Extracting Constraint Knowledge from Code: A Case-Based Reasoning Approach

Andrew Broad[1] and Nick Filer
Department of Computer Science, University of Manchester,
Manchester, England
{broada/nfiler}@cs.man.ac.uk
http://www.cs.man.ac.uk/~broada/

Abstract

The underlying motivation for the authors' research on constraint understanding is *comparative constraint understanding*. The same constraint can be expressed in different ways in different models, and a comparative constraint-understanding system would be able to assess the extent to which two models have equivalent constraint semantics. Understanding constraint semantics can be used to support tasks such as mediation and mapping.

This paper presents a prototype method for understanding constraints in code. In particular, it focuses on understanding the constraints in EXPRESS information models, for which an experimental computerised system has been implemented. A case-based reasoning approach is taken, in which cases suggest so-called *higher-level constraints* to extract from matching fragments of EXPRESS code.

1 Introduction

This paper is about *code understanding*. This means extracting knowledge from code written in a formal language. In particular, this paper focuses on extracting knowledge about *constraints* from code. Constraints are conditions that must not be violated, and can feature, in various guises, in all kinds of code.

The literature on code understanding has tended to focus on program code,

1. This work was supported by the Engineering and Physical Sciences Research Council (EPSRC).

calling it *program understanding* [1] or *reverse engineering* [2]. Figure 1 shows a fragment of program code which searches an array for a given goal element. The conditional expressions in the `while` and `if` statements can be thought of as constraints on when the bodies of these constructs can be executed.

```
int index= 0;
while ((index < length) && (array[index] != goal)) {
   index= index + 1;
}
if (index < length) {
   System.out.println(goal + " found at array[" + index
   + "]");
} else {
   System.out.println(goal + " not found");
}
```

Figure 1. Some program code written in Java[2] [3].

Code understanding need not be limited to program code, however. It can be generalised to code written in *any* formal language. For example, Figure 2 shows excerpts from two formal specifications of a stack: a model-based specification, and an algebraic specification.

- The *model-based specification* (Figure 2(a)) specifies two operations on a stack, `push` and `pop`. It gives their *signatures* and *preconditions* (constraints on the acceptable inputs to these operations), and *postconditions* (constraints on the output).
- The *algebraic specification* (Figure 2(b)) specifies these two operations in a different way. Two axioms specify their behaviour in relation to each other (as opposed to specifying the operations themselves), and to a third operation, `Create` (which returns an empty stack).

```
push (s : stack, x : int)
   pre true
   post s' = x::s
pop (s : stack)
   pre length(s) > 0
   post s' = tail(s)
```

(a) The push and pop operations from a model-oriented specification (adapted from Figure 11.6 in [4]).

```
Pop(Create) = Create
Pop(Push(s,i)) = s
```

(b) Two axioms from an algebraic specification (adapted from Figure 11.5 in [4]).

Figure 2. Some code from two formal specifications of a stack.

---

2. *Java* is a trademark of Sun Microsystems, Inc.

Figures 2(a) and 2(b) both specify the same behavioural constraints on `push` and `pop` in different ways. The constraints are semantically equivalent - implicitly, at an abstract level - in both specifications (except that Figure 2(a) implies that an empty stack cannot be popped, whereas Figure 2(b) *does* sanction popping an empty stack). The authors are particularly interested in *comparative* code understanding, where the same semantics can be expressed in different ways, and in assessing the extent to which the semantics are equivalent.

Information models and *database schemata* are two other kinds of code. They specify integrity constraints on the entities that they model, and so are particularly appropriate to the study of understanding constraints in code. Section 2 of this paper will focus on understanding the constraints in EXPRESS[3] information models. A simple example of an EXPRESS information model is given in Figure 3.

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

Figure 3. A simple EXPRESS information model.

The model in Figure 3 defines an entity `person` (which represents a living person):

- The entity has two *attributes*, `name` and `age`. They are not specified to be optional, so every instance of this entity must have values for these attributes, which must be a string and an integer respectively. These can be thought of as constraints imposed by the structure of the model.
- The entity has a *domain rule* called `age_must_be_sensible`. It constrains the age of a person to be at least zero, and less than 130 years.

Other types of code which can contain constraints include:

- *Database queries* can be thought of as constraints on the records that they retrieve;
- In *rule-based systems*, the antecedents of rules are constraints on when the rules can be fired.
- *Constraint languages* encode networks of constraints (such as adders and multipliers), through which information can propagate (these constraints are tight enough to be able to derive new information from existing information);
- Even languages themselves can be thought of as constraints, which legal strings must satisfy.

