From Specification to Implementation in Logic

by

Paul Anthony Strooper

B.Math., University of Waterloo, Canada, 1986
M.Math., University of Waterloo, Canada, 1988

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor in Philosophy
in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

Dr. M.H. van Emden, Supervisor (Department of Computer Science)

Dr. M.H.M. Cheng, Departmental Member (Department of Computer Science)

Dr. D.M. Hoffman, Departmental Member (Department of Computer Science)

Dr. K.F. Li, Outside Member (Department of Electrical and Computer Engineering)

Dr. E.W. Elcock, External Examiner (Department of Computer Science, University of Western Ontario)

Supervisor: Dr. M.H. van Emden

Abstract

The use of modules to decompose large software systems into smaller, more manageable, programming tasks is now widely accepted. To benefit from such a decomposition, the requirements of each module have to be defined in a module interface specification. The modules must then be implemented according to their specifications. We propose a method for deriving Prolog implementations of modules by transforming interface specifications written in logic.

We present the SCS method for writing module interface specifications in logic. With this method we obtain precision in both syntax and semantics. By using logic, we satisfy the standard criterion for semantic precision: statements about a module's externally observable behaviour are logical consequences of the specification, when regarded as a theory of first-order logic. The specifications can be written so that they are executable as Prolog programs.

Although the specifications are executable, they are often inefficient. Logic-based program transformation allows us to make a logic program more efficient while preserving its meaning. We present a method for specializing a program for a particular query using a "complete set of frontiers." This idea is incorporated into FROST, an interactive transformation system that assists the user with the construction of complete sets of frontiers.

Finally, we show how we can transform an SCS specification so that it can serve as an implementation. Although part of this transformation can be applied systematically, we cannot derive the entire implementation for even the simplest modules.

We discuss the problems that occur and look at alternatives for obtaining confidence in the correctness of an implementation.

Examiners:

Dr. M.H. van Emden, Supervisor (Department of Computer Science)

Dr. M.H.M. Cheng, Departmental Member (Department of Computer Science)

Dr. D.M. Hoffman, Departmental Member (Department of Computer Science)

Dr. K.F. Li, Outside Member (Department of Electrical and Computer Engineering)

Dr. E.W. Elcock, External Examiner (Department of Computer Science, University of Western Ontario)

Acknowledgements

I thank my supervisor, Dr. Maarten van Emden, who has been a great source of inspiration throughout my graduate career. He introduced me to the field of logic programming, and was the source for many important ideas in this thesis. His valuable comments have greatly improved my presentation and writing skills.

I owe much to my personal Software Engineering coach, Dr. Dan Hoffman. He was always available to discuss any problems I encountered, Software Engineering related or not. Dr. Mantis Cheng was instrumental in developing the transformation system discussed in this dissertation. He created my interest in the area of program transformation and provided me with many ideas in this area. I would like to thank my committee members, Dr. van Emden, Dr. Cheng, Dr. Hoffman, Dr. Li, and Dr. Elcock, for all their suggestions on earlier versions of the dissertation.

I also thank my fellow students Rajiv Bagai, Rod Byrne, Jimmy Lee, Mehmet Orgun, David Rosenblueth, Randal Tomczuk, Peter Walsh, Mike Whitney, and Bill Woolley, for providing me with the much needed relief on our numerous coffee breaks and drinking sessions. Will Kastelic was always there to solve any technical problems I encountered, Jocelyn Swallow allowed me to overcome the various administrative obstacles faced by a graduate student, and my brother Rudi helped me in meeting an important deadline in the preparation of this dissertation.

Finally, I would like to thank the Natural Science and Engineering Research Council and the British Columbia Advanced Systems Institute for the financial support they provided.

To my parents

Contents

List of Figures

Chapter 1

Introduction

The primary goal of software engineering is to produce software that is both correct and efficient. The first technique applied to tackle complex programming problems is to *divide and conquer* [12]: a complex system is divided up into smaller programming tasks, called modules. To allow different people to work on different modules, one has to precisely define what each module can do in a module interface specification. The module then has to be implemented according to its specification.

As Dijkstra points out [13], the confidence level of the individual components (e.g., modules) needs to be exceptionally high if we want to have a high level of confidence in the program as a whole. We present a method for obtaining efficient Prolog implementations of modules that behave according to their specifications. We first show how to write specifications in logic that are syntactically and semantically precise and that can be executed as Prolog programs. We then transform these specifications so that they become efficient enough to serve as implementations.

1.1 The software development phases

We assume that a large software system is developed using phases such as described by Parnas [47]. The phases that are typically present are[1]:

System Requirements. During this phase we record the required behaviour of the system in a *requirements specification* document. A system satisfying all the criteria in the requirements specification should be acceptable to the user.

Module decomposition. Typically, a system is too large to be produced by a single programmer. Hence, we divide the system into modules and record this decomposition and the task of each module in a *module guide*.

Module interface specification. To allow programmers to work independently on different modules, we need a *module interface specification*, which comprises the assumptions programmers can make about the behaviour of a module. It should have enough information so that the implementor of the module can implement it and so that programmers of other modules can use it.

Module implementation. After the module interface has been designed, an implementation can be written that behaves according to the specification. For more complicated modules, the implementor can first design and document the internal structure of the module [47].

[1] We have omitted some of the phases mentioned in [47] because they are of little importance here. The phases we discuss are present, in some form, in most proposals for decomposing a system into modules.

Maintenance. This step involves repeating any number of the above phases. The corresponding documents and work products should be updated accordingly.

We address the module interface specification and module implementation phases. We assume that the system has been decomposed into modules and that the module interfaces have been designed. We provide a method for recording these design decisions in a module interface specification. We also propose a method for obtaining module implementations by transforming these specifications. These transformations guarantee that the implementations behave according to the specifications. This eliminates the need for designing the internal structure of a module and for implementing the module.

1.2 From module specification to module implementation

We propose a method for writing module interface specifications based on the SCR method of Hoffman [27]. By using a formal language, the SCR method achieves precision in syntax. By applying Kowalski's axiomatization of the Situation Calculus [37] to this method we also obtain precision in semantics, which means that the relation between the text of the specification and the behaviour being specified is precisely defined. We call our method the SCS, for Situation Calculus Specification, method. The SCS method resembles the trace method of Bartussek and Parnas [2], but it has the advantage that its semantics are defined in terms of the well-established semantics of first-order predicate logic. This means that a statement about the module's externally observable behaviour is a logical consequence of the specification. An advantage the SCS method has over both the SCR and trace methods is that the specifications

can be written so that they can be executed as Prolog programs. This is useful as a form of rapid (in this case instant) prototyping.

Although it is advantageous to have an executable specification, this is not always possible, and even if it is, such a specification will typically not be efficient. A more efficient implementation is then written, often in a programming language unrelated to logic. During this process errors can be introduced and the implementation has to be checked against the specification. Logic-based program transformation provides an alternative for obtaining an efficient implementation; we transform a program simple enough to be regarded as a specification, and as such often inefficient, to a more efficient version. The transformations are applied so that the resulting program behaves in the same way as the original. Thus we obtain an efficient and correct implementation of the original specification.

We attempt to obtain efficient implementations of modules by transforming SCS specifications. We divide the transformation into two steps:

1. During *data reification* we change the representation of the state of the module from an abstract one to a more concrete one.

2. During *algorithm transformation* we transform the clear but possibly inefficient algorithms used in the specification to more efficient, but possibly less clear, ones.

There are several problems with applying these transformation steps, and even for the simplest modules we require more powerful theorem-proving techniques than the transformation steps we propose to derive an implementation from the specification. We discuss program verification and program testing as additional methods for obtaining confidence in the correctness of an implementation.

1.3 Related work

Several methods are discussed in the literature for comparing program specifications with implementations.

Program Verification. The purpose of program verification is to formally prove that an implementation behaves according to its specification [14, 19]. Complicated specifications and the difference between specification and implementation language have limited the use of program verification in practice.

Software Inspection. Fagan [17] proposes software inspection as a method for improving the quality of software. During an inspection meeting, a product is examined in order to find deviations from a set of criteria. In our case, the product is the implementation and the set of criteria its specification. This method is less formal than program verification, and it avoids some of the problems of program verification that were pointed out in [11].

Program Synthesis and Transformation. Here we transform the specification of a problem to an efficient implementation while preserving the correctness. With program synthesis we start from a specification and transform it to an implementation. When we start from an existing program and transform it to a more efficient version we refer to it as program transformation. With executable specifications or programs whose correctness is so obvious that they can be considered as specifications, the distinction vanishes.

A special case of program transformation is *partial evaluation*, where we specialize a program for a particular case by executing (partially evaluating) part of the program

so that the resulting program becomes more efficient. Komorowski [35] distinguishes between *partial deduction* and *partial evaluation* in logic programming. The latter refers to transformations that deal with various non-logical aspects of Prolog, such as the cut predicate. The former deals with the "pure" transformation rules such as discussed in [40, 60]. In this dissertation we only deal with the pure transformation rules. We use the term program transformation throughout the dissertation, although some of the transformations can also be regarded as program synthesis.

Testing. When testing an implementation, we compare the implementation to its specification for a finite set of inputs. Dijkstra [13] points out that testing can only show the presence of bugs, never their absence. Despite this obvious shortcoming, testing is the only method to obtain confidence in an implementation that is widely used in industry nowadays.

It is surprising that despite the shortcomings of the above methods very little research has been done on combining several of these methods. One such proposal is by Hoffman [25], who suggests a combination of software inspection, verification, and testing. Our method focuses on the transformational approach, but where we are unable to use program transformation alone, we discuss the possibility of using formal verification and testing to obtain confidence in the correctness of an implementation.

Our method for obtaining an efficient implementation is similar to the VDM method [33, 34]. With the VDM method, specifications are written using pre-/post-conditions over data objects representing the state of the module. These abstract data objects are then replaced by data types from the chosen implementation language during *data reification.* Then, during *operation decomposition*, these specifications are realized in a programming language. A problem with this method is that the specifi-

cation and the implementation are in different languages, and program verification is used to compare the two. By using program transformation we do not need to verify that our implementation conforms to the specification.

The VDM approach was first applied to logic programming in [38]. The term *algorithm reification* was used to describe the operation decomposition step of VDM. Using Kowalski's "Algorithm = Logic + Control" [37], this step was further decomposed into *logic reification* and *control enhancement*. We prefer to use the term *algorithm transformation*, since we transform the algorithms from highly declarative ones to more procedural ones.

Extensive research has been performed in the area of program transformation for logic programs and other declarative languages (see, for example, [49]). Most of this work differs from the transformations we perform in two significant ways:

- It is unclear what the starting point of the transformation is. A program that is much simpler than the implementation is considered as the specification. We start from a precise definition of what we consider a specification.
- The transformations are applied to single programs (relations in the case of logic programming). We are interested in obtaining implementations of modules, which typically consist of several programs accessing a common data structure. Similarly, the issue of exception handling has to be addressed if we want to obtain efficient implementations of modules.

