Chapter 2:
Code Understanding and Generation

Since the aim of this thesis is to understand the constraints on information models, this chapter reviews the more general topic of code understanding, the closely related topic of code generation, and the interplay between the two.

Code generation, which involves *forward engineering* code from knowledge of its intended semantics, is discussed in Section 2.1.

Section 2.2 follows with code understanding, which involves *reverse engineering* knowledge of the intended semantics from the code itself. The section considers some abstract representations of code, and methods for extracting information from source code.

Having looked at code understanding in its own right, Sections 2.3 and 2.4 consider the roles it can play in code generation:

- Section 2.3 reviews *reengineering*, in which a reverse engineering step is used to understand existing code, followed by a forward engineering step to realise changes to the code or to transform it from one form to another;
- Section 2.4 reviews *software reuse*, in which existing components are used in new programs rather than writing them completely from scratch.

There is inevitably a lot of overlap between the four sections of this chapter, so the four topics do not fit neatly into their designated sections, but rather straddle each other's boundaries.

2.1 Code Generation (Forward Engineering)

Code generation refers to the automatic writing of code in a formal language. From the point of view of this thesis, the term *code generation* usually implies *forward engineering* code from a higher-level representation. Code generation could also mean transforming existing code or converting it into another form at the same level of abstraction. However, Section 2.3 will argue that it is often necessary to understand the code at a higher level in order to be able to carry out such transformations or conversions, so it is reasonable to assume that code generation is a forward engineering process from a higher level of abstraction to a lower level of abstraction.

Code generation is the inverse of code understanding, which will be discussed in the next section. The latter is of greater relevance to understanding constraints on information models, but code generation is relevant to schema-to-schema mapping (see Section 1.3), which is the underlying motivation for the constraint-understanding method presented in this thesis. As mentioned in Section 1.3.2, code generation could be used for transferring constraints from the target model to the source model, and for generating the mapping program.

Much of the research on code generation goes under a variety of names, such as *automatic programming* (Balzer [3]), *program synthesis* (Tyugu [78]) and *software synthesis* (Setliff, Kant and Cain [67]). However, these names all imply the generation of executable code, or *programs*. In this thesis, the term code generation is used in a wider sense which encompasses code written in *any* formal language (in particular, information modelling languages such as EXPRESS [34]).

The idea of automated code generation can be traced back to the upsurge of computers in the 1950s, when high-level languages were introduced. Even conventional compilers were originally considered to be *automatic programming* systems (Balzer [3]). They converted high-level programs to machine code, delegating very low-level concerns (such as register allocation) to the compiler (Waters [82]). This certainly agrees with the definition of code generation given at the start of this section, but it is not considered as automatic programming today.

Waters [82] suggests that automatic programming could have a dramatic impact comparable to that which high-level languages and conventional compilers made in the 1950s. Middle-level concerns (such as the choice of data structures and algorithms for a problem) would be delegated to artificially intelligent automatic programming systems, which would accept specifications that are at a higher level of abstraction than conventional high-level programs. Programming would thus entail writing very-high-level specifications of problems to be solved, from which executable programs could automatically be generated.

Automatic programming would help to alleviate the so-called *software crisis*:[1] software is often delivered late, does not behave as expected, is not adaptable to changed circumstances, and still has errors when it is released (Chapter 1 of van Vliet [79]). Automatic programming would dramatically increase programmer productivity and would eliminate human error in the later stages of the software life cycle.

However, the dream of automatic programming in its full generality is still not achievable in the foreseeable future, due to the limitations of contemporary artificial intelligence techniques. Research on automatic programming appears to be stagnating after some not inconsiderable hype in the 1970s and 1980s. The mythical view of automatic programming (which was dispelled by Rich and Waters [61]) was of a system that would take as input a brief natural language description of the requirements for a program, and produce as output a program satisfying these requirements, with no intermediate human intervention. It was wrongly assumed that the automatic programming system would be able to undertake this ambitious task (which is way beyond the current state of the art in intelligent computing) in any domain, using only knowledge of programming and of the specific problem to be solved.

The truth is, of course, that an automatic programming system would need a vast base of domain knowledge to be flexible enough to write a program to solve any problem in that

1. The software crisis has existed since the 1960s, so it is not so much a crisis as an affliction.

domain. A general-purpose automatic programming system that could solve problems in any domain would need copious knowledge of all domains, which is clearly too tall an order for the state of the art in knowledge-based systems. Not surprisingly, it has been argued that automatic programming must be domain-specific (Barstow [4]).

Parser generators such as lex and yacc (Levine, Mason and Brown [41]), and the Java Compiler Compiler (JavaCC)[2] [74], are simple examples of code generation. They generate programs to parse languages, based on declarative descriptions (in the form of grammars) of those languages at a higher level of abstraction than in conventional programming languages. Such tools are certainly code generators at a higher level than conventional compilers, but are hardly worthy of being considered as automatic programming, because they are restricted to a narrow task and domain, and the limited knowledge that they use is hardwired into them.

Application generators (Section 16.1.6 of van Vliet [79]) generate programs by instantiating templates, filling in the gaps. These templates are large, and are at a very high level of abstraction: namely the level of the application domain rather than the level of a conventional program. The input to an application generator is written in a so-called *fourth-generation* (very-high-level) language. A program written in a fourth-generation language looks much like one written in a so-called *third-generation* (high-level) language, but features very-high-level programming constructs that are specific to the application domain.

The Programmer's Apprentice project (Waters [82]) aims to provide a system that complements the activity of a human programmer. The human performs the creative aspects of programming (such as designing algorithms to meet the specifications, writing code to meet those specifications and selecting data structures to represent abstract concepts), while the computer takes care of the more mundane parts (such as verifying that the design meets its specifications, verifying that the code is consistent with the design, and fixing the code

2. "Java Compiler Compiler" and "JavaCC" are trademarks of Sun Microsystems, Inc.

when the programmer changes the design or the data representation).

