from the experience of all those journeys is generalised - ossified, as it were - into a script, from which some individual episodes may be indexed. Only those (relevant parts of) cases that differ from the norm so as to teach an important lesson about exceptional circumstances that could not easily be inferred from the rest of the memory's knowledge are retained. The internal nodes in a hierarchical case library serve to capture automatically such generalisations, and advanced CBR systems could reason from generalisations rather than specific cases where appropriate (i.e. in normative situations) and deliberately forget (i.e. delete from memory) cases that do not contribute anything over and above their generalisations. (The ideas in this paragraph were culled from the discussions in Chapter 1 of Riesbeck and Schank [65], and Chapters 1 and 4 and Section 8.10.2 of Kolodner [39].)

The advantage of the time-efficiency of retrieval for hierarchical schemes is offset by some minor disadvantages:

- Extra space is required to store the organisational structure (all the links and internal nodes);
- Adding a new case to a hierarchical case library is less efficient than the O(1) time needed to insert a new case in a linear list. Inserting a case in a hierarchy takes roughly as

long as retrieval, because it has to be inserted at the right place, which is located by the same process as retrieval. Indeed, memory update could take *longer* than retrieval if it were necessary to reshuffle some nodes and links in the tree or create new generalisations (internal nodes) on account of the new case.

The major disadvantage of hierarchical case library organisations is that retrieval is not guaranteed to find the optimum case. In a linear case library, retrieval examines every case in the library, comparing it to the new situation, so it is guaranteed to find the best case (in as much as the matching process provides a good measure of how usefully similar a case is to the current situation).

However, retrieval on a hierarchical case library is apt to miss the best-matching cases, because even the best case in the library is likely to match the new situation only partially (the problem that the CBR system has to solve is not usually *identical* to a problem in any of its stored cases). If the retrieval process goes down the wrong link in a discrimination network because the attribute-value pair for the right link is not matched by the new situation, it could well miss cases that match on most of the other features that are discriminated further down the tree. The higher up the tree a wrong turn is made, the worse it will be in terms of the well-matching cases being retrieved, so discrimination networks are usually *prioritised* - the more important an attribute is, the higher up the tree it is discriminated on - but the fundamental problem still remains. Similar remarks apply to shared feature networks (mentioned earlier in this sub-subsection), which are slightly less prone to this problem (because they do not necessarily discriminate on one attribute at a time), but not much.

What are the consequences of failing to retrieve the best case in the library? At best, it will take longer to adapt the best case that *is* retrieved than it would have done for the best case in the library, which compromises the efficiency of hierarchical retrieval. At worst, the system may fail altogether to solve its problem when it could have succeeded using a more appropriate case, or the solution may be of lower quality than could have been achieved if the best case had been retrieved.

Another major disadvantage of the hierarchical schemes mentioned above is that there may not be enough information in the new situation to decide which link to follow, according to Section 8.6 of Kolodner [39]. She proposes either giving up, searching along all the links that cannot be decided between (which rather defeats the object of hierarchical organisation!), making a plausible guess (based on looking ahead or other knowledge), or using redundant discrimination networks.

A *redundant discrimination network* (Section 8.7 of Kolodner [39]) is a collection of discrimination networks, each prioritised differently, that are searched in parallel (either literally or metaphorically), giving up on one discrimination network if an impasse is reached whilst continuing in others.

Redundant discrimination networks suffer from the drawbacks that:

- They require more computing resources (space to store multiple discrimination networks, and either more time or parallel hardware to search them) than any other scheme discussed in this sub-subsection;
- They are more expensive to build, because every discrimination network has to be updated whenever a new case is acquired;
- It is inherently difficult to choose how to prioritise the networks in that there are $n!$ permutations (possible orderings) of the n attributes in the system's indexing vocabulary.

Furthermore, a redundant discrimination network leads to a *set* of cases being retrieved, from which the case(s) that will be used to solve the problem must be selected by matching each case in this set to the current situation, which amounts to linear retrieval on a subset of all the cases in the library. If there are many cases in the set retrieved from the redundant discrimination network (e.g. due to poor discrimination), then the matching process has a lot of work to do, which is precisely what hierarchical organisations were supposed to avoid!

The choice between linear and hierarchical case library organisation comes down to a trade-off between efficiency and accuracy. Linear organisation is more appropriate for small case libraries and when it is important to retrieve the best-matching cases. Hierarchical organisation allows faster retrieval but is more prone to missing good matches. It is also considerably more difficult to implement hierarchical case libraries than linear ones because they are so much more complicated.

Purely linear lists and pure hierarchies are not the only ways to organise a case library, as we shall see in the next sub-subsection.