Some of the research on *rapid prototyping* [56] is closely related to our work, despite the fact that the goals are quite different. The purpose of rapid prototyping is to obtain an implementation of a system or a module so that flaws in the specification or design can be discovered early in the development process. Efficiency of the implementation is not an issue, and to reduce development time, partial implementations

are often used. Executable specifications can be considered as a form of rapid prototyping. Some of the techniques for obtaining rapid prototypes from specifications involve transformations similar to ours (see, for example, [18]).

1.4 Organization of the dissertation

In Chapter 2 we discuss the SCS method for writing module interface specifications in logic that can be executed as Prolog programs. When these specifications are executed as Prolog programs, they are often very inefficient. In Chapter 3 we show how logic-based program transformation can be used to obtain a more efficient logic program from an obviously correct, but possibly inefficient, initial program. We present the Frontier theorem, which allows us to specialize a program for a particular query, guaranteeing correctness and completeness, and avoiding redundancy using a *complete set of frontiers*. An interactive transformation system, FROST, assists the user with the construction of complete sets of frontiers. We combine these ideas in Chapter 4, where we try to transform SCS specifications so that they can serve as implementations. We conclude by summarizing our results and indicating areas for future research in Chapter 5.

We assume the reader is familiar with the standard terminology of logic programming (see, for example, [39]). For all logic programming and Prolog code we use the syntax of ALS Prolog [64], which is an Edinburgh-style Prolog (see [7]).

Chapter 2

Module Interface Specifications in Logic

Precise module interface specifications are a powerful aid in software engineering. In this chapter we take as a starting point the SCR method of Hoffman [27]. We discuss the deficiencies of this method and propose an improvement based on Kowalski's axiomatization of the Situation Calculus [37], a method for describing state and change in first-order predicate logic [41]. Accordingly, we call our method SCS, for "Situation Calculus Specification." We introduce the SCS method with a simple example. The SCS method resembles the trace method of Bartussek and Parnas [2] in that it describes states in terms of traces. This allows us to use "trace equivalences," a concept used in the trace method, in SCS specifications. We show how SCS specifications can be written so that they can be executed as Prolog programs. This allows us to test the specification and use it as an oracle for testing purposes.

2.1 Module interface specifications

2.1.1 What are module interface specifications?

A program that becomes too large to be written by a single person in a fairly short time has to be decomposed. The parts of the decomposition are usually referred to as "modules." Although this term is often used in a loose way, we adopt the definition of D. Parnas [47], who defines a *module* as a programmer's work assignment. For the decomposition to be useful, the implementors of other modules have to rely on precisely defined assumptions about a module's behaviour. Parnas defines the *module interface* as the set of assumptions that programmers using the module are permitted to make about its behaviour [46, 47]. Finally, an *interface specification* is a statement, in whatever form, of these assumptions.

Successful use of the decomposition of a large software system into modules depends, among other things, on precise interface specifications. The precision required has two aspects: syntactic and semantic. Some modern programming languages [61, 65] provide language constructs for specifying interface syntax. However, the semantics of the interface are ignored. This means that the user of the module has to examine the implementation to determine the module's behaviour. Semantic precision means that there is a precisely defined relation between the text of the specification and what is being specified, in this case the behaviour of the module. It is the purpose of the SCS method to achieve semantic as well as syntactic precision.

2.1.2 Interface specification methods

Numerous methods to specify module interfaces have been proposed. These can be divided into three approaches.

An *operational specification* describes the required behaviour by providing a program in a programming language [4, 22]. An implementation is deemed correct if it behaves the same as the specification. A problem with these methods is that the specification focuses attention on module implementation rather than on interface design, thus violating the important principle of *separation of concerns* [14]. Although SCS specifications are executable, they do not specify by their behaviour, and as such we do not consider them operational.

The *state/predicate-based* [27] or *precondition-postcondition* [2] approach introduces explicit data structures [14, 34, 55]. The effects of a program call are expressed using a predicate defining the new data structure values in terms of the values before the call. The use of explicit data structures in these methods still violates separation of concerns.

The third approach, *history-based* [27] or *abstract* [2] specifications, only uses sequences of program calls and algebraic equations [21, 23] or logic assertions [2] on those sequences. This allows the specifications to be fully abstract, and does not bias the programmer towards a particular implementation. The importance of abstract specifications is discussed in more detail in [2, 42].

2.1.3 Concepts and terminology

We view a module as a black box that can be accessed by the rest of the program *only* through a fixed set of "access programs." There are two types of access programs: the *set-calls* change the state of the module and the *get-calls* provide information about the state of the module. It is essential that this is the only way such information can be obtained. Similarly, the set-calls are the only way the state of the module can be changed by the rest of the program.

A *trace* is a sequence of calls to access programs. With most module interfaces it is necessary to distinguish the traces constituting normal operation from those that trigger an error condition. Because of the implied deviation from normality, an access-program call provoking an error condition is often said to cause (or "raise") an *exception.* A typical example is an attempt to pop an empty stack or an attempt to read its top element. Traces representing normal operation are usually designated as "legal." A task of a module interface specification is to define which traces are legal, to categorize exceptions, and to define when each type of exception occurs.

Specifications consist of two parts:

- The *syntax* describes what access programs there are, the parameter types of each access program, and the names of the exceptions the access program may generate. It also describes the constants and types provided by the module.
- The *semantics* describes in what situations a call is legal, and the effect of a call on the legality and the values yielded by other calls.

2.2 The SCR method

2.2.1 An example

We illustrate the SCR method by means of Hoffman's example [27] of a specification of the interface of a character counter module; see Figure 2.1. We adhere to the convention that set-calls and get-calls have the prefixes **s_** and **g_** respectively. For clarity, we use the delimiters **#**, **%**, and **+** to indicate constants, exceptions, and types respectively.

INTRODUCTION

The character counter module provides a counting service for a fixed set of key characters.

SYNTAX

Program name	Inputs	Outputs	Exceptions
s_inc	+char+		%overflow%
g_cnt	+char+	+integer+	
g_tot		+integer+	

CONSTANTS

```
#maxcnt# = 5
```

EFFECTS

```
Initially:
   (forall c) g_cnt(c) = 0 and
   g_tot = 0

After s_inc(c):
   if c in {"U","V","I","C"} then
      g_cnt(c)' = 'g_cnt(c) + 1 and
      g_tot' = 'g_tot + 1
```

EXCEPTIONS

```
s_inc(c):
   {%overflow%, c in {"U","V","I","C"} and g_tot = #maxcnt#}
```

Figure 2.1: SCR specification of character counter module

The syntax of the interface is contained in the SYNTAX, CONSTANTS, and TYPES sections. The SYNTAX section describes the access programs of the module. The CONSTANTS section contains the constants and the TYPES section the types provided by the module.

The EFFECTS section describes the effect of each set-call on the values of the get-calls. Quotes are used to distinguish between the values of the get-calls before and after the invocation of the set-call being described. The first line says that initially the value of `g_cnt(c)` is `0` for each character `c`, and that the value of `g_tot` is also `0`. After `s_inc(c)`, if `c` is one of the key characters, the value of `g_cnt(c)` is one more than it was before, and the value of `g_tot` is also incremented by one. Some of the examples presented in [27] also make use of a FUNCTIONS sections to define functions that simplify the EFFECTS section.

The EXCEPTIONS section states under what conditions each exception is raised. The `%overflow%` exception for `s_inc(c)` is raised when `c` is one of the key characters and the total number of key characters has reached `#maxcnt#`. No quotes are needed in this section, since the get-calls always denote the value before the invocation of the call that raises the exception.

2.2.2 Problematic aspects of the SCR method.

There are two problems with the SCR method as it stands. One concerns the nature of the formal system used. Many formulas suggest that the language is first-order predicate logic. This has the advantage of a well-developed semantics and proof theory. However, the quotes are not part of this logic. Hoffman's explanation, which we followed above, makes the quotes reminiscent of a modal operator. Rather than to attempt to fit the method into one of the well-defined systems of modal logic, we

prefer to use Kowalski's treatment [37] of the Situation Calculus to deal with state and change within first-order predicate logic. We prefer first-order predicate logic since it has a sound proof theory which has been automated and it has more tractable proof procedures.

Another problem with the SCR method is that the specification does not determine which values of get-calls remain unchanged after a set-call. For example, the values for `g_cnt(c)` remain unchanged for all characters `c` after `s_inc(c1)` when `c1` is not a key character. The problem of specifying everything that has not changed after the execution of an operation also occurs in robot plan formation, where it is called the *frame problem.* Kowalski [37] has shown how to deal with it in a computationally feasible way in the clausal form of first-order predicate logic. As we will show, his solution is also applicable to module interface specifications.

2.3 The Situation Calculus

To achieve semantic precision, the SCR method needs to be modified so that answers to queries are logical consequences of the specification. Perhaps the smallest modification that achieves this goal is to use a system of modal logic with a model-theoretic semantics and corresponding proof theory. We have opted for an approach via logic programming based on first-order predicate logic without modal operators. This has the advantage of being a machine-executable method for answering queries.

Thus it is important for us to consider the question: Are modal operators the only way to specify change of state? McCarthy and Hayes [41] proposed the Situation Calculus as a way to use first-order predicate logic to reason about state and change. We show that Kowalski's treatment of the Situation Calculus [37], when applied to module interface specification, leads to a result that is close to the SCR method. We

call the resulting method "SCS," for Situation Calculus Specification.

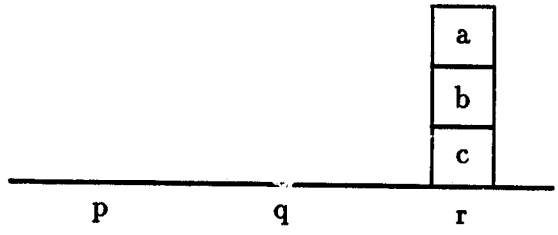
The intended application of the Situation Calculus is robot plan formation. The canonical example is that of the "blocks world," a simplification obtained by stripping away everything but a few logical problems. The blocks world consists of a few "places" and is populated by a robot hand and some blocks. The repertoire of the hand consists of a single action: to move a block from one location to another. As a block may or may not be supported by another block, a "location" can be a block or a place.

In the Situation Calculus, states are described by saying that a *condition holds in a state*. Here "holds" is a binary relation between a condition and a state. A term of the form `X.Y` is the state resulting from performing the action `Y` when in the state `X`. In the blocks world, the conditions are of the form `on(X,Y)`, saying that `X` is on the block or at the place `Y` or `clear(X)`, saying that `X` is clear. There is one action, `move(X,Y,Z)`, where the block `X` is picked up from `Y` and put on `Z`.