For example, Waters [82] presents the example of generating a 58-line Ada program from the six commands shown in Figure 2-1.

```
1. Define a simple_report program UNIT_REPAIR_REPORT.
2. Fill the enumerator with a chain_enumeration of UNITS and REPAIRS.
3. Fill the main_file_key with a query_user_for_key of UNITS.
4. Share the first READ and the second READ.
5. Fill the title with ("Report of Repairs on Unit" & UNIT_KEY).
6. Remove the summary.
```

Figure 2-1. A set of program generation commands (Waters [82]).

Note that the first command is an overall statement of requirements, followed by a list of more specific requirements or successive ways to refine the program. This is typical of the way informal program specifications are usually given.

In the first command in Figure 2-1, `simple_report` is the name of a *cliché*: a code template with some blanks to be filled in (as specified in subsequent commands). There are two other clichés in the figure: `chain_enumeration` and `query_user_for_key`. Section 2.2.2 will have more to say about clichés in a code understanding context.

The Programmer's Apprentice thus operates by instantiating code templates, to which it can make simple changes. This is a step closer to true automatic programming, but is inflexible in that the templates are hardwired in. The system is capable of producing variants of stock programs, but not of creating genuinely new programs.

Nevertheless, the ideas in the Programmer's Apprentice are interesting: in particular, the use of clichés has much affinity with case-based reasoning (see Section 1.4 or Chapter 3). A true case-based approach to the Programmer's Apprentice would be more flexible, allowing cases that match the program specification to be retrieved dynamically rather than simply instantiating clichés by name.

This section has discussed code generation. Its inverse process, code understanding, will be covered in the next section.

2.2 Code Understanding (Reverse Engineering)

Code understanding is the opposite of code generation: the term *code understanding* implies *reverse engineering* a higher-level representation of code written in a formal language.

Code understanding often means extracting knowledge from executable code written in conventional high-level languages (or even machine code). The topic of code understanding is being studied under the banners of *program understanding* (Biggerstaff, Mitbander and Webster [6]), *program comprehension* (von Mayrhauser and Vans [80]), *program analysis* (Harandi and Ning [31]) and *reverse engineering* (Waters and Chikofsky [83]). However, code understanding can be applied to code written in *any* formal language - not just executable program code, but also declarative code such as information models written in EXPRESS, which is what this thesis relates to. Relational database schemata can be reverse engineered into entity-relationship models or object models (Premerlani and Blaha [55]). Reverse engineering can even be applied to hardware; for example, understanding logic electronic design specifications (Mir [47]). It is actually from hardware that the term 'reverse engineering' originates, according to Chikofsky and Cross [15].

Reverse engineering is a term used by different people to mean different things. Section 18.4 of van Vliet [79] defines it as "the process of reconstructing a lost blueprint" and considers it to be applicable to extracting requirements from design as well as design from code. Filer [26] defines it as "the process of reconstructing knowledge of an object from all available information sources". Chikofsky and Cross [15] define it as "the process of analysing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction".

Reverse engineering does not, in itself, involve any modification (Chikofsky and Cross [15]), but the motivation for reverse engineering is usually that understanding code is prerequisite to making changes or reimplementing it in another form. For example, one of the underlying motivations for understanding the constraints on information models in the way presented in this thesis concerns schema-to-schema mapping. As explained in Section 1.3.2, constraints on the target model that do not hold on the source model should be transferred to the source model so that the generated mapping program can check those constraints on source instances prior to a mapping.

Section 2.2.1 introduces some models for representing code, which will be referred to in later sections that discuss reverse engineering and reengineering. Section 2.2.2 discusses the recognition of clichés in programs, and Section 2.2.3 discusses approaches to understanding the architectures of large software systems using views.

2.2.1 Some Models for Representing Code

In the reverse engineering community, various models are used to represent code, each capturing some useful aspect of the code such as its structure, its functionality, its flow of control and/or data, etc. Reverse engineering tools extract information from source code according to these models. This subsection identifies some of the common models that are being used:

- *Clichés*: commonly-used programming structures that implement higher-level abstractions (Section 2.2.2);
- *Plans*: representations of clichés in a language-independent formalism (Section 2.2.2);
- *Control-flow graphs*: graphs in which a node represents a program statement and a link represents that one statement could be executed after another (Cimitile, De Lucia and

Munro [17]);

- *Data-flow graphs*: graphs in which a node represents a program statement and a link represents that one statement needs the output from another. In fact, data-flow links are often superimposed on control-flow graphs, as in the *program dependence graph* of Ferrante, Ottenstein and Warren [24], which represents the necessary control and data dependencies in a program in order to support data-flow analysis for optimising compilers. Plans in the Programmer's Apprentice also represent both the data flow and the control flow in a program, the former being the dominant concept and the latter only used where necessary (Waters [82]);
- *Call graphs*: graphs in which a node represents a function and a link represents that one function calls another (Cimitile, De Lucia and Munro [17]);
- *Subsystem hierarchies*: graphs representing the way software systems are divided into subsystems (Müller, Orgun et al [48]);
- *Slices*: functionally related subsets of statements in a program, grouped together (Section 2.3.1);
- *Views*: snapshots of reverse engineering states representing abstractly the high-level functionality of systems, which can highlight various different aspects of the code such as those listed above (Section 2.2.3).

The application of these code models will be revealed in later subsections and later sections. To begin with, the recognition of clichés in programs is discussed.

2.2.2 Recognition of Clichés

A *cliché* is a commonly-used programming structure that typically implements a higher-level abstraction (Rich and Wills [63]). For example, *sequential search* (Waters [82]) is a cliché for searching a one-dimensional structure for an element that satisfies some criterion. Sequential search works by iterating over the elements in the structure, testing each element to determine whether it satisfies the search criterion. If so, the search terminates and returns that element. The search fails if it gets to the end of the structure without finding a matching element. This is the knowledge that the sequential search cliché encodes.

The idea of *cliché recognition* is to recover the design of a program (or any piece of code, for that matter) by recognising the clichés that are used in the code. This section concentrates on the role of clichés in code understanding, but some of the ideas also apply to the use of clichés for code generation (as already discussed in Section 2.1).