3.2.2.3 Other Ways to Organise a Case Library

Section 8.3 of Kolodner [39] suggests *shallow indexing* (*inverted indexing*) as a variation of linear case library organisation. Each attribute has, for each value of that attribute that occurs in the case library, a pointer to every case in the library which has that value for that attribute, so retrieval becomes a table lookup. However, this retrieval is based on only one attribute at a time; a way is needed of selecting a case from the list of cases it returns (most likely a linear search of the list[5]). It is certainly an improvement in efficiency over straight linear retrieval, but there is a difficulty in which attribute to choose for the lookup. A poorly-discriminating attribute will return many cases to select from. Moreover, the best case could be missed simply because it does not match on that attribute, even though it might match on all the others. This scheme is really only to be recommended if there is an attribute that any satisfactory case *must* match exactly.

---

5. Looking up on another attribute would not be helpful because the list that retrieved would have to be intersected with the first list - an O(n^2) operation.

Chatterjee and Campbell [14] describe a cache-based case organisation scheme. In this scheme, the case library is organised as a multi-dimensional matrix (each dimension corresponds to an index term for cases). Retrieval involves looking up in the matrix the cell which corresponds to the values of the index dimensions in the new situation. The cache is designed so that the best-matching case will be in the vicinity of that cell if not in the cell itself. The advantage of this scheme is that retrieval and insertion of cases is very time-efficient, at the obvious expense of space (the amount of space consumed by a multi-dimensional matrix increases exponentially with the number of dimensions). Retrieval time is uniform (unlike other schemes, it does not vary significantly according to the retrieval prompt or the number of cases in the library). The speed and uniformity of retrieval makes this scheme particularly appropriate for real-time processing, which must satisfy specified time constraints. However, the huge space requirement would probably render it quite unsuitable for CBR systems in more ordinary environments. The cache-based scheme also overcomes some of the awkwardness of hierarchical schemes, such as the difficulty of keeping networks optimal, the sensitivity to the ordering of index terms, the propensity to miss good partially matching cases, and the time cost of adding new cases.

Brown [9] describes a scheme in which the case library is organised as a network, and retrieval is an activation-passing process. His thesis is that this scheme combines flexibility with efficiency. He argues that conventional indexing in CBR is inflexible because the index terms are domain-specific, which is not helpful for remindings across domains. According to Aamodt and Plaza [1], CBR is traditionally thought of as being restricted to intra-domain reasoning, whereas *analogy-based reasoning* (or *analogical reasoning*) implies that problems can be solved by remembering a similar problem from another domain (*inter-domain* or *cross-contextual* reminding). However, Brown [9] claimed that the approaches to retrieval in analogical reasoning, though flexible, were too expensive computationally, and that his scheme achieved flexibility with efficiency, avoiding exhaustive search.

This subsection has discussed the retrieval from memory of cases which are potentially useful for solving a new problem, and the strongly related issue of how the case library is organised to support retrieval. However, some of these retrieval algorithms return multiple cases, competing to solve the problem, so it is necessary to select the one(s) which will actually be used. This entails matching each of these cases to the new situation in order to decide which to use, as discussed in the next subsection.

3.2.3 Matching and Selection

This subsection is concerned with selecting from a set of cases which ones to use in solving the current problem, an issue which arises if the system performs linear retrieval across the whole case library, or if retrieval algorithms (as discussed in Section 3.2.2) return multiple cases.

Selection involves matching each case in a given set to the current problem situation in order to decide which cases to use and which not to use. This may mean choosing the most usefully similar case (or cases) from a set of cases that are competing to solve the same problem. Alternatively, it could mean selecting - with no notion of competition - those cases from which useful inferences can be drawn, and rejecting those which are not useful for the situation at hand.

There are two important factors that determine how well a case matches the current situation (Section 8.1 of Kolodner [39]):

- the degree of importance of each attribute in the case;
- the degree to which the value of that attribute in the case is matched by the new situation.

In order to match a new problem to the ‘problem’ part of a case, it is necessary to know the correspondences between features in the new situation and features in the case. Sometimes it is obvious what corresponds to a case attribute in a new situation: if the situation is represented in the same way as the case, it is simply a matter of comparing attributes with the same name.

However, finding correspondences is not always that simple (Section 9.2.1 of Kolodner [39]) - if an attribute in the case does not correspond straightforwardly to an attribute in the situation, then the reasoner needs to find something in the situation which plays the same *functional role* as that attribute. For example, if CHEF was trying to match a problem specification which said “include broccoli, include beef” to a case in which the problem specification said “include chicken, include peas”, it would have to draw on its semantic knowledge to deduce that beef corresponds to chicken (because they are both meat) and broccoli corresponds to peas (because they are both vegetables). Section 9.2.1 of Kolodner [39] proposes commonsense reconciliation heuristics or the use of a causal model to determine which features of a case correspond to a new situation when there is no obvious correspondence.