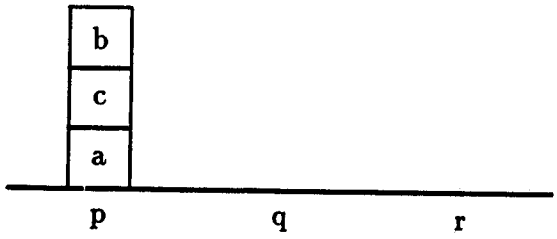
There is a distinguished state named `start` in which the blocks are arranged as described in the following six clauses (see also Figure 2.2):

```
holds( on(a,b), start ).
holds( on(b,c), start ).
holds( on(c,r), start ).
holds( clear(a), start ).
holds( clear(p), start ).
holds( clear(q), start ).
```

An important feature of the Situation Calculus is its system for naming states. A term of the form `state.action` names the state resulting from performing the action named after the "." when in the state named before it. Thus, `state.action` is the state *after* `action` if `state` is the state *before*. Typically, the term `state` is also of the



start state



final state

Figure 2.2: Initial and final state in blocks example

form `state.action`. But ultimately the blocks world must have a beginning, which happens in a state named by a term not of this form; in this example we have called it `start`. We will later see how this naming system using "result-terms"[1] allows us to avoid the modal operators of the SCR method.

The effect of picking up a block is described by saying that certain conditions hold after it:

```
holds( on(X,Y), S.move(X,Z,Y) ).
holds( clear(Z), S.move(X,Z,Y) ).
```

An action is not always possible. For example, in the blocks world there are preconditions for `move(X,Y,Z)`, namely, that `X` be on `Y`, and that `X` and `Z` be clear. This last precondition expresses the constraint that only one block can be on any block or at any place. Moreover, the preconditions have to ensure that only a block is ever picked up; never a place. This restriction is achieved by distinguishing blocks as *manipulatable* using the clauses:

```
manip( a ).
manip( b ).
manip( c ).
```

Result-terms have the right type for denoting a state, whether they do denote an actual state depends on whether the preconditions of each action are satisfied. In the Situation Calculus preconditions are specified, for each type of action, by singling out the result-terms that are "possible."

[1]The name "result-term" comes from the fact that Kowalski uses the function symbol "result" rather than the infix function symbol "." to name states obtained by applying an action to a state.

```
poss( start).
poss( S.move(X,Y,Z) ) :-
   poss( S ),
   holds( on(X,Y), S ), holds(clear(X), S ), holds(clear(Z), S ),
   manip( X ).
```

Finally, it needs to be specified that when, for example, a is picked from b and placed on c, the world only changes in those aspects directly affected by this action: that the only new conditions that hold are that a is on c and that b is clear. Moreover, that the only conditions that cease to hold are that a is on b and that c is clear. Supplying this information in such a way that an automatic inference procedure can use it is called the "frame problem." An important contribution by Kowalski [37] in his version of the Situation Calculus is that he succeeded in doing this. One of his methods is to use what was later called the SLD-resolution inference procedure [37] in combination with a number of clauses called the "frame axioms." For the blocks world, there is one frame axiom:

```
holds( C, S.move(X,Y,Z) ) :-
   different( C, on(X,Y) ), different( C, clear(Z) ),
   holds( C, S ).
```

We assume **different** defines all pairs of conditions that are not equal. One manifestation of the frame problem is that when the number of conditions becomes large, such a definition is cumbersome. As a solution, we define **different** using the clause

```
different( X, Y ) :- not X = Y.
```

where **not** is defined by negation as failure [6].

A typical query for the above clauses is:

```
?- holds(on(c,a),S),holds(on(b,c),S),poss(S).
```

There is a successful derivation by SLD-resolution substituting for the logical variable S a term that can be read as a plan to get from start to a possible state having the desired properties:

```
S = start.move(a,b,p).move(b,c,q).move(c,r,a).move(b,q,c)
```

The state that results after this plan has been executed is shown in Figure 2.2.

The Prolog language, although based on SLD-resolution, often does not find all successful SLD-derivations because it uses an incomplete depth-first search rule [39]. For example, using the above clauses, it does not find a successful derivation for the query shown. But if we ask to check whether the state shown above possesses the desired properties, then the clauses can be used as a Prolog program in the form presented here.

2.4 The character counter module interface in the SCS method

We will now show how the Situation Calculus can be applied to module interface specifications. We use the character counter interface as an example.

2.4.1 Set-calls, get-calls, and traces

We regard the set-calls and get-calls of the SCR method as the actions and the conditions of the Situation Calculus. Our notation for set-calls is the same. Our treatment causes differences in get-calls as explained below.

The traces of the SCR method closely correspond to the result-terms of the Situation Calculus. Recall that a result-term is either a constant representing the initial state (we choose `start`), or a term of the form `S.C`, where `S` is a result-term and `C` is an action. Now, since in the SCS method the actions are set-calls, the result-terms consist of the constant `start` followed by a number of set-calls separated by dots. Traces on the other hand can contain get-calls as well as set-calls and do not contain a term representing the initial state of the module.

The result-terms that are possible are an abstract representation of the state of a module. With a variable of logic it is possible to refer to a generic state resulting from performing a particular set-call: for example, `S.s_inc(C)` is the state resulting from incrementing the counter for the character `C` in the state `S`.

We now have the apparatus to treat the terms with quotes used in the SCR method. Consider the SCR clause

```
g_tot' = 'g_tot + 1
```

describing the effect of `s_inc(C)` when `C` is a key character. The quote after the term refers to the value after the set-call; the one before the term to the value before. We can obtain the SCS version by a sequence of transformations.

First, we write the equivalent clause

```
g_tot' = N+1 :- 'g_tot = N.
```

With `g_tot = N` we say that `N` denotes the same object as `g_tot`. But it is more natural to regard the access program `g_tot` as a function with an output parameter. That is, to think of `N` as having the property of being the result of the function `g_tot`, leading to the condition, in the sense of the Situation Calculus, `g_tot(N)`.

To say that the condition holds in state `S`, the Situation Calculus uses the formula

```
holds( g_tot(N), S ).
```

If S is the state before the action s_inc(C), then S.s_inc(C) is the state after it. Thus, in the Situation Calculus we express the effect of s_inc(C) on g_tot by

```
holds( g_tot(N+1), S.s_inc(C) ) :- holds( g_tot(N), S ).
```

In logic programming a term such as N+1 denotes a binary tree with symbol + as root and is unrelated to any number. For this reason the above clause has to be rewritten to

```
holds( g_tot(N1), S.s_inc(C) ) :-
   holds( g_tot(N), S ), N1 is N + 1.
```

Here N+1 denotes a term in logic that is related to N1 through the built-in predicate is.

So far we have ignored the fact that we only want to increment the total after s_inc(C) when C is a key character. To include this restriction, we have to add a condition in the body of the clause, as in

```
holds( g_tot(N1), S.s_inc(C) ) :-
   member( C, ['U','V','I','C'] ),
   holds( g_tot(N), S ), N1 is N + 1.
```

where member is the standard membership relation.

2.4.2 The EFFECTS section

The SCS version of part of the character counter specification is shown in Figure 2.3. Since the INTRODUCTION and SYNTAX sections remain the same they have been

CONSTANTS

```
maxcnt( 5 ).
```

RELATIONS

```
key_char( C ) :- member( C, ['U','V','I','C'] ).
```

EFFECTS

```
% Initially:
holds( g_tot(0), start ).
holds( g_cnt(_,0), start ).

% After s_inc(C):
holds( g_cnt(C,N1), S.s_inc(C) ) :-
   key_char( C ), holds( g_cnt(C,N), S ), N1 is N + 1.
holds( g_tot(N1), S.s_inc(C) ) :-
   key_char( C ), holds( g_tot(N), S ), N1 is N + 1.
% Frame axioms:
holds( g_cnt(C,N), S.s_inc(C1) ) :-
   not key_char( C1 ), holds( g_cnt(C,N), S ).
holds( g_tot(N), S.s_inc(C) ) :-
   not key_char( C ), holds( g_tot(N), S ).
holds( g_cnt(C1,N), S.s_inc(C2) ) :-
   key_char( C2 ), C1 \== C2, holds( g_cnt(C1,N), S ).
```

Figure 2.3: SCS specification of character counter module

omitted here. We define constants as unary relations so that we can treat them in the same way as in the SCR method. To avoid duplication, we have added the relation `key_char` to the specification. This makes the specification easier to read and makes it simpler to change the key characters that the module counts.

2.4.3 Frame axioms

As illustrated by Figure 2.3, there is a close correspondence between the SCR and SCS specifications. However, the frame axioms of the SCS specifications have no counterpart in the SCR specifications.

Yet the frame axioms are essential. If we omit the frame axioms then, for example,

```
holds( g_tot(0), start.s_inc('X') )
```

is no longer a logical consequence of the specification. When we call `s_inc(C)` with C not being a key character, all counters have to remain the same. This is expressed by the first two frame axioms. The third frame axiom tells us that the call `s_inc(C)`, where `C` is a key character, has no influence on the counters for all characters other than `C`. Apparently, the reader of the specification is supposed to assume these facts in the SCR method.

2.4.4 Exceptions

It may happen that a module is in a state such that it cannot provide its normal service when a certain access program is called. An attempt to make such a call is said to give rise to an *exception.* We call states, in which such calls occur, *exceptional.* The other states we call *normal.* For example, since our character counter can handle

at most maxcnt key characters, calling s_inc(C) with a key character when the total number of key characters has reached maxcnt raises an exception.

In the blocks world example we used to illustrate the Situation Calculus, the failure of a precondition to hold is similar to an exception. For example, we can apply a blocks world concept to the character counter module by saying that a precondition for calls to s_inc with a key character is that g_tot has not reached maxcnt. Kowalski specified the preconditions indirectly via a specification of the subset of states that are *possible*, taking the preconditions of the actions into account. This suggests that in the SCS method we specify by means of preconditions which states are *normal*. Exceptions then become the failure of one of these preconditions to hold. Kowalski's poss predicate becomes the normal predicate in the SCS method.

Thus, for the character-counter module we specify:

```
normal( start ).
normal( S.s_inc(C) ) :-
   normal( S ),
   key_char( C ), holds( g_tot(N), S ), maxcnt(M),
   N < M.          % overflow exception if not satisfied.
normal( S.s_inc(C) ) :- normal( S ), not key_char( C ).
```

In the definition of normal we associate exceptions with certain conditions in the body of the clauses. For example, the condition N < M in the second clause above is associated with the %overflow% exception.

We distinguish between the *occurrence* of an exception, its *cause*, and the exception itself. The occurrence of an exception is a result-term that, while failing to be normal, yields a normal result-term when its last call is omitted. This last call is then the cause of the exception. If the last call would not have caused an exception, then there is a clause (there can be more than one such clause) for normal that would prove the

new result-term normal. If the last call does cause an exception, such a clause is still a candidate, but fails because one of its conditions fails. If such a condition has an exception associated with it, then this is the exception itself. An action may cause more than one exception, in which case the condition described above is not unique.