Rich and Wills [63] identify two broad classes of cliché:

- *algorithmic clichés* such as sequential search, binary search and successive approximation;
- *data-structure clichés* such as sorted lists, hash tables and balanced binary trees.

Both varieties are worth recognising in programs.

A *plan* is a representation of a cliché in a language-independent formalism called the *plan calculus* (Rich and Wills [63]). Both clichés and individual programs are represented in the plan calculus. A plan is essentially a data-flow graph which can be augmented with additional information such as control flow, preconditions and postconditions.

Clichés are stored in a *cliché library* (Rich and Wills [63]). Each entry in the cliché library is an *overlay* which associates an *implementation cliché* (a plan that can be recognised from the program) with a *specification cliché* (an abstract characterisation of the implementation cliché).

Figure 2-2 (a slightly edited adaptation of the figure on page 13 of Rich and Waters [62]) shows an example of an overlay, written in a graphical notation for the plan calculus. A list's push operation (the overlay's specification cliché) is implemented by incrementing the index of an indexed sequence and then updating the sequence to store the new term at the position denoted by the index (the overlay's implementation cliché).

The name of the overlay in Figure 2-2 is `bump-and-update-as-push`. The implementation cliché consists of a plan called `bump-and-update`. Whenever this plan is recognised in a program, the plan represented by the specification cliché will be extracted.

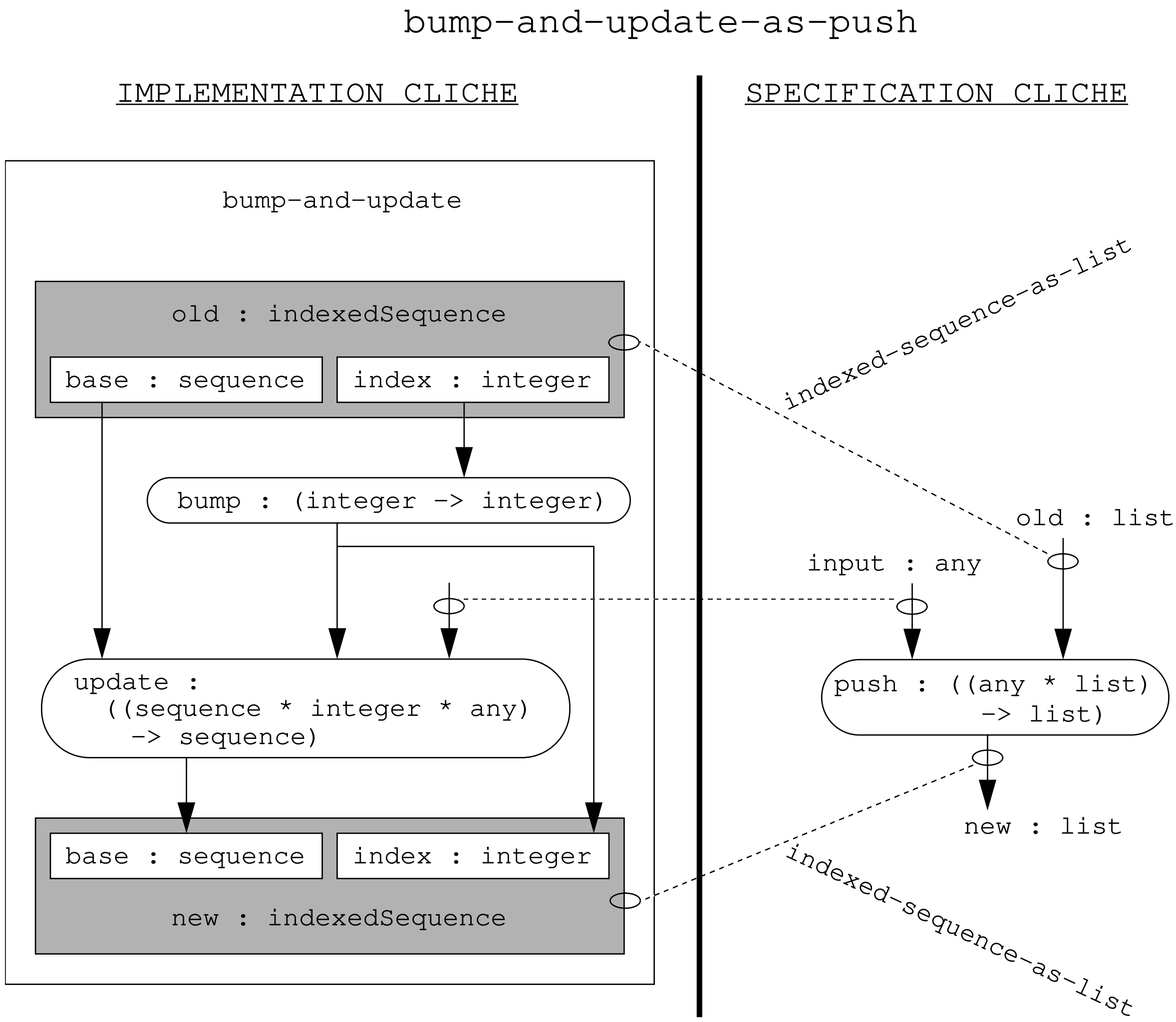

Figure 2-2. A sample overlay (adapted from Rich and Waters [62]).

Figure 2-2 adopts the following notation (slightly adapted from that used in Rich and Waters [62]):

- Rectangles represent data values;
- Rounded rectangles represent functions;

- Arrows represent data flow;
- The dashed lines with the tiny ovals at the ends denote *correspondences* between the implementation cliché and the specification cliché. Correspondences may be tagged with the names of other overlays, such as `indexed-sequence-as-list` on two of the correspondences in this example (meaning that the input and output lists are implemented as indexed sequences).