3. EXPRESS [5] is an international standard (ISO 10303-11) information modelling language.

2 Understanding Constraints in Information Models

The authors' research has focused on understanding the constraints in information models written in EXPRESS. An *information model* is a precise formal specification of the entities in a domain, the relationships between those entities, and the constraints on those entities and relationships [6]. The purpose of an information model is to communicate the semantics of a domain, often with no particular regard for the implementation of applications in that domain.

A *constraint* in an information model is any condition that must not be violated by instances of entities in a repository that conforms to the model. This includes both *explicit conditions* (e.g. domain rules) and *implicit conditions* (e.g. an entity instance must have a value for each of its mandatory attributes, and the value for an attribute must be of the type of that attribute).

```
SCHEMA complicated_people_model;
   ENTITY person ABSTRACT SUPERTYPE OF (ONEOF (man,woman));
      name : STRING;
      age : INTEGER;
      father : OPTIONAL man;
      mother : OPTIONAL woman;
   DERIVE
      orphaned : BOOLEAN
         := NOT EXISTS(father) AND NOT EXISTS(mother);
   UNIQUE
      name;
   WHERE
      age_must_be_sensible:
         (age >= 0) AND (age < 130);
      parents_must_be_older_than_self:
         (father.age > age) AND (mother.age > age);
   END_ENTITY;

   ENTITY man SUBTYPE OF (person);
      wife : OPTIONAL woman;
   END_ENTITY;

   ENTITY woman SUBTYPE OF (person);
   INVERSE
      husband : SET [0 : 1] OF man FOR wife;
   END_ENTITY;
END_SCHEMA;
```

Figure 4. A more complicated EXPRESS information model.

Section 1 has already given an example of an EXPRESS information model and discussed some of the constraints in it. Figure 4 gives a more complicated toy example to illustrate a broader range of constraints in EXPRESS. The constraints, other than the kinds that were covered in the previous example, are:

- The entity `person` is an *abstract supertype* of one of `man` and `woman`. This means that every person must be either a man or a woman. The fact that `person` is abstract rules out the possibility of a person being neither a man nor a woman. The `ONEOF` expression rules out the possibility of a person being both a man and a woman.
- The entities `man` and `woman` are *subtypes* of `person`. This means that every instance of `man` or `woman` must be linked in the subtype-supertype relationship to an instance of `person`. The subtypes `man` and `woman` can be thought of as *inheriting* the attributes and constraints from their supertype, `person`.
- The attributes `person:father`, `person:mother` and `man:wife` are *optional attributes*, which means that they are not constrained to have values (they can be 'indeterminate' instead), but if an optional attribute does have a value, that value must be of the type of that attribute. The types of these attributes happen to be entities, which means that their values will be references to entity instances - this establishes association relationships between entities. The model is implicitly restricted to people who are alive at a particular moment in time, so if `father` or `mother` is indeterminate, this could be taken to mean that that parent is dead.
- The entity `person` has a *derived attribute*, `orphaned`. The type of this attribute is `BOOLEAN`, which means its value will be either `TRUE` or `FALSE`. A person is orphaned if both of his/her parents are dead, i.e. if both `father` and `mother` are indeterminate (`EXISTS` is a built-in predicate which returns `true` if an attribute has a value and `false` if it is indeterminate).
- The attribute `person:name` is constrained to be *unique*. That is to say, no two instances of `person` may have the same value for `name` in this simplified model.
- The entity `woman` has an *inverse attribute*, `husband`. The type of `husband` is a set of at least zero and at most one instances of `man`. That is to say, a woman can either have no husbands or one husband. The inverse attribute constrains the woman's husband (if she has one) to have that woman as his wife. The attributes `man:wife` and `woman:husband` together establish a one-to-one relationship between the entities `man` and `woman`.

3 Higher-Level Constraints

The first author has implemented a *constraint-understanding system (CUS)* for EXPRESS models. This work is described in detail in [7].

The CUS 'understands' the constraints in the input model by extracting *higher-level constraints (HLCs)*. Each HLC makes explicit some aspect of the model's constraint semantics, representing this knowledge at a higher level of abstraction than the EXPRESS code itself.

For example, the CUS extracts two instances of the `bound` HLC class from the domain rule `age_must_be_sensible` in Figure 3. One HLC, extracted from (age >= 0), encodes the fact that there is a lower bound of zero on age, and that the bound is inclusive and discrete. The other HLC, extracted from (age < 130), encodes the upper bound of 130 on age, which is exclusive and discrete.