For example, consider the result-term

```
start.s_inc('U').s_inc('V').s_inc('I').s_inc('C').
   s_inc('U').s_inc('V')
```

This is an occurrence of the exception **%overflow%**. The last call **s_inc('V')** is the cause. Of the three clauses defining **normal**, the second is one of the candidates that could prove the result-term normal, given that the result of omitting the last call is normal. But this clause cannot prove normality, because the condition **N > M** fails. Thus **overflow** is the exception. The third clause of **normal** is another candidate for proving the result-term normal. In this case the condition **not key_char(C)** fails, but there is no exception associated with this condition.

We have approached the specification of exceptions by starting from normality. Our result is that the exception itself is a proof-theoretic concept, not an object denoted by a term of logic. In this approach, exceptions are defined rather indirectly.

Another approach to defining exceptions starts from a definition of the subset of result-terms that are exceptional. Such a definition is easy to write, once we have the one for normality available. The definition is further simplified when we realize that the only result-terms of interest are those that are normal up to the last call. Then we only have to specify when a normal result-term turns into an exceptional one. Each such incident can be associated with a specific exception. As a result we can be more specific than a mere definition of the set of exceptional result-terms: we

can associate each exceptional result-term with an exception by means of a binary relation.

For the character counter module this approach gives:

```
s_exception( overflow, S.s_inc(C) ) :-
   normal( S ),
   key_char( C ), maxcnt( N ), holds( g_tot(N), S ).
```

We have flagged our predicate `s_exception` by an `s_` because it is only exceptions for set-calls that can be treated in this way. As we will see later, exceptions for get-calls can be handled similarly.

We have to ask ourselves whether all exceptions can be defined in this way. Whether an exception occurs depends on whether the module is in a well-defined set of states. So the question can be rephrased to whether we can define all sets of states. Unfortunately this is not the case, as there can be undecidable sets of states. However, in practice we have found that all the exceptions we were interested in depended on decidable sets of states.

2.4.5 Queries

To illustrate the use of the SCS specification, Figure 2.4 shows some queries about the behaviour of the module. The first asks the value of the counter for the character `'U'` for the result-term shown. Prolog proves that the formula in the query is a logical consequence of the specification, provided that the variable `N` is substituted by `1`. Similarly, in the next three queries, the Prolog attempt at proof is successful, with `N` substituted by `0`, `0`, and `2` respectively. The fifth query is successful, indicating that the result-term shown does not raise an exception. The success of the last query indicates that its trace does give rise to an exception; `E` is instantiated with `overflow`.

QUERIES

```
?- holds( g_cnt('U',N), start.s_inc('U') ).
                Answer: Yes, N = 1.
?- holds( g_cnt('V',N), start.s_inc('U') ).
                Answer: Yes, N = 0.
?- holds( g_cnt('X',N), start.s_inc('X') ).
                Answer: Yes, N = 0.
?- holds( g_tot(N), start.s_inc('U').s_inc('X').s_inc('V') ).
                Answer: Yes, N = 2.
?- normal( start.s_inc('U').s_inc('V') ).
                Answer: Yes.
?- s_exception( E, start.s_inc('U').s_inc('V').s_inc('I').
     s_inc('C').s_inc('U').s_inc('V') ).
                Answer: Yes, E = overflow.
```

Figure 2.4: Character counter queries

2.5 Using state equivalences

2.5.1 Stack specification

Let us now consider the interface specification of an unbounded integer pushdown stack module. The INTRODUCTION and SYNTAX sections for such a module are shown in Figure 2.5.

An interesting feature of this stack module is that, in the sense of systems theory, it is not "observable"[2]: its future behaviour cannot be predicted based on the present values of the get-calls. For example, it is impossible to predict the value of **g_top** after a call to **s_pop** based on the values of **g_top** and **g_depth** before the call. As a result it is not possible to specify the module interface only in terms of how the access programs interact.

[2]Others would say that the module has *delayed effects* [2] or that it is not *state apparent* [27].

INTRODUCTION

The stack module provides an unbounded pushdown stack of integers.

SYNTAX

Program name	Inputs	Outputs	Exceptions
s_push	+integer+		
s_pop			%empty%
g_top		+integer+	%empty%
g_depth		+integer+	

Figure 2.5: Stack INTRODUCTION and SYNTAX

One proposed solution to this problem is the use of "hidden programs" in the specifications [2, 27]. These hidden programs need not be implemented; they only serve to help specify the interface. Unfortunately, these hidden programs do suggest data structures and possible implementations of the program [2] and as such violate the separation of concerns principle. We would like to define the interface of a stack abstractly, without using hidden programs. What we need is the concept of an *abstract* state.

This concept is familiar in automata theory. For example, the well-known text by Minsky [45] considers a machine M with two identical copies M_1 and M_2 subjected to possibly different input histories. Minsky defines the histories equivalent with respect to M if every possible sequence of future inputs would elicit the same behaviour from M_1 and M_2. The relation between histories thus defined is an equivalence relation. Minsky calls the equivalence classes induced by this relation the "internal states" of the machine M. They seem as abstract as a state can get.

A module can be considered as a machine; set- and get-calls are inputs in the sense of Minsky. Hence states of a module can be defined as equivalence classes of

histories of set- and get-calls. Bartussek and Parnas [2] have done this, referring to these histories as "traces." They define the equivalence classes by asserting equations between sets of traces defined by including variables in terms denoting a trace.

2.5.2 Using state equivalences in the SCS method

As we already noted, the normal *result-terms* of the SCS method represent the possible states of the module. They are similar to the *histories* in the sense of Minsky and the *traces* of Bartussek and Parnas. We follow Bartussek and Parnas by asserting equations between result-terms to define abstract states, which are the equivalence classes induced by the equations. In [2] these equations are called "trace equivalences." As we are dealing with result-terms denoting states we will refer to them as "state equations." We will refer to the equivalences induced by these equations as "state equivalences."

There are several differences between our result-terms and the traces of Bartussek and Parnas. One is that result-terms only contain set-calls, whereas traces contain both set- and get-calls. Another is that a result-term is either a term representing the initial state or the composition of a result-term and an action. Thus it has the form

$$start \cdot a_1 \cdot \ldots \cdot a_n$$

where *start* names the initial state and $a_1, \ldots, a_n$ are actions. At first sight this may look like a sequence of actions, but it is not the same: at each occurrence of the dot operator we have a result-term (i.e., a term representing a state) to the left and an action to the right.

A more substantial difference is that in the SCS method we define equivalences in the set of *all* result-terms, not only the normal ones. In the trace method, two traces

can only belong to the same equivalence class if they both represent legal states. Since the `normal` relation already classifies which result-terms represent normal states and which ones do not, there is no need to make this restriction.

A third difference is in connection with exceptions. In Kowalski's axiomatization there is a distinction according to whether a result-term denotes a state (is "possible"). In the trace method the corresponding term for traces is "legal." We find this usage puzzling, as it means that such a trace does not give rise to an "exception." We prefer to use "normal" for the counterpart of "legal" in the trace method and of "possible" in Kowalski's axiomatization, as "normal" is the contrary of "exceptional."

But the main principle of the trace method, that equivalence classes of traces are abstract states, carries over to the SCS method with undiminished force. In logic, a term names an object. Result-terms in the SCS method are terms of logic denoting states. An equality between two such terms means that they denote the same abstract state.

In the present example we specify a stack module interface by means of state equations in the SCS method. We can adopt the same equation as in [2] to express the fact that `s_pop` cancels `s_push(I)`:

```
T.s_push(I).s_pop = T.
```

This is a clause of first-order predicate logic. As it consists of a single atomic formula (with predicate symbol "`=`"), it unconditionally asserts that the relation named by "`=`" holds between the left-hand side and the right-hand side. As all clauses, it is implicitly universally quantified in all its variables (in this case `T` and `I`).

We need to explain an apparent contradiction: although we took *equivalence* classes as a starting point, we ended up writing an equation, an assertion involving the *equality* relation. The terms in the two sides of the equation denote the same

EFFECTS

```
% State equation(s):
T.s_push(I).s_pop = T.

% Equivalence axiom:
holds( Cond, State ) :- State = State1, holds( Cond, State1 ).

% Initially:
holds( g_depth(0), start ).

% After s_push(i):
holds( g_depth(N), S.s_push(I) ) :-
   holds( g_depth(N1), S ), N is N1+1.
holds( g_top(I), S.s_push(I) ).
```

Figure 2.6: Stack EFFECTS

object, as they do in $2 + 2 = 4$; hence the equality relation. But the terms themselves are not the same. We can classify terms according to whether they denote the same object. This gives rise to equivalence classes of *terms*, based on equality among *objects*.

Apparently, equations can be used to define equivalence classes of terms. Result-terms represent states, so equations can be used to define equivalence classes of these terms, that is, abstract states. With the use of state equations, the EFFECTS section of the SCS specification of the stack module interface becomes as shown in Figure 2.6.

Using the state equation, we can now rewrite any normal state to one containing **start** followed by zero or more **s_push** calls. This means that in the EFFECTS section we do not need to describe the effect of **s_pop**.

2.5.3 Logic "with" or "without" equality

The SCS method is based on logic programming: an automatable method of deduction allowing us to execute module interface specifications on a computer. This automation has been achieved at the cost of dropping certain features often taken for granted in logic, such as built-in equality.

In logic programming we use logic without equality. That is, equality is a relation without any special status. For example, although it may seem obvious that in equal states the same conditions hold, it is necessary to include in Figure 2.6 a clause saying this explicitly:

```
holds( Cond, State ) :- State = State1, holds( Cond, State1 ).
```

The need to say more about the equality relation becomes apparent when we try to answer the query

```
?- holds( g_depth(N), start.s_push(1).s_pop.s_push(2) ).
```

We expect success with N equal to 1, but instead we obtain a response indicating that no N can be found satisfying the stipulated conditions. This problem can be traced back to the fact that

```
start.s_push(1).s_pop.s_push(2) = start.s_push(2)
```

is not a logical consequence of the program although the two sides of the equation denote the same state. This is because = is treated like any other predicate symbol, so that nothing is known about this binary relation except the state equations given in the specification.

This defect can be remedied by including the clauses

```
T = T.
T = U :- U = T.
T = V :- T = U, U = V.
T1.C1 = T2.C2 :- T1 = T2, C1 = C2.
```

expressing reflexivity, symmetry, transitivity, and substitutivity for the function symbol ".". These are the standard equality axioms, but it will turn out that we will not use symmetry and that we will use the remaining clauses in a different form.