In Rich and Wills' [63] approach, the program under consideration is translated into the plan calculus, and the task of cliché recognition is to extract occurrences of clichés from the program. Fragments of the (plan representing the) program are matched to the implementation clichés of overlays. If a program fragment matches the implementation cliché of an overlay, then it is replaced with an instantiation of the specification cliché of the overlay. Such instantiations of specification clichés can themselves match the implementation clichés of other overlays and likewise be replaced with even more abstract clichés. Proceeding in this way, the recogniser builds up a tree of recognised clichés which represents a hierarchical design of the program.

Quilici [58] classifies program understanders which recognise clichés as working either top-down or bottom-up. The *bottom-up* approach, of which Rich and Wills' [63] work is an example, starts from the program and attempts to recognise clichés from the program itself and from instantiations of clichés already recognised, whereas the *top-down* approach starts with a top-level cliché and expands it into lower-level clichés, proceeding in this way until it can match clichés to the actual program.

The top-down approach is too restrictive in that it requires a hypothesis as to the top-level cliché the program is implementing (e.g. a sorting algorithm, or a stock control system), which is not helpful for understanding code when it is not known what it is supposed to accomplish. Furthermore, if it fails to recognise the top-level cliché then it has completely failed to understand the program, whereas the bottom-up approach is building up a partial understanding of the program whenever it extracts a cliché.

Unfortunately, the bottom-up approach is prone to combinatorial explosion because the smallest unit the recogniser is prepared to consider (e.g. a program statement) could potentially match any overlay in the whole library, which could itself match any overlay in the whole library, and so on. What is needed is a way of pruning the search space so that only potentially relevant overlays are considered for matching. Quilici [58] uses indexing: each plan[3] is linked to other plans that are only considered for matching under certain conditions, which eliminates irrelevant plans from consideration. For example, incrementing a variable in the body of a loop may be indexed to a plan for counting the iterations of a loop.

Quilici [58] further alleviates the combinatorial explosion problem by taking a hybrid bottom-up/top-down approach. Although his approach is primarily bottom-up, there are certain plans, called *implied plans*, whose occurrence is implied by the occurrence of regular plans. These implied plans are not recognised in a bottom-up manner, but are sought for top-down whenever a plan is recognised which is known to imply such plans. For example, a point in two-dimensional space might be represented by two variables, but it is clearly a bad idea to consider every pair of variables in the same scope as a point! However, a plan for calculating the distance between two points using Pythagoras' Theorem is distinctive enough to be recognised bottom-up; this plan is known to imply two occurrences of points, so these are recognised in a top-down manner.

This subsection has discussed the recognition of clichés (or plans) in programs. As mentioned in Section 2.1, clichés clearly have some affinity with case-based reasoning. The matching of fragments of particular programs to the implementation clichés of overlays is in line with what one might expect from a case-based approach to code understanding (c.f. Section 1.4).

3. Quilici [58] tends to refer to *plans* where Rich and Wills [63] refer to *clichés* and *overlays*.

Cliché recognition has much in common with the approach to understanding constraints on information models that will be presented in this thesis:

- Overlays are analogous to cases. In the system presented in this thesis for understanding constraints, a case associates a fragment of EXPRESS code with a higher-level constraint that captures the abstract constraint semantics implied by that fragment of code.
- Specification clichés are hence analogous to higher-level constraints.
- Implementation clichés are analogous to fragments of EXPRESS code (or higher-level constraints that have already been extracted from the model, from which further higher-level constraints may be extracted).

Although the cliché-based approach to code understanding and generation is similar in principle to case-based reasoning, it is not referred to as case-based reasoning in the literature. Indeed, Rich and Wills [63] consider it simply as graph parsing, while Quilici [58] goes further and calls it a memory-based approach. The main differences between these approaches and case-based reasoning are that:

- Case-based reasoning can match the ‘problem’ parts of cases *partially* to new situations and *adapt* solutions to compensate (Section 12.1.1 of Kolodner [39]). Other approaches use exact matching and no adaptation, and thus lack the flexibility that case-based reasoning offers.
- Case-based reasoning can also encode larger chunks of knowledge as cases than other techniques (Section 1.3 of Kolodner [39]). This allows case-based reasoning to make inferential jumps where other methods would take a series of small reasoning steps, requiring more search. In a case-based approach to cliché recognition, for example, several specification clichés could be composed into a single case and extracted together, rather than having to extract them individually, using an overlay for each.

The fact that overlays can be used for both code generation and code understanding (Rich and Waters [62]) suggests that cases should also be useful for both. This could lead to a bidirectional form of case-based reasoning in which not only could the ‘problem’ parts of cases be matched to a new problem situation in order to suggest a solution (e.g. to generate code), but the ‘solution’ parts of cases could be matched to situations in which the inverse problem (e.g. code understanding) is to be solved.

The next subsection will discuss code understanding on another level, namely the comprehension of large software systems.

2.2.3 Understanding the Architecture of Large Software Systems

For large software systems, it is more important to understand the overall *architecture* of a system than the details of its individual algorithmic components (Wong, Tilley et al [87]). The Rigi system (Müller, Orgun et al [48]) aims at understanding the architecture of software systems from the source code. It creates abstract representations of a system’s higher-level functionality, called *views*.

However, these views are generated for the benefit of humans trying to comprehend a system in order to maintain it; it is unclear how useful the views would be if the maintenance itself were computerised. Wong, Tilley et al [87] admit that even the extraction of the views is only partially automated because “human cognitive abilities are still more powerful than hardwired algorithms”. However, they do not seem to have considered what could be achieved using artificial intelligence techniques, which rely on explicit knowledge rather than hardwired algorithms. Case-based reasoning in particular uses experience rather than hardwired knowledge, and is much more flexible and powerful than hardwired algorithms.

A *view* is a snapshot of a reverse engineering state (Wong, Tilley et al [87]) that highlights different aspects of the software system under investigation (Tilley, Müller and Orgun [77]). This appears to mean that reverse engineering states are saved as views so that they can be used later.