When the reasoner knows the correspondences between attributes in the new situation and the case it is trying to match, it can then carry out the actual matching. A common way to do this is to use a *similarity metric*, which is a function that calculates the degree to which a case matches a new situation (Section 9.3.1 of Kolodner [39]). Similarity metrics are used to select the best of a set of cases that are competing to solve the problem at hand. A CBR system that uses similarity metrics typically selects the case(s) with the highest similarity score(s) for use in solving the new problem. As has already been mentioned in Section 3.2.1, similarity metrics are often realised as weighted polynomials or similar mathematical formulae.

It should be noted that there is nothing intrinsically correct about particular similarity metrics. They are just arbitrary numerical functions such that the better a case matches the current situation (in terms of its potential usefulness for solving the current problem), the higher similarity score it should attain. Nevertheless, the extent to which a similarity metric distinguishes useful cases from cases which are not so useful for solving the current problem has a very important effect on the overall quality of the case-based reasoner.

A CBR system could use a global similarity metric for all cases and for all the tasks for which it uses CBR. However, this might not be adequate, because it can be necessary to take context into account in judging similarity (Section 9.3.3 of Kolodner [39]). Different similarity metrics may be appropriate for different cases and for different reasoning goals. In particular, the coefficients of importance for case attributes can vary according to the reasoning goal to be satisfied, and can vary across sections of the case library or even across individual cases (Section 9.3.4 of Kolodner [39]).

Similarity metrics are one way of choosing between cases which are competing to solve a problem, but there are other ways. For example, Section 9.3.5 of Kolodner [39] suggests using *preference heuristics* to choose between cases. Evaluating a similarity metric for each case in isolation yields a fairly meaningless number for that case, with no explanation of why it matches as well as it does. Alternatively, cases can be compared to each other using preference heuristics which encode the criteria for choosing one case over another. For example, cases that address the current reasoning goal, cases that are more specific, cases that are easier to adapt and cases that have been used more frequently or more recently are preferred to cases that do not have these characteristics.

Similarity metrics and preference heuristics are used to select the best case(s) from a set of cases that are competing to solve the same problem. However, not all problems are of this nature - sometimes it is a matter of whether each case should be applied or not, rather than which cases are best to apply. For example, the problem of understanding constraints on information models, which is the subject of this thesis, entails applying all of the cases that match and none of the cases that do not match, in order to extract just those higher-level constraints that are appropriate to extract from the input model.[6] So matching a case which suggests extracting a higher-level constraint either succeeds or fails, rather than returning a degree of match.

This subsection has discussed the matching of cases to a problem situation, in order to decide which cases to use to solve the problem. This whole section has been concerned with retrieving cases from memory which will be used to solve the problem. The next section considers how the selected case(s) will be used to actually solve the problem.

3.3 Using the Retrieved Cases: Adaptation and Repair

Section 3.2 discussed the retrieval from memory of a case (or cases) suitable for solving the reasoner's current problem. This section considers how the retrieved case(s) can be used to actually solve the problem. The old case's solution serves as a ballpark solution to the new problem, but it often has to be *adapted* to compensate for the differences between the old and new problem situations.

The task of adaptation is to figure out what needs to be changed in the old solution to get it to fit the new situation (Section 3.3.1), and to carry out the necessary changes, resulting in a satisfactory solution to the new problem (Section 3.3.2).

3.3.1 Identifying What Adaptation is Needed

One way to see what has to be adapted in an old solution is to consider the differences between the old problem specification and the new (Section 12.1.1 of Kolodner [39]). These differences can be kept track of during the matching of an old case to the new situation (see Section 3.2.3), and hence the system will know what to compensate for at adaptation time.

6. The luxury of extracting all higher-level constraints that can be extracted from a model can be very costly. Systems that *use* higher-level constraints for a particular task (such as transferring constraints for schema-to-schema mapping (see Section 1.3.2)), will not need to extract higher-constraints indiscriminately, but since the system presented in this thesis merely extracts higher-level constraints for the sake of understanding the constraints on the input model, it has no basis for omitting to extract applicable higher-level constraints. Extraction control is an advanced research issue for future development (Appendix G, Section G.3.3).

However, as good as it sounds in principle, using the differences encountered during matching to guide adaptation can be inefficient if the case descriptions are very large and if the differences that are important for adaptation are actually derived attributes that are expensive to compute. In such circumstances it may be better to direct adaptation in other ways, such as having a *checklist* of standard tests for differences that have to be adapted for (Section 12.1.2 of Kolodner [39]). Such checklists are especially helpful if they also associate an appropriate adaptation method with each standard test.