Having extracted the two bounds, the CUS then extracts an instance of the range HLC class to encode the fact that there is a numeric range constraint on age. So the CUS can extract HLCs directly from the input EXPRESS model, and then more. It can build up a hierarchy of HLCs at increasing levels of abstraction.

Figure 5 shows what these HLCs actually look like when represented formally. They are represented as frames in a language called FIL.[4] Figure 5 is part of a much larger network of frames representing the input model and the HLCs extracted therefrom. Thus, the input model is represented as data which the extraction engine operates on. Figure 5 is annotated with comments to indicate what the other frames referred to represent.

```
INSTANCE bound#19;
   extracted_from = {binary_operation#23}; (* age >= 0 *)
   kind = lower;
   inclusive = true;
   discrete = true;
   variable = explicit_attribute#110; (* age *)
   limit = 0;
END_INSTANCE;

INSTANCE bound#20;
   extracted_from = {binary_operation#24}; (* age < 130 *)
   kind = upper;
   inclusive = false;
   discrete = true;
   variable = explicit_attribute#110; (* age *)
   limit = 130;
END_INSTANCE;

INSTANCE range#2;
   extracted_from = {bound#19, bound#20};
   lower_bound = bound#19;
   upper_bound = bound#20;
   variable = explicit_attribute#110; (* age *)
END_INSTANCE;
```

Figure 5. Three HLCs extracted from the model in Figure 3.

4. *FIL* [7] stands for *Frame Instance Language*. Frames are actually instances of entities in EXPRESS models - EXPRESS is used as the frame class language. Any formal language can be represented by an EXPRESS model, and any code in that language can be represented in FIL as an ‘abstract syntax tree’ of frames that conform to the EXPRESS model.

The underlying motivation for the authors' research on constraint understanding is *comparative constraint understanding*. Whereas the current CUS just extracts HLCs from a single input model, a long-term aim is to be able to compare the semantics of the constraints in two models. The same constraint can be expressed in different ways in different models, and a comparative constraint-understanding system would be able to assess the extent to which two models have equivalent constraint semantics (and the idea generalises to all kinds of code, not just EXPRESS models). The purpose of comparative constraint understanding has driven the invention of HLC types to capture the common semantics of different ways of expressing constraints.

For example, compare Figure 4 with the alternative model of people given in Figure 6. This model represents gender by an enumeration type, rather than by subtypes of `person`. However, it still has the abstract constraint that an instance of person must have one of two properties to encode his or her gender. This abstract constraint is represented by a HLC called `exactly_one`, which the CUS can extract from the models in both Figure 4 and Figure 6.

```
SCHEMA flat_people_model;
   ENTITY person;
      name : STRING;
      age : INTEGER;
      gender : gender_type;
   END_ENTITY;

   TYPE gender_type = ENUMERATION OF (male, female);
   END_TYPE;
END_SCHEMA;
```

Figure 6. An alternative model of people.

Some of the other simple HLC classes recognised by the CUS are:

- `at_most_one`: This is like `exactly_one` except that it encodes the fact that instances of an entity are constrained to have at most one, rather than exactly one, of a set of properties. This HLC would be extracted from Figure 4 if the supertype were not abstract, and from Figure 6 if `gender` were an optional attribute.
- `non_null`: This HLC represents the fact that an attribute is constrained not to be indeterminate. It is extracted from all non-optional attributes (such as `name` and `age` in Figure 3), and also when an attribute is constrained by the `EXISTS` predicate to have a value.
- `equation`: This HLC encodes the fact that two (or more) expressions are constrained to be equal. It can be extracted from '=' operations in domain rules, and from derived attributes (such as `orphaned` in Figure 4).

- `adder` and `multiplier`: These HLCs encode the fact that three expressions x, y and z are constrained by the equations ($x + y = z$) or ($x * y = z$) respectively. They are extracted from an `equation`, one of whose expressions is a '`+`' or '`*`' operation respectively. These two HLCs are other examples of further HLCs (`adder`, `multiplier`) being extracted from HLCs already extracted (`equation`).

4 A Case-Based Reasoning Approach

The CUS takes a *case-based reasoning (CBR)* approach to understanding constraints.

CBR [8] is the enterprise of solving new problems by analogy with old ones, rather than solving all problems from first principles. Essentially, a *case* consists of an old problem and its solution. Cases are stored in a *case library.* To solve a new problem, a case-based reasoner *retrieves* a case in which the old problem is usefully similar to the new problem, so that the old solution can be *adapted* to solve the new problem. The new problem and its solution may then be stored in memory as a case, which can be used in turn to solve future problems.