By the completeness of SLD-resolution [39], we know that an SLD-derivation can be found for

```
start.s_push(1).s_pop.s_push(2) = start.s_push(2)
```

from a program containing these axioms. Unfortunately, when Prolog is given the query

```
?- start.s_push(1).s_pop.s_push(2) = start.s_push(2).
```

it does not find a successful derivation[3]. This is because the search space contains at least one infinite derivation to the left of the leftmost successful derivation. With its depth-first, left-to-right traversal algorithm, Prolog continues to construct the leftmost infinite derivation [62].

2.5.4 Simplification rules and canonical forms

We saw that the equations by themselves do not say enough about the equality relation. Supplementing them with the standard equality axioms (reflexivity, symmetry,

[3]Logic programmers would blame the incomplete search strategy used by Prolog for this behaviour, whereas Prolog programmers would blame the clauses defining the equality axioms.

etc.) gives a logically complete definition, that requires deductions of a kind more complex than Prolog will normally handle. How do we continue towards our goal to make module interface specifications executable by Prolog, even when they contain state equations?

We first need to review the purpose of state equations and to introduce the concepts of *simplification rule* and *canonical form.* That done, we can present our method of making Prolog perform the required simplification operations. Considerable research has been performed in the area of term rewriting and in using equations as simplification rules. We will only mention the concepts of importance to us here; a more thorough discussion is contained in [32].

The main role of an equation is to state that its left- and right-hand sides denote the same object. But usually there is an important subsidiary role as well: to simplify expressions. This can be done by finding an expression containing an occurrence of an instance of the left-hand side of the equation. The simplification is obtained by replacing that occurrence by the corresponding version of the right-hand side of the equation: of course the result is a simplification only if the right-hand side is simpler than the left-hand side. Not all equations can be used as a simplification rule, but all the ones we used in our specifications could be used as such.

Thus, it is no coincidence that our example state equation is written as

```
T.s_push(I).s_pop = T.
```

although it is as true when written the other way around. As given, it is not only an equation, but also a simplification rule.

This rule allows any result-term that denotes a stack state to be simplified to one not containing any occurrences of **s_pop**. Such a term cannot be simplified any

further and is called *canonical* for this reason. Certain sets of simplification rules have the property, called *confluence*, that any given term can be simplified to exactly one canonical form. That is, simplification always ends up in the same canonical form. A set of simplification rules is *noetherian* or *finitely terminating*, if the rewriting is always guaranteed to end. This requirement excludes the possibility of continually going around in a circle while simplifying. Sets of rules that have both properties are called *canonical.*

Canonical sets of simplification rules have the attractive property that every equivalence class has exactly one canonical form. This form is easy to find, as one can start from any member of the equivalence class and continue simplifying until this is no longer possible. As observed by Bartussek and Parnas [2], these canonical forms make module interfaces easier to specify, as one only needs to specify properties of terms in canonical form. For example, in Figure 2.6 we did not need to say anything about traces ending in `s_pop` when defining the `holds` relation, as these do not occur in canonical forms.

To make Prolog perform the required simplification operations, we use the method proposed in [62]. We replace the conventional equality axioms by others of which it can be shown (see [62]) that they have the same meaning when the two new predicate symbols `eq` and `eq1` are taken to denote the same relation as `=`.

```
eq( X, Z ) :- eq1( X, Y ), eq( Y, Z ).
eq( X, X ).

eq1( X, Y ) :- X = Y.
eq1( X.C, Y.C ) :- eq1( X, Y ).
```

The query

```
?- eq1( T1, T2 ).
```

succeeds if T2 can be obtained from T1 by applying a state equation to T1 itself or to part of it. To prove that the eq relation holds, one uses transitivity if an equation can be applied, otherwise one uses reflexivity. If we add these axioms to our specification, and put the equivalence axiom

```
holds( Cond, State ) :- eq( State, State1 ), holds( Cond, State1 ).
```

at the end of the specification, then the specification behaves as desired.

2.6 Writing SCS specifications

2.6.1 What is the SCS method?

Although we have given only two examples, we have referred to the "SCS method," suggesting that the method is more generally applicable. We have verified that this is the case by defining interfaces for various examples from the literature ([2, 27, 28, 30]). These include: symbol table, lexical scanner, queue, traversing stack, tree, and a shortest path module.

When do we consider a module interface specification to be one that follows the SCS method? In general, it has to be the result of applying Kowalski's axiomatization of the Situation Calculus to the concepts of module, module interface, and access program as introduced by Parnas. Within this general framework, the following adaptations are made in the SCS method:

- A set-call plays the role of an action in the Kowalski axiomatization.
- A get-call plays the role of a condition.

- Exceptions correspond to the failure of preconditions to hold.

Although most of the SCS method is an application of the Kowalski axiomatization, SCS is an extension of Kowalski's work by incorporating the main idea of the trace method. Note that our adaptation of this idea by means of the state equivalence axiom

```
holds( Cond,State ) :- State = State1, holds( Cond, State1 ).
```

and the various state equations can be part of a Kowalski-type axiomatization independently of whether a module or something else is being specified.

2.6.2 Simplifying SCS specifications

Kowalski's axiomatization of the Situation Calculus has proven to be a good way to deal with state and change. It allows us to write both SCR and trace-like specifications in first-order predicate logic. When using this method to actually write specifications, we find that there are some awkward features to it. Therefore, we have made some changes that facilitate the use of the SCS method. To illustrate this, Figure 2.7 shows the actual SCS specification of the stack module we use.

Some of the changes are made to make the specifications usable as Prolog programs. Lines preceded by a % are treated as comments. The SYNTAX section has been altered so that it can be easily entered and maintained in a text file. Result-terms are represented as lists. For example, the result-term `s_init.s_push(1)` is represented as the list `[s_push(1),s_init]`. The order of the calls is reversed in the list representation so that the last call can easily be accessed as the head of the list.

Other changes are more substantial. We will discuss each of these as well as the remaining differences with the trace method.

```
%
% INTRODUCTION
%
%   The stack module provides an unbounded pushdown stack
%   of integers.
%
% SYNTAX
%
%   s_init.
%   s_push( integer ).
%   s_pop.                  %empty%
%   g_top -> integer.       %empty%
%   g_depth -> integer.

%
% EFFECTS
%
% State equation(s):
equal( [s_pop,s_push(I)|T], T ).
% Equivalence axiom:
holds( C, S ) :- eq( S, CS ), holds1( C, CS ).
% Initially:
holds1( g_depth(0), [s_init] ).
% After s_push(I):
holds1( g_top(I), [s_push(I)|S] ).
holds1( g_depth(N1), [s_push(_)|S] ) :-
   holds1( g_depth(N), S ), N1 is N + 1.

%
% EXCEPTIONS
%
% empty
exception( empty, s_pop, S ) :- holds( g_depth(0), S ).
exception( empty, g_top(_), S ) :- holds( g_depth(0), S ).
```

Figure 2.7: SCS specification of stack module

A call to initialize the module. So far, we have used `start` to indicate the initial state of the module. In practice it is useful to have a set-call to initialize the module. We refer to this set-call as `s_init`, and use it in the result-terms to represent the initial state of the module. Note that in a result-term `s_init` represents a state, whereas all other set-calls represent actions that transform one state to another.

The `holds1` relation. We introduce the `holds1` relation so that the specification behaves better when executed as a Prolog program. As a result, the equivalence axiom has been changed from

```
holds( C, S ) :- eq( S, CS ), holds( C, CS ).
```

to

```
holds( C, S ) :- eq( S, CS ), holds1( C, CS ).
```

Previously, the specification would not succeed with a query such as

```
?- holds( g_depth(D), [s_pop,s_push(1),s_init] ).
```

because the search space contains an infinite derivation (one that uses the equivalence axiom at every step) to the left of the leftmost successful derivation. By changing the equivalence axiom, and by expressing the value of `g_depth` and `g_top` using the `holds1` relation, there are no infinite derivations in the search space. The intended meaning of `holds1` is the same as that of `holds`; the difference is that it only specifies the value of get-calls for result-terms in canonical form.

The `exception` relation. To treat exceptions for set- and get-calls in the same way, we have replaced the **`s_exception`** and **`g_exception`** relations by **`exception`**. Using **`s_exception`** and **`g_exception`** the EXCEPTIONS section would have been

```
s_exception( empty, [s_pop|S] ) :-
   normal( S ),
   holds( g_depth(0), S ).
g_exception( empty, g_top(_), S ) :-
   normal( S ),
   holds( g_depth(0), S ).
```

This shows how we can deal with exceptions for get-calls in a similar way as we deal with exceptions for set-calls. Just as **`s_exception`** defines why some result-terms fail to be normal, **`g_exception`** defines why certain instances of the **`holds`** relation fail to be true.

For all practical purposes, an exception means that an access program was used in a way not intended by the designer of the module. As such, there is no difference whether the exception is raised by a set-call or a get-call. For this reason we have replaced the two relations by the single relation **`exception`**. We have also dropped the condition that the state in the third argument must be **`normal`**. However, we have not changed the intended meaning of the **`exception`** relation. It still explains why an access program cannot be called when the module is in a certain state.

The `normal` relation. We have eliminated the definition of the **`normal`** relation from the specification. The clauses

```
normal( [s_init] ).
normal( [C|S] ) :- normal( S ), not exception( _, C, S ).
```

could be used in the above specification to define the **normal** relation. Since this would be the desired definition of the **normal** relation, there is no need to define it in another way. Such a redundancy would be a likely cause for errors in the specification. We prefer the **exception** relation over the **normal** relation, since it defines the exceptions more directly and it provides a means for naming the exceptions.

Remaining differences with the trace method. With the above changes, the SCS method closely resembles the trace method for module interface specifications. There are some remaining differences that warrant the introduction of a new method.

The first, and most important one, is the language used and its semantics. By using first-order predicate logic, we combine a number of important advantages:

1. We can rely on the standard semantics of first-order predicate logic.
2. It has a sound proof theory that is automated with an efficiency allowing deduction steps to be used as computation steps.
3. It allows a wide range of concepts of interest in computing to be expressed in a computationally tractable way.

McLean [42] has also addressed the problem of providing a well-defined semantics for the trace method of Bartussek and Parnas. McLean has chosen not to build on the well-established semantics of first-order predicate logic, but to start from scratch. This does not seem to be harder to do than for first-order predicate logic. The problem that we see is that it does not seem to be easier either. Since we have shown that the trace method can be viewed as a straightforward application of first-order predicate logic, it seems that there is little incentive for an independent semantical treatment as given in [42].

The traces of the trace method and our result-terms have an important difference. While traces can contain both set- and get-calls, result-terms only contain set-calls. Since result-terms are used to represent states and get-calls cannot alter the state of a module, we see no need to include get-calls. The concept of a trace as a sequence of access program calls is still useful in our method (for example, as we will discuss in the next section, when we talk about testing a specification) and therefore we use result-terms to refer to the terms representing states.

A final difference is that we define exceptions explicitly through terms in logic. The trace method only defines the traces that are "legal," the counterpart of our "normal." This is not a major difference, and in [48] exceptions are also dealt with more directly than in the original version of the trace method [2].