A view contains directed *graphs* which represent the program in various ways (Wong, Tilley et al [87]). For example, a *resource-flow graph* represents structural dependencies of interest, such as function calls and data accesses. Wong, Tilley et al [87] generated a view of the entire *call graph* for a system, but found that the developers of that system did not recognise the abstractions thus generated. They had more success by decomposing the system with regard to the names of modules in the system (which followed a certain naming convention), and by generating a *subsystem hierarchy* based on top-down flow of control.

Figure 2-3 gives an example of a view (adapted from Figure 1 in Tilley, Müller and Orgun [77]). It depicts the high-level architecture of a C program for processing tables, in the form of a subsystem hierarchy. Rigi could expand the subsystems further, producing detailed views of them if required.

Müller, Orgun et al [48] present a method for subsystem composition (extracting a subsystem hierarchy from source code):

- Firstly, system components and the dependencies between them are automatically extracted from the source code to form flat resource-flow graphs (in which the nodes represent components and the links represent dependencies);
- Subsystem hierarchies are then composed semi-automatically on top of these resource-flow graphs, and the interfaces between these subsystems are computed;
- The subsystem structures can also be evaluated using software engineering metrics such as coupling and cohesion;
- Finally, snapshots of the reverse engineering environment can be saved as views.

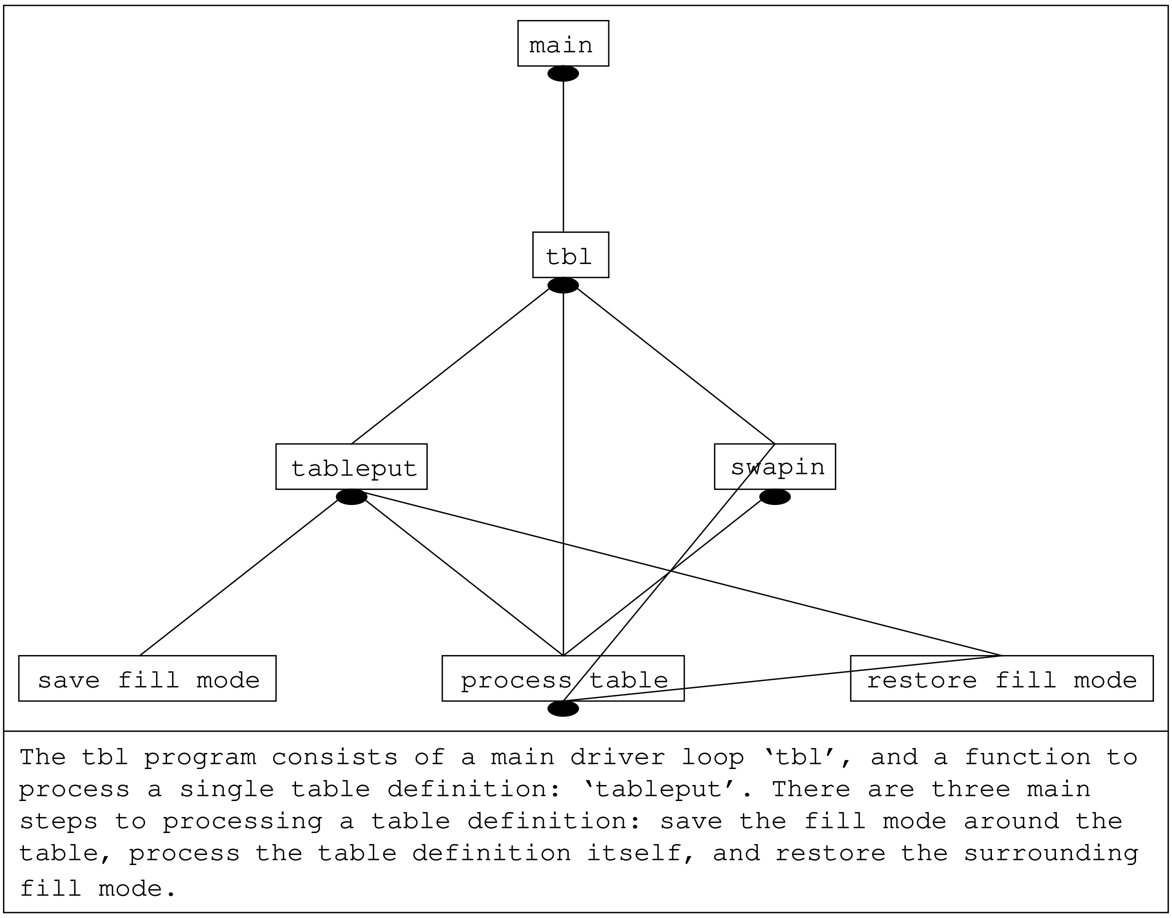

Figure 2-3. A high-level view of a program's architecture (taken from Figure 1 in Tilley, Müller and Orgun [77]).

In summary, the work referred to in this subsection concerns understanding the architecture of large software systems by extracting views from the source code (for example, a system may be broken down into subsystem hierarchies, making it easier to comprehend). Views serve to capture structural rather than functional abstractions (Müller, Orgun et al [48]), suppressing algorithmic details. The wider implication that can be drawn from this is that a system that reasons about code, whatever its degree of automation, may need to understand the code on a continuum of different levels, from the low-level details (such as how algorithms are implemented) to the overall structure of the code.

The fundamental limitation of view-based code understanding is not so much in the views themselves, but in the lack of any attempt at automated systems which *use* the views that can be extracted from code. The purpose of views, according to the literature cited in this subsection, is to aid human comprehension of software so that humans can maintain the software. Only automation of the reverse engineering side of the task of maintenance has been attempted. The way forward would be to investigate how views could be used by forward engineering processes which automatically maintain the software. Automatic software maintenance would thus consist of a combination of reverse engineering and forward engineering (this is known as *reengineering*: see Section 2.3).

The extraction of views referred to in this subsection suffers from the weakness of being only semi-automated, in that the Rigi system can extract views automatically but cannot decide for itself which views to extract. This is because only humans understand the goal (in terms of the maintenance that is to be done) that would be achieved by extracting a particular view. However, if the maintenance itself were automated then a computer *would* understand the maintenance goals, and could use that knowledge to guide the extraction of views.