4.1 The Applicability of CBR to Code Understanding

This section justifies the application of CBR to constraint understanding by justifying its application to the more general topic of code understanding.

One of the main advantages of CBR is that cases can suggest solutions to problems in weak-theory domains [8]. Domains such as medicine and law cannot be completely formalised by a hard-wired algorithm, a set of rules, or a general model. However, there are many *cases* of solving problems in such domains - experiences that are useful for solving similar problems in the future. Code understanding is also a weak-theory domain, in that the link between code and higher-level knowledge - or implementation and specification - is often ill-defined. However, there are plenty of cases of code from the history of computer science, for which a human understanding exists.

A key feature of CBR is that cases cover not just *normative* situations, but also *idiosyncratic* situations [8]. For example, a CBR system might be able to recognise a piece of code as being an implementation of a standard algorithm that differed in some way from the norm. The code might match a case only partially, and the CBR system might have to adapt the 'solution' accordingly. It might even record the experience as a case in memory, to help it to recognise similar situations in the future. Whereas a non-CBR system that relied on exact matching might fail to understand the code at all.

The most obvious advantage of CBR is that it should be more efficient to solve problems by adapting old solutions, rather than trying to solve all new problems from first principles [8]. A case can be thought of as several inferences chunked together, which can be used to solve a new problem in one step rather than many small inference steps. The case holds these inferences together so that the system does not have to search the space of possible inferences, which could lead to combinatorial explosion (code understanding is an NP-complete problem [9]).

Above all, case-based reasoners can learn from experience (by acquiring new cases), which occurs as a natural by-product of reasoning [8]. This enables a CBR system to become more efficient and more competent as it gains in experience, as opposed to traditional expert systems which are brittle because they cannot improve beyond their particular plateaux of expertise [10].

Further advantages of CBR are that it alleviates the *knowledge-elicitation bottleneck* [11] - it is often easier, in complex domains, to elicit cases than a complete set of rules. A CBR system is often easier to build and maintain than the equivalent rule-based system [8], which suggests that CBR is good for prototyping if not for final implementations as well (a case library could be compiled into a hard-wired algorithm for efficiency, at the expense of CBR's ability to learn from experience, and of the flexibility and comprehensibility of a data-driven system).

4.2 A Simple Application of CBR to Constraint Understanding

For code understanding, a case associates a fragment of code (the 'problem' part) with some higher-level knowledge about that code (the 'solution' part). Specifically, in our task of understanding constraints in information models, a case associates a fragment of EXPRESS code with a HLC (or more than one) that expresses the abstract constraint semantics of that fragment. Figure 7 shows a case for extracting an inclusive lower bound from an expression of the form (`attribute >= limit`).

This is the case that would be used to extract `bound#19` in Figure 5. The CUS retrieves this case whenever it encounters a `binary_operation` frame in (the internal representation of) the input model. It then proceeds to match that frame to the top-level problem frame (`binary_operation#1`) in the case. For the case to match the input model, every slot in `binary_operation#1` must match the corresponding slot in the `binary_operation` frame from the input model. When a slot filler is a reference to another frame in the case, that frame is matched recursively to the corresponding frame in the input model. Variables (such as `?limit`) are matched by unification.

If the 'problem' part of the case (i.e. all frames in the transitive closure of the `case_frame`'s `problem` slot) successfully matches the input model, then the frame(s) in the 'solution' part is reinstantiated for, and attached to, the input model. Any references the solution makes to frames in the case are replaced with the frames they matched in the input model, and variables are replaced with the values they were bound to during matching.

The CUS's cases are all stored in a case library. Cases encode how to extract HLCs of particular classes (such as those listed in Section 3) from an EXPRESS model. There is usually more than one case for each HLC class, either because the semantics of that HLC can be expressed in different ways in EXPRESS, or because of combinatorial variations in its parameters (there are 16 cases to extract bounds, which can be either lower or upper, either inclusive or exclusive, either discrete or continuous, and the relational operation can be written either way round, e.g. `a > b` is equivalent to `b < a`).