Another way to guide adaptation is to notice inconsistencies between the current problem and the proposed solution (Section 12.1.3 of Kolodner [39]). For example, JULIA (Section 2.3 of Kolodner [39]) achieves this by posting and propagating constraints, and using a reason maintenance system to notice any violations of these constraints.

Another way to identify what needs to be adapted is to *project* the outcome of the proposed solution in the current situation (Section 12.1.4 of Kolodner [39]), which can be done either by using a causal model, or in a case-based manner, or by simulation - these are three ways of anticipating what could potentially go wrong with the solution. An alternative is to actually carry out the solution in the real world, and adapt the solution using the feedback obtained from doing so (Section 12.1.5 of Kolodner [39]). Adaptation that is done in the light of projection is often called *repair*.

Having identified ways of identifying what needs to be adapted in this subsection, we now consider how to carry out the necessary adaptations.

3.3.2 Methods and Strategies for Performing Adaptation

This subsection discusses various methods for carrying out adaptations in order to make the solution fit the new problem.

A simple form of adaptation is *reinstantiation* (Section 11.1.1 of Kolodner [39]), in which values in the old solution are replaced with corresponding values from the new situation that are determined by matching the old and new problems. For example, if CHEF was required to create a recipe for chicken and peas based on a recipe for beef and broccoli, it would start by reinstantiating the beef and broccoli recipe by substituting chicken for beef and peas for broccoli (based on the correspondences found during matching, as discussed in Section 3.2.3). Reinstantiation is a common first step for systems which do adaptation.

A variation of reinstantiation is to have variables in cases which get bound to values in the new problem situation during matching by unification (Section 3.3 of Filer [26]). When the solution in the case is first instantiated as a ballpark solution to the new problem, the variables are replaced with their values (as determined during matching). This is a good alternative to reinstantiation when it is sensible to have variables in cases (i.e. when the lesson that the case teaches does not rely on particular values).

Parameter adjustment (Section 11.1.2 of Kolodner [39]) is an adaptation technique where the values of attributes in the solution are changed due to differences in attribute values in the old and new problem situations. It is useful for adapting attributes which are *quantitative* rather than qualitative. Parameter adjustment typically entails scaling numerical values by a multiplicative factor, for example doubling the quantities of each ingredient in a recipe in order to make twice as much. This adaptation process is guided by specialised *parameter adjustment heuristics*: rules which tell the reasoner how to adapt values in the solution based on differences between values in the old and new problems.

Local search (Section 11.1.3 of Kolodner [39]) involves replacing a component in a solution with a similar one that is found close by in a semantic network, which is structured as an abstraction hierarchy such that conceptually similar items are near to each other. An example of local search adaptation would be replacing lasagne with vegetarian lasagne when adapting a meal with meat in it for vegetarians. Unconstrained local search can be inefficient, so it may need guidelines for walking the hierarchy and a cut-off to bound the time-cost of adaptation.

Section 2.6.2 of Riesbeck and Schank [65] discusses a technique called *abstraction and respecialisation*, which is a special case of local search. Abstraction and respecialisation entails going up in the abstraction hierarchy to the parent of the item that is to be replaced, and then down to a sibling node with which to replace the original item. More generally, the technique of abstraction and respecialisation involves going up the hierarchy until a node is encountered which does not have the problem that the item to be replaced has, and then going down a different branch of the tree to find a replacement.

Query memory (Section 11.1.4 of Kolodner [39]) is an alternative to local search where the reasoner constructs a partial description of a suitable substitute and searches globally (e.g. from the top of the abstraction hierarchy) to find an item which fits that description. For example, to find a substitute for lasagne, query memory would search the space of pasta dishes.

Another alternative to local search is *specialised search* (Section 11.1.5 of Kolodner [39]), which uses specialised heuristics to tell it *how* to search, which is more efficient than general methods when such knowledge is available.

Yet another alternative to local search is *case-based substitution* (Section 11.1.6 of Kolodner [39]), where the reasoner looks for suitable substitutes in similar cases - sometimes, cases are the places where substitutes are to be found. For example, to find a substitute for lasagne, case-based substitution would search the space of cases that include pasta dishes (as opposed to the space of pasta dishes).

Section 11.1 of Kolodner [39] classifies the above six methods cited from her book - reinstantiation, parameter adjustment, local search, query memory, specialised search and case-based substitution - as *substitution* methods. This implies replacing some component in the old solution. The next two methods that will be discussed are classified as *transformation* methods (Section 11.2 of Kolodner [39]). This implies adding or deleting components to or from the old solution. For example, a meal with lasagne could be adapted for vegetarians by *substituting* vegetarian lasagne for the lasagne, and the lasagne itself could be adapted for vegetarians by *transforming* it to remove the meat (or alternatively by substituting tofu for the meat).