Views are not so relevant to the task of understanding constraints on information models, because this entails a very localised understanding of the EXPRESS code (looking at entities individually to see what the constraints are). However, the idea of views could be very relevant to the wider topic of understanding information models (e.g. views of the inheritance hierarchies in models, or of containment relationships between entities). Realistic information models are typically very large (thousands of lines of code), so views could be an invaluable aid to overall comprehension of those models by humans, let alone computers!

That concludes Section 2.2, which has focused on reverse engineering (code understanding). The next section considers how reverse engineering can combine with forward engineering to modify existing code, a process known as *reengineering*.

2.3 Reengineering

Reengineering is the process of modifying existing code, and consists of a reverse engineering step (to understand the old code) followed by a forward engineering step (to implement the required changes) (Chikofsky and Cross [15]). This could mean extracting a higher-level description of the code, transforming it at that level and then reimplementing it. Alternatively, it could mean modifying the code directly, but with an understanding of it sufficient to know how to modify it. Figure 2-4 illustrates diagrammatically the notion of reengineering.

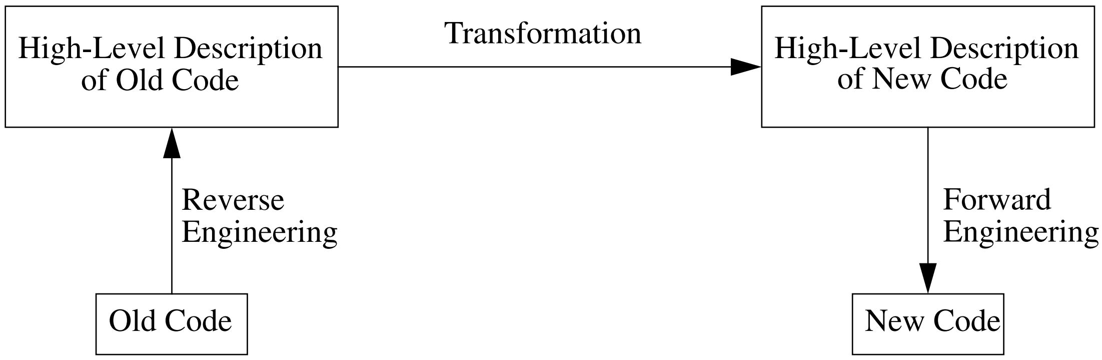

Figure 2-4. Reengineering = reverse engineering + forward engineering.

There are various motives for reengineering, many of which can be traced to the need for software maintenance. It is very common for the requirements of software systems to change, and these changes must propagate through the specification and design of the system, and ultimately to the implementation. It is necessary to understand the implementation (program code) in terms of how the various parts of it are linked to the corresponding parts of the design, specification and requirements.

Another reason for reengineering is to deal with *legacy systems*: programs written decades ago, which can often be large, unstructured, written in archaic programming languages and written to run on long since outdated platforms (Ning, Engberts and Kozaczynski [51]). It is undesirable simply to throw them away and reimplement them from scratch, due to the not inconsiderable programming effort required, and also because important domain knowledge may have been embedded in the code without having been satisfactorily documented elsewhere. So researchers have been looking to reverse engineer the legacy code to extract knowledge and understand it well enough to forward engineer a reimplementation (preferably automatically).

The problem that legacy code is unstructured can be attacked with program transformation techniques such as eliminating `GOTO` statements. Böhm and Jacopini [7] show that it is possible to normalise any flow diagram (representing a program) into a semantically equivalent flow diagram which is decomposable into a finite number of occurrences of three base diagrams corresponding to:

- concatenation of statements;
- conditional statements;
- loop statements;

There is no need for arbitrary links (`GOTO` statements). Programs can also be transformed more radically, for example by restructuring them so that the modules in the program correspond one-to-one to meaningful functional abstractions - this can be achieved by program segmentation, as discussed in Section 2.3.1.

The problem that legacy code is written in an archaic language can be solved by translating it into a more appropriate language for the time. For example, Andrews, Del Vigna and Molloy [2] discuss a translator from pTAL (an archaic block-structured programming language) to C++ (a modern object-oriented programming language). However, it would seem that the C++ code that their translator generates is largely procedural rather than object-oriented (since they admit that it is dominated by the C language subset of C++).

Programming language translation can be thought of as a reengineering activity, as some understanding of the code is needed in order to translate it (Quilici [58]). This is especially true when translating from one programming paradigm to another (e.g. converting a procedural C program into an object-oriented C++ program, as in Quilici [58]) - this requires a deeper understanding of the code.

Reengineering can also assist the following tasks:

- Debugging - it is necessary to understand code in order to locate the causes of bugs, and fixing them is a forward engineering task;
- *Automatic parallelisation*: transforming sequential programs into parallel programs so as to exploit as much parallelism as possible whilst guaranteeing that the parallel program is semantically equivalent to its sequential counterpart (Chapter 2 of Sakellariou [66]).

It turns out that program segmentation is particularly useful for both these tasks (see Section 2.3.1).

The task of transferring constraints for schema-to-schema mapping can also be thought of as a reengineering problem. As mentioned in Section 1.3.2, it is necessary to understand the comparative semantics of the constraints on the two models, in order to see which of the target model's constraints need to be transferred to the source model, and to be able to transform them to apply to the source model. This understanding, which is the subject of this thesis, is the reverse engineering step. The forward engineering step is actually transferring the constraints, which remains a future research issue.

2.3.1 Program Segmentation and Slicing

Program segmentation (Ning, Engberts and Kozaczynski [51]) is about isolating functionally related pieces of code from programs. The problem, especially with large and unstructured programs, is that the functionality of a program is often distributed throughout the source code - one function (in the sense of a distinct behaviour as observed externally) might be implemented by several modules in the program, and one such module might be implementing several different functions. The purpose of program segmentation is to assist program understanding by grouping together statements in the program that collectively implement the same functionality, even though these statements might be separated from each other in the source code. The issue of what functionality to look for will be considered later in this section.