2.7 Executable specifications

2.7.1 Why executable specifications?

Specifications must not be ambiguous. We have shown that not only the syntax, but also the meaning of specifications can be precisely defined. But semantic precision is not enough. Lack of it is not always the cause of specifications being interpreted in a way different from the one intended by the author. Experience has shown that authors often write specifications that say with absolute precision something different from what they intend.

Thus it is useful to have an impartial agent confront the author with logical implications of what is specified. Then the author has a better chance of detecting deviations between what is specified and what is intended, a procedure sometimes provocatively referred to as "debugging the specification." The impartial agent can

be a logically sound inference system running on a computer and proving that, for a given result-term, get-calls yield certain values. In analogy to programs, we can call this procedure the "running" of the specification. Running specifications can also be viewed as a form of rapid prototyping. Except that, in this case, the term "instant prototyping" is more appropriate.

There is another reason for having executable specifications. When testing a module implementation, one has to determine the expected output for each selected test input. To allow for a generous number of test cases, it is important that these outcomes are generated automatically as logical consequences of the specification.

Several authors [15, 30, 43, 44] have worked at translating trace specifications systematically to Prolog to obtain executable forms of the specifications. Although there is a resemblance with our work in that this also leads to Prolog programs, these authors chose Prolog as a programming language only because it is more convenient than its alternatives. They do not show that the semantics of the specifications coincides with the semantics of the resulting Prolog programs.

2.7.2 Running the specification

We obtain instant prototyping by writing the specification such that it can be run as a Prolog program. Because Prolog contains, apart from a purely logical core, various non-logical features, we have to avoid in the specification constructs that require the "occurs check" [39], as it is lacking in most Prolog implementations. In addition, we have to ensure that negated conditions are used in a way that does not violate the restrictions of Prolog's "negation by failure" method (see [54]). If these precautions are observed, then the non-logical features are not activated: Prolog acts as a sound theorem prover.

For example, when we load the stack specification as a Prolog program, we can ask it queries such as

```
?- holds( g_depth(N), [s_push(7),s_push(3),s_init] ).
```

and Prolog will instantiate N with 2. This direct use of the specification is clumsy and requires a lot of work to run even simple cases. We obtain a more natural behaviour of the specification by building an interface to it.

The interface we use for the stack module is

```
s_init :- execinit( s_init ).
s_push( I ) :- integer( I ), execset( s_push(I) ).
s_pop :- execset( s_pop ).
g_top( I ) :- var( I ), execget( g_top(I) ).
g_depth( I ) :- var( I ), execget( g_depth(I) ).
```

With this interface the user can enter calls to the module one at a time. If the call is syntactically correct (this is checked by the interface) it prints out any exceptions caused by such a call and, if the call is a get-call, it returns the value by instantiating the appropriate variable.

The relations **execinit**, **execset**, and **execget** provide the interface to the specification. The definition of these three relations is shown in Figure 2.8. These predicates maintain the state of the module, which can be done in several ways in Prolog. One of these is to add two arguments to each of the access programs representing the state before and after the call to the access program. Another method is to use Concurrent Prolog. It allows processes with internal states by using tail recursion and local variables [53]. We use the method that is also used in [30]. The state of the module is maintained in a global variable by using the builtin procedures **abolish** (which removes the definition of a predicate from the program), **assert**, and **retract**.

```
execinit( Call ) :-
   abolish( state, 1 ), initstate( Call, S ), assert( state(S) ).

execset( Call ) :-
   state( S ), exception( Exc, Call, S ), !, p_addexc( Exc ).
execset( Call ) :-
   retract( state(S) ), !, newstate( Call, S, NS ),
   assert( state(NS) ).

execget( Call ) :-
   state( S ), exception( Exc, Call, S ), !, p_addexc( Exc ).
execget( Call ) :- state( S ), holds( Call, S ).

initstate( Call, [Call] ).
newstate( Call, State, [Call|State] ).
```

Figure 2.8: Prolog interface to SCS specifications

The clauses

```
initstate( Call, [Call] ).
newstate( Call, State, [Call|State] ).
```

in Figure 2.8 express how result-terms are used to represent the state of the module. The first clause says that, given call `Call` to initialize the module, `[Call]` represents the initial state of the module. The second one says that, given set-call `Call` and current state `State`, `[Call|State]` is the result-term representing the state after executing `Call`.

2.7.3 Testing specifications

The above interface allows us to run a specification and test if it behaves according to our expectations in particular situations. When this is not the case we change the

specification and test it again. Similarly, when we change the specification at a later point in time, the specification will have to be tested again. Systematic module testing requires tens or hundreds of test cases. Consequently, automation has considerable potential. With automation, the test cases can be executed economically after every change to the specification. Later on, when we implement the module, we can use the same test cases to test the implementation. To automate part of the testing, we use PROTEST, a system which helps the user maintain and execute large numbers of test cases [29].

PROTEST consists of several Prolog programs that test C or Prolog implementations of modules. Test cases are defined using a language based on module traces, which was originally developed for PGMGEN [26], a system written in C to test modules implemented in C. PROTEST consists of two systems: PROTEST/1 and PROTEST/2. PROTEST/1 tests a single implementation of a module, and PROTEST/2 compares the behaviour of two implementations of the same module.

With PROTEST/1, we can test an SCS specification using a large number of test cases. The same test cases can later be used to test an implementation of the module. A test case consists of a trace to exercise the module and a description of the required behaviour of the module in response to that trace. Test cases are defined using the unary predicate **case** whose argument is a term of the form

case(*trace*,*expexc*,*actval*,*expval*,*type*)

where:

trace is a trace to exercise the module.

expexc is the name of an exception that *trace* is expected to generate, or **noexc** if no exception is expected.

```
/*****Exceptions*****/
/*empty*/
case([s_init,g_top(_)],empty,dontcare,dontcare,dontcare).
case([s_init,s_push(1),s_pop,s_pop],
   empty,dontcare,dontcare,dontcare).
/*****Normal case*****/
case([s_init],noexc,g_depth(_),0,int).
case([s_init,s_push(10)],noexc,g_top(_),10,int).
case([s_init,s_push(10)],noexc,g_depth(_),1,int).
```

Figure 2.9: PROTEST/1 test cases for the stack module

actval is a get-call to be evaluated after *trace*, whose return value is taken to be the "actual value" of the trace (or **dontcare** if no value checking has to be performed).

expval is the expected return value of *actval*.

type the data type of the value returned by *actval*.

For example, the Prolog code in Figure 2.9 defines five test cases for the stack module.

For each test case, PROTEST/1 executes the calls in the trace while monitoring the behavior of the implementation. When the observed behaviour is not the same as the expected behaviour, as specified in the test case, a message is printed out.

2.7.4 Specifications as oracles

The use of an executable specification as a test oracle has tremendous benefits for testing an implementation. The tester can concentrate solely on generating inputs, unconstrained by the significant burden of generating the expected outputs. PROTEST2

```
/*****Exceptions*****/
/*empty*/
case([s_init,g_top(_)],docare,dontcare,dontcare).
case([s_init,s_push(1),s_pop,s_pop],docare,dontcare,dontcare).
/*****Normal case*****/
case([s_init],docare,g_depth(_),int).
case([s_init,s_push(10)],docare,g_top(_),int).
case([s_init,s_push(10)],docare,g_depth(_),int).
```

Figure 2.10: PROTEST/2 test cases for the stack module

compares the behaviour of two implementations of the same module. A possible application is to compare the behaviour of an implementation to an SCS specification.

A PROTEST/2 test case consists of a trace and a description of what part of the behaviour of the two implementations needs to be compared. As for PROTEST/1, test cases are defined using the **case** predicate. Each test case is of the form

case(*trace*,*excflag*,*actval*,*type*)

where *trace*, *actval*, and *type* are as for PROTEST/1, and

excflag is either **docare** or **dontcare**, indicating whether the exception behaviour of the implementations has to be compared or not.

The test cases in Figure 2.10 are the PROTEST/2 test cases corresponding to the PROTEST/1 test cases shown in Figure 2.9.

For each test case, PROTEST/2 executes the calls in the trace on both implementations and monitors the behaviour of them. When the behaviour of the two implementations is not the same, a message is printed out.

2.7.5 Do we need implementations?

So far we have assumed that after a semantically precise, executable specification has been obtained, a module is implemented in a programming language unrelated to logic. Efficiency is the principal reason for requiring any implementation beyond an executable SCS specification. And indeed, logic programs (which is what our executable specifications are) often require more memory or time than equivalent programs in typical implementation languages.

However, there are developments in logic programming suggesting that our method may be the starting point of partial automation of the implementation stage. In the next section we will look at program transformation; a technique that allows us to make a program more efficient while maintaining its meaning.

Chapter 3

Logic-based Program Transformation

Program transformation allows us to convert programs that are obviously correct, but possibly inefficient, into more efficient ones such that the correctness of answers is preserved. In this chapter, we discuss the transformational style of programming, which allows the programmer to first work on the correctness of a program while ignoring efficiency, and then to increase the efficiency while preserving the correctness. We present a method for using sets of "frontiers" to obtain a specialized version of a program for a particular query using the "unfold" transformation rule. Where existing methods either do not preserve completeness or introduce redundancy, this method guarantees completeness while avoiding redundancy. We then describe the other transformation rules we use to transform logic programs. Finally, we describe FROST (which stands for "frontier transformation system"), a program transformation system for logic programs based on this method.

3.1 Transformational approach to programming

Logic programming, as defined by Kowalski [36], is a form of declarative programming. This allows the programmer to specify *what* has to be computed, rather than giving an algorithm that describes *how* the desired result has to be computed, as is done in imperative programming languages. In Kowalski's words [37], it allows the programmer to define the logic component of the algorithm; the control component is provided by the Prolog interpreter or any other execution mechanism[1]. This makes it easy to understand and modify a logic program, and we can regard it as a specification of the problem we are trying to solve.

Unfortunately, these specifications are often not efficient enough as computer programs, and hence unacceptable as final implementations. In these cases, we want to obtain a more efficient version of the program that is equivalent to it. This can be done by writing a program in a language unrelated to logic and then showing that this program is equivalent to the specification. Another approach is to transform the specification to make it more efficient while preserving its meaning. We discuss both approaches.

Implementing the program. The first approach is to write an efficient program and then show that it is equivalent to the specification. With this approach, the program can be written in any programming language. By chosing an appropriate programming language, such as Pascal or C, it can be easier to obtain an efficient program than when programming in logic. However, this also complicates the process of showing the equivalence of the program and the specification.

[1]It can be useful to alter the control strategy. However, the advantage that the logic and the control of the program are separated remains.

With *program verification* we try to prove that the program behaves according to the specification. Ideally, program verification will help with the production of correct software, but there are several problems with this method that have limited its use.