Commonsense transformation (Section 11.2.1 of Kolodner [39]) uses heuristics to transform the old solution, such as deleting a component which is violating constraints and is not necessary to the solution, or adding a component to satisfy a goal in the new problem.

Model-guided repair (Section 11.2.2 of Kolodner [39]) transforms the old solution using a causal model of the relationships between components. The solution is transformed according to the differences between the new problem and the problem in the case, as determined during matching. As mentioned in Section 3.2.3, a causal model can be used to determine correspondences between components so that the differences can be evaluated. Model-guided repair takes this a step further by associating heuristics with classes of differences, which prescribe how to adapt for those differences.

The adaptation methods discussed so far in this subsection are general-purpose, domain-independent methods. However, special-purpose, domain-specific adaptation and repair heuristics (Section 11.3 of Kolodner [39]), when available, can produce better results. Section 2.6.2 of Riesbeck and Schank [65] calls this *critic-based adaptation*. For example, CHEF (Section 2.1 of Kolodner [39]) uses *object critics*, which are special-purpose rules for adapting recipes such as inserting a step to chop broccoli into chunks when peas are replaced with broccoli. This rule is associated with broccoli, and is fired whenever broccoli is encountered and the rule is applicable.

Derivational replay (Section 11.4 of Kolodner [39]) adapts a solution by recomputing it with different parameters, rather than fixing the final solution stored in the case, as all the other methods considered here do. Derivational replay is also known as *derivational adaptation* (and all the other methods are called *structural adaptation*) (Section 2.6.2 of Riesbeck and Schank [65]). To support derivational adaptation, cases must record not only the solution to an old problem, but also the chain of operators that was used to derive that solution. These operators are then rerun in the new situation (which should be faster than searching for operators to apply). Derivational adaptation is often used by humans to solve problems in physics, calculus and probability, where remembering how a similar problem was solved is more useful than just remembering the solution. It is useful for many planning problems.

A variation of adaptation is *merging* the solutions of several old problems to solve a new problem (Chapter 14 of Kolodner [39]). For large problems, a single case is often insufficient for solving the whole problem, so different cases can be used to solve different parts of it. For example, one case might be used to suggest a framework for the solution, and other cases could be used to solve subproblems within that framework. Pieces of cases, called *snippets*, can be used for the purpose of solving subproblems (Section 5.4.1 of Kolodner [39]).

This section has discussed how cases can be used to solve new problems. In particular, it has discussed how the solutions to old problems can be adapted to solve the new problems, including a discussion both of determining what adaptation needs to be done (Section 3.3.1) and methods for carrying out that adaptation (Section 3.3.2). It usually fits together in the following steps (Section 12.4 of Kolodner [39]):

- Firstly, the old solution is reinstantiated;
- The solution is then adapted using whatever methods are appropriate (preferring methods that are easier and more specific);

- The solution is then evaluated, projecting the outcome and repairing[7] the solution as necessary to fix any mistakes;
- Further adaptation might be necessary to compensate for problems caused by earlier adaptation;
- The solution is then tried out ‘for real’, collecting feedback and repairing if necessary.

Adaptation is the major problem to be solved in CBR. In spite of the impressive-sounding methods referred to in this section, it has proven difficult to implement in practice. Section 16.3.3 of Kolodner [39] admits that guidelines on adaptation are lacking, particularly special-purpose adaptation heuristics. According to Section 2.3.5 of Watson [84], many CBR systems in the real world do not even do any automated adaptation at all - either the problem is so simple that it needs no adaptation, or adaptation is performed by humans. Section 9.4 of Watson [84] suggests that, for many practical CBR systems, adaptation should not be automated as a matter of principle! However, it is the Author’s strong belief that the full potential of CBR for machine intelligence can never be realised without automating adaptation as well as retrieval.

The next section concludes this chapter with a discussion of the advantages and disadvantages of CBR, and a justification of its applicability to code generation, code understanding and understanding constraints on information models.

3.4 Advantages and Disadvantages of CBR

This section discusses the advantages and disadvantages of CBR, first in general (Section 3.4.1), and then in the context of code understanding and generation, in order to justify the application of CBR to the understanding of constraints on information models (Section 3.4.2).

7. Repair is just adaptation that is carried out in response to feedback which shows that a solution is faulty (Section 11.3 of Kolodner [39]).