Cases can include HLCs, as well as EXPRESS code, in their problem parts. These cases are used to extract further HLCs from HLCs already extracted, such as a range from two bounds, or an adder from an equation.

```
INSTANCE case_frame#1;
   problem = binary_operation#1;
   solution = bound#1;
END_INSTANCE;

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

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

INSTANCE explicit_attribute#1;
   tipe[1] = simple_type(integer_type);
END_INSTANCE;

INSTANCE literal_expression#1;
   literal_value = integer_literal(?limit);
END_INSTANCE;

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

INSTANCE bound#1;
   kind = lower;
   variable = explicit_attribute#1;
   limit = ?limit;
   inclusive = true;
   discrete = true;
   extracted_from = {binary_operation#1};
END_INSTANCE;
```

1. `type` is a reserved word in EXPRESS.

Figure 7. A case (written in FIL) for extracting a bound HLC.

The case in Figure 7 is actually very rule-like, in that it associates a tiny fragment of EXPRESS code with a single HLC, and entails no partial matching and adaptation other than frame substitution and variable binding. However, there could be more 'case-like' cases, which are larger and do require partial matching and adaptation. An example might be interpreting whether an instance of a given entity actually represents an individual object or a class of individual objects (e.g. whether an instance of an entity `book` represents a book or a copy of a book). There is no hard-and-fast rule for deciding this; there might be clues in the model, but any general rule is likely to have exceptions.

5 Related Work

Code understanding, as reported in the literature, has tended to focus on program code. The motivations for program understanding usually relate to *software maintenance*. It is necessary to understand existing code in order to modify it to meet new requirements (e.g. for extra functionality, improved performance, improved user interface, or debugging).

A common problem mentioned in the literature is that of legacy systems. A *legacy system* is an old piece of software, typically written in an archaic language such as COBOL to run on a long-since outdated platform. Legacy systems often need to be reimplemented in more modern languages, to run on more modern platforms. Also, it is not uncommon for vital knowledge to be embedded in legacy code without having been satisfactorily documented elsewhere.

The process of *reengineering* consists of a reverse engineering (code understanding) step, followed by a forward engineering (code generation) step [2]. Reengineering can be used for such tasks as reimplementing legacy systems, restructuring existing code to make it more readable, efficient or reusable, and translating code from one language to another. Such tasks may not be achievable by direct transformation of the code; rather, they may need to take higher-level knowledge of the source code (as extracted by code understanding techniques) into account.

An important class of code understanding techniques is based on *clichés* [12]. Clichés are commonly-used computational structures. There are *data-structure clichés* such as stacks, queues and hash tables, and *algorithmic clichés* such as sorting or binary search. A system that uses clichés for code understanding has a *cliché library*, which contains *overlays*. Each overlay associates an *implementation cliché* with a *specification cliché*, and encodes the correspondences between the two. Implementation clichés can comprise both fragments of concrete code and intermediate clichés (from which further, more abstract clichés may be extracted). Cliché-based code understanding algorithms work by matching actual fragments of code (and clichés already extracted) to the implementation clichés of overlays, and extracting an instance of the specification cliché of each matching overlay. In this way, a hierarchy of clichés representing the design of the code is built up.

Cliché-based code understanding algorithms can work bottom-up or top-down. Bottom-up approaches start from the code and extract clichés, and can be prone to combinatorial explosion. Top-down approaches start with hypothesised clichés and try to extract them from the code. Quilici [1] uses a hybrid approach which is predominantly bottom-up, but uses top-down recognition for clichés whose presence is implied by others. The CUS presented in this paper works in a wholly bottom-up manner, but will incorporate some top-down extraction for comparative constraint understanding in the future (see Section 6).

There is an analogy between cliché-based and case-based approaches to code understanding. Overlays correspond to cases, and clichés correspond to higher-level knowledge (e.g. HLCs) extracted from code. However, overlays do not admit of any partial matching and adaptation, they correspond to single-step inferences, and they are not learned automatically.

6 Conclusions and Future Work

This paper has presented a system for understanding constraints in EXPRESS information models. The CUS extracts HLCs from EXPRESS code, which embody an understanding of the constraints at an abstract level.

The CUS has been tested on 'toy' EXPRESS models such as Figure 4, and on 'real' EXPRESS models, namely AP212 [13] and the EDIF 4 0 0 Level 0 model [14]. Table 1 shows the number of HLCs, of each class described in Section 3, extracted from each of these models.

	Figure 4	AP212	EDIF 4 0 0 Level 0
`bound`*	2	0	0
`range`	1	0	0
`exactly_one`	1	35	249
`at_most_one`	0	10	76
`non_null`	2	415	911
`equation`	1	241	274
`adder`	0	3	8
`multiplier`	0	3	4

*. The cases for extracting bounds are currently too inflexible to match in all situations.