Program segmentation is useful for code understanding in its own right, as it is helpful for recognising delocalised plans in programs. Ning, Engberts and Kozaczynski [51] imply that program segmentation could help cliché recognition techniques (as discussed in Section 2.2.2) to extract clichés from program statements that are functionally related but scattered throughout the source code. As mentioned in Tan and Ling [75], program segmentation (and a technique called *slicing* in particular) is also useful for reengineering tasks such as debugging (because it helps to isolate the sources of bugs in programs) and automatic parallelisation (because it isolates independent fragments of program code that can be executed in parallel).

Slicing is an approach to program segmentation in which control flow paths are followed, in order to select a subset of the statements in a program[4] based on some criterion (as discussed in the following paragraphs). A *slice* is the set of statements thus selected, which collectively implement an external function of the program.

The commonest variety of slicing is *backward slicing* (Ning, Engberts and Kozaczynski [51]), the purpose of which is to determine which statements could affect the values of a set of variables, or which statements contribute to a particular output. It works by tracing backwards from a given statement (usually one which produces an output or returns a result) to determine which of the statements that are executed before the given statement could affect the output (i.e. affect the values of a given set of variables, e.g. by assigning to

4. Not necessarily contiguous statements.

them).

For instance, Tan and Ling [75] use the example of a C program which counts the number of lines, words and characters in a given text. They slice it with respect to the variable `nw` (number of words) across the whole program, which results in a program to count just the number of words in the input text, because the code that counts the lines and characters has been sliced out. In other words, slicing has isolated one of the three external functions of the program.

Another, less common variety of slicing is *forward slicing* (Ning, Engberts and Kozaczynski [51]), which isolates the code that depends on the values of given input variables by tracing forwards from a given statement to find those statements which are affected by the values of those variables (e.g. statements which read those variables).

Ning, Engberts and Kozaczynski [51] also define *condition-based slicing*, which, given a logical expression, extracts those statements that are executed when the expression is true (as determined by the conditions of 'if-then' statements and loops).

Ning, Engberts and Kozaczynski [51] present a system called Cobol/SRE, a System Renovation Environment, which can perform forward, backward and condition-based slicing for Cobol programs, as well as other operations such as combining slices using set union, intersection and difference operations. In their notion of program segmentation, these operations comprise the first major step, which they call *focusing*. Their second major step is called *factoring*, in which the code segments (slices) produced in the focusing step are packaged into modules that are more reusable, since the slice that each such module is based on supposedly represents a single external function of the program.

Program segmentation is therefore useful for migrating code out of legacy systems, which typically *are* large, unstructured and written in archaic languages such as Cobol. Program segmentation helps to achieve better structure by remodularising the program so that modules have a one-to-one correspondence with the external functions isolated by slicing. It could also translate the program into another (more modern) language while it's at it, since slicing operates on a program's control-flow graph rather than directly on the source code, which helps to abstract away from language-specific details (Tan and Ling [75]).

Program segmentation also promotes software reuse, because a module that implements a single external function is clearly going to be more reusable than modules in badly structured programs, which can have a many-to-many relationship with the external functions they implement. Slicing helps to build reusable components, as an example from Tan and Ling [75] illustrates, in which they had a program to compute the average mark from a database of student records. Slicing this program with respect to 'student record' yielded a program that simply iterated over the student records in the database. Although not useful as a stand-alone program, such a program is more reusable than the original because it is more widely applicable: it is useful as the basis for any program which needs to iterate over student records. What the contemporary literature on program slicing seems to ignore is the possibility of inserting code into such slices to create a useful program in a reuse situation. It is the Author's belief that keeping track of what is sliced *out* of the original program, and from *where* it is sliced out, may reveal insights into the inverse process of inserting code into slices when they are reused. Software reuse will be considered in its own right in Section 2.4.

When slicing in order to isolate a functional abstraction, knowledge of the functional abstraction that is being sought is very important, but the work discussed so far in this subsection does not make a systematic attempt to capture this knowledge - rather, it is left to the human user to provide as input a slicing criterion (i.e. a set of variables to slice with respect to, and where in the code to start and stop slicing). Choosing a slicing criterion requires having some hypothesis as to the functional abstraction to look for, and where in the code to look for it.

In *specification-driven program slicing* (Cimitile, De Lucia and Munro [18]), slicing criteria can be derived automatically from a specification of the functional abstraction that is being sought (its signature, precondition and postcondition). This is done by performing *symbolic execution* on the program, which is like ordinary execution except that the values of variables in each state of execution are symbolic expressions rather than constants, and execution may have to bifurcate (and merge together again) if it encounters a conditional branch. Symbolic execution associates each statement in the program with a *symbolic state*, which is a set of bindings of variables to their symbolic values, together with a path condition (a logical expression of the assumptions of state before the current statement is executed). The specification-driven program slicing process determines for each statement whether the precondition or the postcondition of the functional abstraction being sought holds in the symbolic state of that statement - if so, then that statement is a candidate to be the first or the last statement respectively for the slicing criterion. Having identified a suitable slicing criterion, the specification-driven program slicing process can then use it to slice the program and thus isolate the said functional abstraction.

This subsection has discussed program segmentation, and program slicing in particular. As with views (see Section 2.2.3), slicing has no intrinsic limitation. Rather, the issue is how slicing could be used as a building block for a reuse reengineering system. *Reuse reengineering* is the process of reengineering code to make it reusable, and will be discussed further in Section 2.4. By isolating the functional abstractions in programs, slicing can be used to produce reusable components.

Slicing has limited relevance to understanding the constraints on information models, because it operates on imperative code, and most code in information models is declarative. However, imperative code *does* feature in EXPRESS: it can appear in functions, procedures and global rules, the purpose of which is to express constraints. Techniques for understanding imperative code would thus be needed for a complete system for understanding the constraints in EXPRESS code (the system presented in this thesis is currently restricted to the declarative aspects of EXPRESS, though it could be extended to deal with imperative code in the future). Slicing would come into its own for imperative code fragments that check several higher-level constraints (see Section 1.2.3) at once, or if a particular higher-level constraint is checked by several delocalised imperative code fragments. Slicing would be helpful for top-down extraction of higher-level constraints in imperative code (a higher-level constraint which the system is trying to recognise could form the basis for a slicing criterion).