3.4.1 Advantages and Disadvantages of CBR in General

The obvious advantage of CBR over more traditional problem-solving methodologies such as hardwired algorithms and rule-based systems is that it saves time, because intuitively it should be much faster to solve problems by modifying the solutions to similar old problems than to solve them from scratch (Section 1.5.2 of Kolodner [39]). Effectively, CBR caches chains of inference - what would take many steps to solve using rules can be encoded as a single case (Section 3.2 of Kolodner [39]).

However, as was seen in Section 3.2.2, the effort needed for retrieval and adaptation can militate against the speed-up that CBR promises. As mentioned in Section 3.2.2.1, this is known as the *utility problem* (Smyth and Keane [69]). The obvious way to alleviate the utility problem is to implement retrieval algorithms which are efficient, and effective in terms of minimising the cost of adaptation by finding close matches. The utility problem also depends on the nature of the task CBR is being applied to. Easy, well-defined problems that can be solved quickly using a hardwired algorithm are unlikely to derive much benefit from CBR, but large, difficult problems, which take a long time to solve from scratch, should really reap a big reward from using CBR. For example, there are no known algorithms for solving NP-complete problems deterministically in polynomial time (Section 9.2 of Floyd and Beigel [27]). This means that any algorithms for finding *optimal* solutions to such problems will have at least exponential time complexity, which is utterly infeasible for large instances of such problems. However, *heuristic* methods, of which CBR is an example (Section 15.1 of Kolodner [39]), can find solutions in polynomial time which, though not guaranteed to be optimal, tend to be very good in practice (Section 2.2.2 of Rich and Knight [64]). CBR achieves this by searching the space of actuality (what has happened in the past) rather than the space of possibility (Section 1.1.2 of Brown [9] attributes this to a verbal quote from Kristian Hammond).

Although the above discussion implies that the solutions proposed by CBR can be sub-optimal, a major advantage of CBR is that it can propose solutions in *weak-theory domains* (Section 15.1 of Kolodner [39]): domains which are so poorly understood that it would not be possible to solve problems competently, or even at all, without the sort of heuristic guidance that cases can provide. Examples of weak-theory domains to which CBR has successfully been applied are law (e.g. HYPO (Section 2.4 of Kolodner [39])), medicine (e.g. CASEY (Section 2.2 of Kolodner [39]) and PROTOS (Section 2.5 of Kolodner [39])), and curing airplane parts in an autoclave (CLAVIER (Section 2.6 of Kolodner [39])). Such domains cannot be characterised accurately and completely by a set of general rules nor a causal model, but there are plenty of cases which exemplify what has and has not worked well in the past.

Other advantages of CBR are that it can help to evaluate solutions when there are no algorithmic ways of doing so (Section 1.5.2 of Kolodner [39]), and it can avoid mistakes that were made in the past by recalling cases that warn of potential failures (Section 14.2 of Kolodner [39]).

That a CBR system can *acquire* cases which suggest solutions to problems, and cases which warn of the potential for failure, implies that the system can *improve* over time both its *efficiency* (speed of problem solving) and its *competence* (the range of problems that it is capable of solving, and the mistakes that it is able to avoid) (Section 1.1 of Kolodner [39]). This is in contrast to traditional expert systems, which are brittle because their efficiency and competence are fixed - they are stuck at their particular plateaux of expertise (Section 1.2.2 of Brown [9]).

There is empirical evidence that CBR systems are easier to build and maintain than their non-CBR counterparts (Section 3.4.1 of Kolodner [39]). For example, Section 9.3.2 of Watson [84] implies that it is easier to add a case to a case base than to add a rule to a rule base. This is because cases tend to be large pieces of composed knowledge, whereas rules tend to be smaller pieces of knowledge that have to be composed into chains of inference during reasoning (Section 3.4.1 of Kolodner [39]), so rules interact with each other more than cases. Furthermore, CBR alleviates the knowledge-elicitation bottleneck (Section 1.6 of Watson [84]) due to its ability to acquire cases automatically from its own experiences, and also due to the fact that it is often easier to elicit cases from human experts than to elicit a set of rules which covers the domain (Section 1.6.2 of Kolodner [39]).

That case-based systems are relatively easy to build makes CBR good for rapid prototyping, even if the CBR system is not to be the final implementation. In the future, it might be feasible to compile a mature case base into a hardwired algorithm (a task for which automatic programming would be appropriate) which would be more time-efficient (because the knowledge from the cases would be accessed directly via conditional tests) and space-efficient (by eliminating the redundancy of storing similar cases), though it would of course lose its flexibility and its ability to learn from experience.