Table 1. Table to show the numbers of higher-level constraints of each kind extracted from three sample models (adapted from Figure 7-1 of [7]).

There are several ways that the CUS could be extended in the future. A short-term goal is to extend it to understand code in languages other than EXPRESS (as discussed in Section 1). The range of HLCs will be extended to cover more of EXPRESS as well as the other languages.

The current version of the CUS is a first step towards case-based code understanding, which is itself useful. However, to reap the full benefits of CBR, it needs to be able to learn its own cases, have cases that it can use to make inferential leaps such as extracting several HLCs at once, and it needs partial matching and adaptation beyond variable unification and frame substitution.

A major area for future research is *comparative constraint understanding* (and, more generally, *comparative code understanding*). This will entail understanding the comparative semantics of the HLCs on two (related) models, assessing to what extent they are semantically equivalent. We will investigate when one constraint *semantically implies* another, in the sense that if the first constraint is satisfied then the second constraint will be satisfied. This would have applications in, for example, transference of constraints for schema-to-schema mapping (see [7]). Comparative code understanding need not be limited to constraints in EXPRESS models, of course, but may involve comparing other kinds of code (which may or may not be written in the same language as each other).

An issue which arises when code is being understood for a specific purpose (such as *comparative code understanding*) is the *discriminate* extraction of higher-level knowledge. That is to say, some of the higher-level knowledge which the code understanding system knows how to extract might not be useful for the purpose at hand. It will be desirable, in terms of time and space efficiency, to avoid extracting unnecessary knowledge from code. This is likely to entail *meta-level reasoning* about which pieces of knowledge are worth extracting. Meta-level reasoning could also be used to ensure termination within a reasonable length of time (although the current CUS always does so by design).

Finally, the inverse of code understanding, *code generation*, is of interest to the authors. This could range from generating constraints for information models to a CBR approach to the total goal of automatic programming. A *bidirectional* CBR approach may be considered, in which the cases used for code understanding would be used the other way round, for code generation. As well as comparative code understanding, we may also consider *comparative code generation* - generating code to bridge the semantic gap between two pieces of code.

References

1. Quilici A. A memory-based approach to recognizing programming plans. *Communications of the ACM* 1994; 37(5):84-93
2. Chikofsky EJ, Cross JH II. Reverse engineering and design recovery: A taxonomy. *IEEE Software* 1990; 7(1):13-17
3. Grand M. *Java language reference*. O'Reilly & Associates, Sebastopol, 1997
4. van Vliet JC. *Software engineering: Principles and perspective*. John Wiley & Sons, Chichester, 1993

5. ISO TC184/SC4. Industrial automation systems and integration - Product data representation and exchange - Part 11: Description methods: The EXPRESS language reference manual. ISO standard, reference no. ISO 10303-11. ISO, Switzerland, 1994
6. Kasadha FS. An approach to the application of information models. MSc thesis, University of Manchester, Manchester, 1996
7. Broad AP. The application of case-based reasoning to the understanding of constraints on information models. MPhil thesis, University of Manchester, Manchester, 1999. Internet: `http://www.cs.man.ac.uk/~broada/cs/mphil/thesis/`
8. Kolodner JL. *Case-based reasoning*. Morgan Kaufmann Publishers, San Mateo, 1993
9. Woods S, Yang Q. The program understanding problem: Analysis and a heuristic approach. In: Proceedings of the 18th International Conference on Software Engineering (ICSE-96), Berlin, Germany, 1996, pp 6-15
10. Brown MG. A memory model for case retrieval by activation passing. PhD thesis, University of Manchester, Manchester, 1993. Technical report UMCS-94-2-1. Internet: `ftp://ftp.cs.man.ac.uk/pub/TR/UMCS-94-2-1.ps.Z`
11. Watson ID. *Applying case-based reasoning: Techniques for enterprise systems*. Morgan Kaufmann Publishers, San Francisco, 1997
12. Rich C, Wills LM. Recognizing a program's design: A graph-parsing approach. *IEEE Software* 1990; 7(1):82-89
13. ISO TC184/SC4. Industrial automation systems and integration - Product data representation and exchange - Part 212: Application protocol: Electrotechnical design and installation. ISO committee draft, reference no. ISO 10303-212. ISO, Switzerland, 1996. Internet: `http://www.nist.gov/sc4/step/parts/part212/current/part212.exp`
14. Lau RYW. EDIF: Electronic Design Interchange Format version 4 0 0 level 0 information model. Electronic Industries Association, EDIF Steering Committee, 1996