This section has discussed reengineering, which is one way in which forward and reverse engineering interact. They also interact in software reuse, which is the topic of the next section.

2.4 Software Reuse

Software reuse (Mili, Mili and Mili [46]) is another area where forward engineering and reverse engineering come together. Software can be constructed by reusing pieces of old software rather than writing new software from scratch, an approach to software development which clearly has much affinity with case-based reasoning (see Section 1.4 or Chapter 3). The major benefit of reuse is that it increases productivity considerably and thereby alleviates the software crisis (c.f. Section 2.1).

The relevance of reverse engineering to reuse is that to reuse something, it is clearly necessary to understand it. A human or automatic programmer that reuses a software artifact needs at least to understand the *functionality* of that artifact (i.e. *what* it does), in order to judge its applicability to the situation at hand (i.e. to the software system being developed).

The simplest kind of reuse is simply to plug into the new software system a module[5] from an existing system, without modification, and to call it normally. Unfortunately, a lot of modules that are *almost* reusable in this manner are not suitable to be simply plugged in without modification (e.g. because their functionality is not quite the same as required for the new system, or they rely on their old environment in such a way that they will not work in their new environment). Due to such problems, programmers are often inclined not to reuse such modules at all, and instead to ‘reinvent the wheel’. This is one of the reasons why software reuse has not caught on very well in practice, despite the increased productivity it promises.

To exploit such nearly-reusable modules so as to reap the maximum benefit of reuse, it is clearly necessary to modify them so that they implement exactly the required functionality and work in the target environment. However, human programmers are often reluctant to modify existing code, especially code that they themselves did not write in the first place. To modify existing code requires understanding it first - not just *what* it does but *how* it does it, and the links between the ‘what’ and the ‘how’. This is where reverse engineering technology (as discussed in Section 2.2) becomes very useful, whether it is to aid human code comprehension so that modifications can be carried out by hand, or to enable machine comprehension of the code so that modifications can be carried out automatically. Modifying existing software components for reuse in a new system can be thought of as a reengineering activity, so the techniques discussed in Section 2.3 apply here too.

The above discussion suggests that software reuse is a process of *retrieving* a reusable component that is relevant to the requirements of the new software system and then *adapting* it to fit. *Retrieval* and *adaptation* are key terms in case-based reasoning (see Chapter 3), and indeed, they are exactly the words that Mili, Mili and Mili [46] use to describe the process of building with reusable software.

---

5. *Module* is used here as a generic term to refer to natural units of reuse such as subroutines, functions and procedures, or classes in an object-oriented language, or packages that group these modules together.

The need for adaptation can be obviated by writing components which are more reusable in the first place. Generic code is more reusable than specific code (e.g. a polymorphic sorting routine is more reusable than one which only sorts integers). Modern programming languages have features which promote writing more reusable code. For example C++'s templates (Chapter 8 of Stroustrup [72]), SML's functors (Chapter 7 of Paulson [53]) and many features of object-oriented programming such as encapsulation and inheritance (Chapter 2 of Eliëns [20]) promote writing reusable code. These features encourage programmers to structure their code into self-contained and hence more reusable modules.

Reuse reengineering (Cimitile, De Lucia and Munro [17]) is the enterprise of reengineering existing code to make it more reusable. Rather than writing reusable components from scratch, they can be reengineered from legacy code. As already discussed in Section 2.3.1, program segmentation identifies functional abstractions and packages them as reusable modules.

There are other techniques for reuse reengineering, such as identifying objects in procedural code (Canfora, Cimitile and Munro [12]) so that the code can be rewritten in an object-oriented style. The main advantage of such reengineering is that it achieves *data abstraction*. An object, which encapsulates data and provides operations as the sole mechanism for manipulating that data, is more reusable than the corresponding procedural code, where all procedures that access the data manipulate it directly.

Section 16.1 of van Vliet [79] subdivides reuse technologies into two categories: composition-based and generation-based:

- In a *composition-based* technology, existing components are composed to form the new system. There is no adaptation of the components that are being reused. In the extreme, this could mean simply using a library of subroutines.
- *Generation-based* technologies, however, are more akin to automatic programming (van Vliet [79] considers application generators a prime example). The knowledge that is

being reused is often embedded in the program generator (as in lex and yacc (Levine, Mason and Brown [41])) rather than in components.

There is, in fact, a continuum between composition-based and generation-based technologies, with many reuse systems falling between the two extremes, just as case-based reasoning falls between the extremes of solving a problem from first principles and solving a problem by recycling the solution to exactly the same problem as it was solved in the past.

This chapter has given an overview of the general field of code generation (forward engineering) and code understanding (reverse engineering), including two areas in which they interact (reengineering and reuse).

This thesis is about understanding the constraints on information models, which is a special case of code understanding. The underlying motivation is to understand the *comparative* semantics of the constraints on two models, which would be needed for such tasks as schema-to-schema mapping (see Section 1.3). In order to be able to generate a mapping program (a case of forward engineering) which checks *before* a mapping that the records in the source repository will not violate the constraints on the target model, Section 1.3.2 proposed transferring constraints from the target model to the source model. This transference of constraints is a reengineering process, as it requires first understanding the constraints (the reverse engineering step) and then transferring constraints from the target model to the source model (the forward engineering step).

The next chapter discusses case-based reasoning, the problem-solving paradigm which is to be applied to the understanding of constraints on information models. Case-based reasoning is based on the same philosophy as software reuse: problem solving is regarded as a process of adapting the solutions to similar old problems, rather than solving new problems from scratch. Chapter 3 will give an overview of the field of case-based reasoning, concluding with a justification of its applicability to code understanding and generation, and to understanding the constraints on information models.