The obvious disadvantage of CBR is that large case libraries take up a lot of space (Section 15.1 of Kolodner [39]). This in turn can exacerbate the utility problem if retrieval time depends on the number of cases in the library. The following are some ways to reduce the number of cases stored in the case library:

- Adaptation avoids having to store all variants of a case in the library (Section 3.2 of Filer [26]), by enabling variants of cases to be derived from each other. Having to store all possible variants of a case in the case library could lead to gigantic combinatorial explosion!
- *Case retention heuristics*[8] control whether a newly-solved problem should be inserted as a new case in the library (Section 16.1 of Kolodner [39]). Instead of retaining cases indiscriminately, a case is only retained if it contributes problem-solving capabilities that

8. *Case retention heuristics* and *case deletion heuristics* are the Author's own terms for notions which crop up occasionally in CBR without having widely-accepted names. For example, page 567 of Kolodner [39] refers to "the controls one might put on accumulation of cases", and Smyth and Keane [69] use the term "case deletion policy".

could not easily be derived by adapting the other cases in the library.

- *Case deletion heuristics* remove cases from the library that do not contribute to the reasoner's capabilities beyond the other cases in the library (Smyth and Keane [69]).

CBR will not succeed if cases are used inappropriately, so the selection process must validate the applicability of the best case retrieved to check that it *is* suitable for solving the problem. CBR has also been criticised for it being difficult to evaluate the inferences it makes because the meaning of the data structures it uses is unclear from a logical viewpoint (Chapter 7 of Pratt [54]), but the real world is not that logical anyway! There are some problems that are very hard to propose solutions to (e.g. NP-complete problems), and yet it is easy to verify that a solution is indeed a solution (it can be verified in polynomial time) (Section 9.2 of Floyd and Beigel [27]). CBR could be used to hypothesise a solution quickly, and then sound logical inference could be used to evaluate that solution. For example, medical doctors often hypothesise diagnoses by remembering similar combinations of symptoms they have seen in the past, and then use logical medical judgement to check their hypothesised diagnoses (Section 1.5.1 of Kolodner [39]).

The conclusion of this subsection is that CBR is a very attractive methodology for building knowledge-based systems, and a very promising step forward for machine intelligence. The following are the main reasons for this:

- The ability to solve problems in weak-theory domains;
- The flexibility and efficiency of not having to solve a problem from first principles when a similar problem has already been solved;
- The ability to learn from experience as a natural by-product of reasoning (Section 1.1 of Kolodner [39]), and thereby become more efficient and competent over time;
- The relative ease of building and maintaining case-based systems.

The only caveat is that CBR is not universally appropriate for solving all computing problems. CBR is only worth applying to problems for which the advantages of CBR outweigh the disadvantages. It can be worthwhile for problems which are somewhat vague and not well-defined, and for weak-theory domains, as long as there are cases which teach lessons that help to achieve reasoning goals. On the other hand, CBR is still a *symbolic* artificial intelligence technique, and some problems (e.g. face recognition) are so theory-weak that *subsymbolic* techniques such as neural networks (Pao [52]) are currently the best-known way to solve them. CBR is likely to have a better payoff in terms of efficiency for large problems, rather than small problems which can be solved quickly using non-CBR methods.

This subsection has discussed the advantages and disadvantages of CBR in general. The following subsection considers how these might apply to code generation and code understanding, and in particular to understanding constraints on information models, in order to justify the application of CBR to these domains.

3.4.2 The Applicability of CBR to Code Generation and Understanding

This subsection discusses the applicability of CBR to code generation and code understanding in general, and to the understanding of constraints on information models in particular, which is the aim of this thesis. The goal of this subsection is to justify the application of CBR to these tasks, and to point out some of the benefits and contributions of CBR over non-CBR approaches to these tasks.

Understanding the constraints on information models is a special case of code understanding. Information models are written in formal languages such as EXPRESS [34], and understanding EXPRESS code is fundamentally similar to understanding code written in programming languages. Both involve extracting higher-level knowledge from the code itself, although in program code the emphasis is often on flow of control rather than more ‘static’ semantics such as the constraints implied by an information model. However, constraint-like conditions *do* manifest themselves in program code, e.g. as the conditions of 'if-then' statements.

The most important advantage of CBR, from the point of view of code understanding and code generation, is its ability to solve problems in weak-theory domains. Automatic programming, especially in its full generality, is theory-weak. The links between a requirements specification and code which implements that specification are not well understood. This makes it extremely difficult to write a general algorithm to take as input a specification and produce as output a program. It also makes it very difficult to write a general algorithm to take as input a program and produce as output a specification for that program which captures the semantic intent of that program at a very high (domain-oriented) level of abstraction. However, there are plenty of cases in the history of computer science of programs, whose semantic intents are known (to humans). This discussion implies that CBR should be the most successful way to tackle code understanding and code generation. Cases which associate code with higher-level knowledge about that code could be useful for both understanding and generating code.