Since the specifications themselves are often complicated, it may be hard to decide whether the specification correctly represents our intentions or not. In this case it would not be very helpful to prove that the implementation behaves according to the specification. When we regard our initial logic program as a specification, we have the advantage that we can run it and see if it behaves as expected.

Another problem occurs when the implementation is written in a language unrelated to logic. In this case the program verification step has to compare a formula from predicate logic (the initial program) to a program written in some other language.

De Millo and Lipton argue that the formal verification of programs will never play the same role in computer science as proofs do in mathematics [11]. This is not only caused by the above two problems, but the absence of continuity and the inevitability of change (think of what a small change can do in a large system, where small refers to the amount of text involved) make it impractical to build such a verification system.

Testing a program means comparing the behaviour of the program to a specification for a selected set of inputs. For practical applications, the number of tests needed to obtain a reasonable level of confidence in the correctness of a program is extremely large. As Dijkstra points out [13], testing can only show the presence of bugs, never their absence. This limits the use of testing for obtaining confidence in the correctness of an implementation.

The transformational approach. We prefer the transformational approach to programming, where, after we have a correct but possibly inefficient version of the

program, we apply certain transformation rules to increase the efficiency. The transformation rules are chosen so that they preserve the meaning of the program. There are two problems with this method.

We have to verify that the transformation rules preserve the meaning of the program. This can be just as hard as program verification, but in this case we only need to verify each rule once and then we can use it repetitively.

As is pointed out in the literature, there is also a problem with the practical implementation of the transformational approach to programming [1, 9, 49, 51]. This is because the process of program transformation is hard to automate. Most systems mentioned in [49] are interactive: the user selects the strategy and/or rule to use, and the system applies it. If we could obtain a fully automatic system, it would operate like an extremely powerful optimizing compiler. However, such a system would be unmanageable in size.

3.2 The frontier theorem

3.2.1 Soundness and completeness

The "unfold" and "fold" transformations were introduced by Burstall and Darlington [3] to make functional programs more efficient. What they had in mind was a system of "computer-aided programming" where a specification in functional form is transformed by means of a computer program into a functional program of acceptable efficiency. Independently of each other, Clark [5] and Hogger [31] have shown that these transformations also apply to logic programs. They pointed out that in this context there is an additional advantage: the transformed version is a logical consequence of the original. As a result, any answer obtainable from the transformed

program is also an answer obtainable from the specification. The properties of logical consequence guarantee soundness with respect to the specification. This soundness is sometimes called *partial* correctness: it is not always clear whether all answers obtainable from the specification can also be obtained from the result of the transformation. When this is the case, we say that the transformed program is *complete* with respect to the specification.

The unfold transformation is also called "partial evaluation" or "symbolic execution." As such, it appeals more directly to programming intuition: the transformed program is the result of replacing procedure calls by the appropriately instantiated bodies of their definitions.

But partial evaluation is only useful if we know it yields a complete result, otherwise we cannot discard the inefficient specification. Tamaki and Sato [59] show that certain transformations on clauses, including unfolding, preserve completeness. To make the results of Tamaki and Sato applicable to our purpose, it is necessary to eliminate from the program those clauses that are redundant with respect to the desired query. This can be achieved by means of a dependency analysis done after the transformations, but we can also obtain the desired result in a single operation by building what we call a "complete set of frontiers."

In doing so, we take as starting point the idea of Vasey [63], who shows that the result of unfolding can be regarded as a "qualified answer." We obtain the desired completeness by considering frontiers of "conditional answers" (our preferred terminology for Vasey's qualified answers) in a suitable derivation tree. As a first approximation, our completeness is based on the fact that in the derivation tree no computation can "escape" the frontier. This observation is only an approximation, because in all but a few trivial cases one has to consider more than a single derivation tree with frontier. We characterize when enough frontiers have been found. These

are then a set of clauses obtained by unfolding which is complete with respect to the specification.

Using a different criterion for completeness, Lloyd and Shepherdson [40] independently found a result similar to the frontier theorem.

3.2.2 Complete sets of frontiers

From now on, we assume without loss of generality that each query posed to a logic program consists of a single goal. If this is not the case, and we have a query of the form $?G_1, \ldots, G_n$ for $n > 1$, we can add the clause $g(X_1, \ldots, X_m) \leftarrow G_1, \ldots, G_n$ to the program and replace the query by $?g(X_1, \ldots, X_m)$. Here g stands for any predicate symbol not appearing elsewhere in the program, and $X_1, \ldots, X_m$ are the variables in $G_1, \ldots, G_n$.

A *derivation tree* for a query Q is a tree with Q at the root. Each node in the tree is a query consisting of a conjunction of atomic formulae, called the *goals* of the query. If the query is not empty, it has a *selected goal*. A node N, consisting of the query $?G_1, \ldots, G_n$ with selected goal G_i has a child for each clause whose head unifies with G_i. If G_i unifies with the head of the clause $A \leftarrow B_1, \ldots, B_m$ with the most general unifier θ, then the corresponding child consists of the query $?(G_1, \ldots, G_{i-1}, B_1, \ldots, B_m, G_{i+1}, \ldots, G_n)\theta$.

A *derivation* is a path starting from the root in the derivation tree which is either infinite or ends in a leaf node of the derivation tree. A *successful* derivation is a derivation ending in an empty query, denoted by $\Box$. A *failed* derivation is a derivation ending in a non-empty query with no children. It follows from the definition that this only happens if the selected goal does not unify with the head of any clause of the program. A *partial derivation* is a path starting from the root in the derivation tree.

It can end in a non-leaf node of the derivation tree.

It should be noted that Prolog finds its answers by constructing a particular type of derivation tree, the *Prolog derivation tree*. This derivation tree is obtained by always selecting the leftmost goal in each query, and by ordering the children of a node in the same way as the corresponding clauses in the program. A successful response by Prolog corresponds to a successful derivation in the Prolog derivation tree, the answer substitution being the composition of all substitutions made in that derivation.

Starting with a program P, if there exists a successful derivation of the query $?G$, with θ being the composition of all substitutions in the derivation, then we say that $G\theta$ is an *(unconditional) answer* to the query. In this case, the answer is a logical consequence of the program P, which we write as:

$$P \models \forall G\theta$$

where the universal quantification is over all variables in $G\theta$.

If, starting from the query $?G$, we have derived a non-empty query $?G_1, \ldots, G_n$, where θ is the composition of all substitutions, then the clause $G\theta \leftarrow G_1, \ldots, G_n$ is a *conditional answer* to the query. Again, as can be shown from the soundness of resolution, we have:

$$P \models \forall (G\theta \leftarrow G_1, \ldots, G_n)$$

where the universal quantification is over all variables in the clause $G\theta \leftarrow G_1, \ldots, G_n$.

Definition 1 (Partial derivation tree) *A partial derivation tree is a finite initial subtree in which non-empty leaf queries need not have a selected goal.*

In a partial derivation tree there are three types of leaves: empty queries, failed queries (these have a selected goal, but no children nodes) and non-empty queries

with no selected goal. For each empty query, there is a corresponding unconditional answer. Each non-empty query with no selected goal has a corresponding conditional answer.

For example, consider the following program, defining the reverse relation using the append relation:

```
% reverse( L, R ): R is the reverse of list L
% example: reverse( [1,2,3], [3,2,1] )
reverse( [], [] ).
reverse( [X|Xs], L ) :- reverse( Xs, R ), append( R, [X], L ).

% append( U, V, W ): list W is list V appended to list U.
% example: append( [1,2], [3,4,5], [1,2,3,4,5] )
append( [], L, L ).
append( [X|Xs], Y, [X|Z] ) :- append( Xs, Y, Z ).
```

A partial derivation tree for the query `?reverse(X,Y)` is shown in Figure 3.1 (the selected goals are underlined).

Definition 2 (Frontier) *The frontier of a partial derivation tree is the set of unconditional and conditional answers corresponding to the leaf nodes.*

The frontier for the partial derivation tree in Figure 3.1 consists of the clauses

```
reverse( [], [] ).
reverse( [X], L ) :- append( [], [X], L ).
reverse( [X,Y|Ys], L ) :-
   reverse( Ys, R ), append( R, [Y], R1 ), append( R1, [X], L ).
```

The frontier of a partial derivation tree consists of clauses, and thus we can regard it as a logic program. We would like to replace our original program by a frontier, but

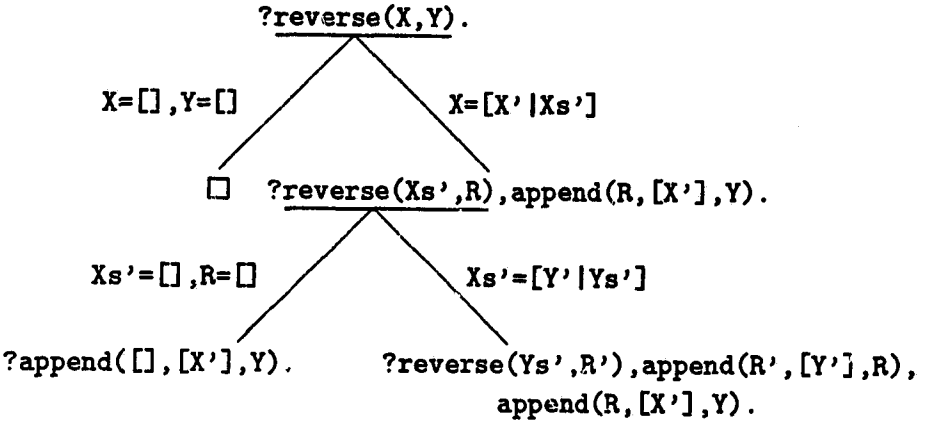


Figure 3.1: Partial derivation tree for reverse

if we do that, the resulting program will not be complete. To obtain completeness, we have to consider a suitable set of frontiers, giving rise to a logic program by including the clauses in all of the frontiers. We formalize this idea by defining a *complete set of frontiers.*

In the following discussion, $L(F)$ denotes the set of atoms appearing in the bodies of the clauses of a frontier F. Similarly, for a set of frontiers S, $L(S)$ denotes the union of the $L(F)$ for every frontier F in S. For a frontier F, $R(F)$ denotes the root of the partial derivation tree having F as a frontier. Finally, G_1 is an *instance* of G_2, if there exists a substitution θ such that $G_1 = G_2\theta$.

Definition 3 (Complete set of frontiers) *A set of frontiers S for a query Q is complete iff the following three conditions hold:*

1. *Each frontier F in S is non-trivial, that is, at least the root itself has a selected goal.*

2. *There is a frontier F in S such that the goal in the query Q is an instance of $R(F)$.*

3. *For each atom G in $L(S)$, there is a frontier F in S such that G is an instance of $R(F)$.*

The frontier for the partial derivation tree in Figure 3.1 is not complete by itself. The append atoms are not instances of the root of the partial derivation tree. We can obtain a complete set by adding a frontier for the query `?append( _, [_], _ )`, whose goal is the most general form of these atoms. We obtain the frontier

```
append( [], [X], [X] ).
append( [X|Xs], [Y], [X|Z] ) :- append( Xs, [Y], Z ).
```

by unfolding the root once. This is a version of the append program specialized for the purpose of the `reverse` program. The two frontiers together form a complete set of frontiers for `?reverse(X,Y)`.