Software reuse has been identified in the software engineering community as having an important role in software development (as discussed in Section 2.4). New software is developed by reusing old software, rather than developing all software from scratch. This approach clearly stems from the same mentality as CBR, but most research in this area is not billed as CBR. Also, reuse is, for the most part, a manual process - there are software tools to support human programmers, but there appears to have been little in the way of an attempt at automatic programming in the reuse community. It is the Author's belief that this could be achieved by integrating reuse and CBR research (though we are still a long way from achieving automatic programming in its full generality). There has been little research so far into linking CBR to reuse explicitly (see for example Fernández-Chamizo, González-Calero et al [23], and Maguire, Szegfue et al [45]).

There appears to have been little in the way of explicit cross-fertilisation of CBR and automatic programming, although Smyth and Cunningham [68] present a CBR system for software design called Déjà Vu. This achieves automatic programming in the limited domain of controlling autonomous vehicles in a steel mill. Interestingly, Déjà Vu uses a hierarchical CBR process which involves the recursive use of CBR for adaptation. This is appropriate because programming is hierarchical in nature, with each module implementing a specification for the modules that call it, which in turn implement specifications.

Of the techniques discussed in Chapter 2, clichés clearly have much affinity with CBR. Clichés are useful both for code generation (see Section 2.1) and code understanding (see Section 2.2.2). Indeed, it is stated in Rich and Waters [62] that overlays (which associate specification clichés with their corresponding implementation clichés) are used for both analysis and synthesis of program code. Overlays play an equivalent role to cases in CBR. However, overlays do not admit of any partial matching and adaptation, which makes cliché-based approaches to code understanding and generation less flexible than they could be if CBR were used.

As for the other techniques discussed in Chapter 2 - namely views (Section 2.2.3) and program segmentation (Section 2.3.1) - it is not so much a case of how CBR should replace them as how CBR could make use of them. Extracting views from low-level source code constitutes a semi-automatic understanding of the code: it is up to the human programmer to decide what views are to be extracted and what to do with them (the motivation for views is typically for humans to understand code so that they can modify it). The extraction of views could be a useful building block in a case-based automatic programming system which controlled the extraction and subsequent use of views. Program segmentation, which isolates the external functionalities of a program from the source code so that they can be packaged into reusable modules, could be a useful aid to seeding the case library of such an automatic programming system.

The following chapters will focus on understanding constraints on information models, and will present a CBR system which operationalises this by extracting higher-level constraints (HLCs) from EXPRESS models. In this system, a case associates a fragment of EXPRESS code with an HLC which captures the abstract semantics of the constraint implied by that fragment. It is analogous to the way an overlay associates an implementation cliché with a specification cliché.

Extracting HLCs from an EXPRESS model is theory-weak, but there are plenty of cases of extracting particular HLCs from particular EXPRESS fragments. As discussed in Section 3.4.1, this gives CBR an advantage over hardwired algorithms and non-CBR symbolic artificial intelligence techniques. The task is theory-weak in several senses:

- It is unclear what kinds of HLCs are appropriate to extract from EXPRESS models. The approach taken in this thesis is primarily purpose-driven, i.e. classes of HLC are identified (manually) by analysing worked examples of schema-to-schema mapping in order to intuit what HLCs are useful to support a comparative understanding of the semantics of constraints on two models.
- It is impossible to characterise in full generality all the different combinations of code fragments from which each kind of HLC could be extracted, and all the HLCs which could be extracted from a given code fragment. The possibilities are practically infinite. The best strategy seems to be to retain as cases those associations (of EXPRESS code fragments with HLCs) that have been experienced to be useful for understanding the constraints on real models, rather than to attempt to encode all possible associations. It would be too difficult to write a general algorithm, causal model or set of rules to cover all possible models, because one can always contrive example models which the system cannot handle. It is better to have a case library which covers a sensible range of features in input models, which can be extended on an as-needed basis by acquiring new cases.
- It is unclear how to decide which HLCs to extract, of all those that could be extracted from a particular input model. Rather than extracting all such HLCs indiscriminately, extraction could be controlled (c.f. Footnote 6 (page 91)).

Another benefit of CBR for the task of understanding constraints on information models is that, by nature, it contributes to the data-driven approach which was identified as an objective in Section 1.5. This is because cases constitute explicit knowledge, as opposed to knowledge which is hardwired into the program code of the constraint-understanding system. Cases can be held in data files, in a form which is much easier to understand than if this knowledge were buried in the program code. It is easy to seed the system with new cases because they can simply be typed into data files, without having to modify or recompile the program.

The next chapter will introduce the aforementioned constraint-understanding system, describing the architecture of the system and its external functionality.