We can find a complete set of frontiers for any program P and query Q. For example, if we construct a frontier for the most general form of every predicate symbol appearing in P and Q, then we have a complete set of frontiers (note that a frontier can be empty, which takes care of undefined calls in P). In fact, if we consider the frontiers in which we select the root and do not select atoms for any of the queries appearing at depth 1 in the partial derivation tree, then we obtain our original program as a complete set of frontiers (assuming the atom appearing in the query is defined in the program).

The complete sets of frontiers given above are not very useful. The following algorithm suggests a general way to obtain a complete set of frontiers S for a query Q and a program P.

1. Start with the frontier of any partial derivation tree for the query Q as the only element in S.

2. Select any atom, G, from $L(S)$ that is not an instance of $R(F)$ for any frontier F in S. Add the frontier of any partial derivation tree for G to the set, after removing any frontier F for which $R(F)$ is an instance of G (since the answers in such a frontier will also be included in the frontier of G).

3. If no such atom exists, then S is a complete set of frontiers for Q in P.

Although we can select any atom that is not an instance of a derivation tree already constructed in step 2, it is better to choose the *most general form* of all occurrences of atoms with the same predicate symbol.

3.2.3 The frontier theorem

The intuition behind a complete set of frontiers is that they are expanded versions of the original program specialized to the query. The following results formalize this idea.

We first prove a more general form of the lifting lemma ([39, page 47]).

Lemma 1 *Given a program P, a goal G and a substitution θ. If there exists a (possibly incomplete) derivation D in P starting from $G\theta$ and ending in $G_1, ..., G_m$ with mgu's $\theta_1, ..., \theta_n$ appearing along the way, such that none of the variables of the variants of the clauses used in the derivation appear in $G\theta$ or G (this is a stronger requirement than the usual standardizing apart [39, page 41]). Then:*

1. *There exists a derivation D' in P starting with G, in which the same goals are selected as in the corresponding nodes in D, and the same clauses from P*

are used to resolve these goals. If $\theta'_1, ..., \theta'_n$ are the mgu's along the way and $G'_1, ..., G'_{m'}$ are the final goals then $m = m'$ and there exists a substitution δ such that $\theta\theta_1...\theta_n = \theta'_1...\theta'_n\delta$ and $(G'_1...G'_m)\delta = G_1...G_m$.

2. *A variant of the (un)conditional answer $G\theta'_1...\theta'_n \leftarrow G'_1, ..., G'_m$ can be used in a one step resolution of $G\theta$, such that $G\theta\theta_1...\theta_n \leftarrow G_1...G_m$ is an instance of the resulting (un)conditional answer.*

Proof We will prove the first part by induction on the length of D.

Base case: length of D is 1. Then $G\theta$ unifies with the head of a clause $F \leftarrow F_1...F_n$ with mgu θ_1. Thus $F\theta_1 = G\theta\theta_1$ and so F and G unify, say with mgu θ'_1. Since θ'_1 is the mgu of F and G, there exists a substitution δ such that $\theta'_1\delta = \theta\theta_1$. Hence, $(F_1...F_n)\theta'_1\delta = (F_1...F_n)\theta\theta_1 = (F_1...F_n)\theta_1$ (we assume θ does not act on any variables of $F \leftarrow F_1, ..., F_n$).

Induction step: let D be a derivation of length $n+1$ starting from $G\theta$. Assume that after n steps the goals are $G_1, ..., G_m$, and that G_i is the selected goal which unifies with the head of the clause $F \leftarrow F_1, ..., F_k$. Let $\theta_1...\theta_{n+1}$ be the mgu's along the way.

By the induction hypothesis, there exists a derivation D' of length n with mgu's $\theta'_1...\theta'_n$ ending in $G'_1, ..., G'_m$. There also exists a substitution δ such that $\theta'_1...\theta'_n\delta = \theta\theta_1...\theta_n$ and $(G'_1...G'_m)\delta = G_1...G_m$. Since $G'_i\delta = G_i$, G'_i also unifies with the head of $F \leftarrow F_1, ..., F_k$, say with mgu θ'_{n+1}. We also have $G_i\theta_{n+1} = G'_i\delta\theta_{n+1} = F\theta_{n+1}$, so there exists a substitution δ' such that $\theta'_{n+1}\delta' = \delta\theta_{n+1}$, and hence $\theta'_1...\theta'_{n+1}\delta' = \theta\theta_1...\theta_{n+1}$. Since we can assume that δ does not act on any variables of $F \leftarrow F_1, ..., F_k$, we have $(G'_1...G'_{i-1}F_1...F_kG'_{i+1}...G'_m)\theta'_{n+1}\delta' = (G_1...G_{i-1}F_1...F_kG_{i+1}...G_m)\theta_{n+1}$.

For the second part, we note that $G\theta'_1...\theta'_n\delta = G\theta\theta_1...\theta_n$, which means that $G\theta$

and $G\theta'_1...\theta'_n$ unify. Let γ be the mgu.

Since γ is the mgu of $G\theta$ and $G\theta'_1...\theta'_n$, there must exist a substitution α such that $G\theta'_1...\theta'_n\gamma\alpha = G\theta'_1...\theta'_n\delta$, Now, $(G\theta \leftarrow G'_1...G'_m)\gamma = (G\theta'_1...\theta'_n \leftarrow G'_1...G'_m)\gamma$, and $G\theta\theta_1...\theta_n \leftarrow G_1...G_m = (G\theta'_1...\theta'_n \leftarrow G'_1...G'_m)\delta$. From which it follows that $(G\theta \leftarrow G'_1...G'_m)\gamma\alpha = G\theta\theta_1...\theta_n \leftarrow G_1...G_m$, since we can assume that γ and δ do not act on any variables that appear in a body but not the corresponding head of the two clauses. ■

Using this lemma, we prove that if there is a derivation for a goal in a program, then there is a corresponding derivation for that goal in a complete set of frontiers, as long as that goal is an instance of one of the roots of the partial derivation trees.

Lemma 2 *Let S be a complete set of frontiers for a query Q and a program P. If the goal G is an instance of $R(F)$ for a frontier F in S, and if there is a successful derivation D of G in P, then there is a successful derivation D' of G in S. Moreover, if θ_1 and θ_2 are the compositions of all substitutions in D and D' respectively, then $G\theta_1$ is an instance of $G\theta_2$.*

Proof The proof will be by induction on the length of D. Let θ be the substitution such that $G = R(F)\theta$, and let T be the partial derivation tree corresponding to F.

We can use the switching lemma [39, page 50] repeatedly to obtain a successful derivation D'' of G in P in which the goals are selected in the same way as the corresponding path in T. By the previous lemma, which we can use since $G = R(F)\theta$, the corresponding path must exist in the frontier and we either obtain an empty query within the frontier, or we cross it with some remaining goals. For the queries beyond the frontier of T we always select the leftmost goal in D''. Then D'' is of the same

length as D, and if θ_3 is the composition of all substitutions in D'', then $G\theta_3$ and $G\theta_1$ are variants.

There are two cases to consider:

1. We reach success within the frontier, which includes the base case where the length of D is 1. Since G is $R(F)\theta$, we can use the previous lemma and conclude that F, and hence S, contains an unconditional answer of the form $R(F)\theta'$, which unifies with G. Thus we can use this unconditional answer to obtain a successful derivation of G in S. If θ_2 is the mgu of G and $R(F)\theta'$ then we can also use the lemma to conclude that $G\theta_3$, and hence also $G\theta_1$, is an instance of $G\theta_2$.

2. We cross the frontier at a point with the remaining subgoals $G'_1, ..., G'_n$. By the previous lemma, the corresponding query in D'' contains the goals $G_1, ..., G_n$ and there exists a substitution δ so that $G_1...G_n = (G'_1...G'_n)\delta$. This means that the frontier contains the conditional answer $R(F)\theta' \leftarrow G'_1, ..., G'_n$, which can be used to resolve G in S to obtain the goals $(G'_1, ..., G'_n)\gamma$, where γ is the mgu of G and $R(F)\theta'$ (without loss of generality, we can assume we need not use a variant here). Moreover, $G\alpha \leftarrow G_1...G_n$ is an instance of $(G \leftarrow G'_1...G'_n)\gamma$, where α is the composition of the substitutions in D'' before $G_1...G_n$ is reached.

   We always select the leftmost goal in D'' from $G_1...G_n$ on. Hence, it must contain a successful derivation D_1 of G_1, say with α_1 being the composition of the substitutions. Now G_1 is an instance of $G'_1\gamma$, and G'_1 appears in the frontier of S, so G'_1, and hence G_1, must be the instance of a root of a frontier of S. Since D_1 is also shorter than D, we can use the induction hypothesis to conclude that there is a successful derivation of G_1 in S. But G_1 is an instance of $G'_1\gamma$ and so there is a derivation D'_1 of $G'_1\gamma$ in S. If γ_1 is the composition of

the substitutions in D'_1, then $G_1\alpha_1$ is an instance of $G'_1\gamma\gamma_1$.

Similarly, D'' must contain a successful derivation D_i of $G_i\alpha_1...\alpha_{i-1}$, with α_i being the composition of the substitutions. By the induction hypothesis, there must exist derivations D'_i of $G'_i\gamma\gamma_1...\gamma_{i-1}$ in S such that $G_i\alpha_1...\alpha_i$ is an instance of $G'_i\gamma\gamma_1...\gamma_i$, where γ_i is the composition of the substitutions in D'_i.

We can thus obtain a successful derivation of G in S by first using the rule $R(F)\gamma' \leftarrow G'_1...G'_n$, and then using the derivations $D'_1, ..., D'_n$, and $G\alpha\alpha_1...\alpha_n$ is an instance of $G\gamma\gamma_1...\gamma_n$. ■

We can now prove the frontier theorem, which shows that a complete set of frontiers for a particular query preserves the meaning of the program with respect to that query.

Theorem 1 (Frontier Theorem) *If P is a program, Q a query, and S a complete set of frontiers for Q, then*

$$[Q] \cap \textit{success set of } P = [Q] \cap \textit{success set of } S$$

where $[Q]$ is the set of all ground instances of Q.

Proof By the soundness of SLD-resolution we know that

$$\text{success set of } P \supseteq \text{success set of } S$$

and hence we also have

$$[Q] \cap \text{success set of } P \supseteq [Q] \cap \text{success set of } S.$$

Conversely, if $G \in [Q] \cap$ success set of P, then $G \in [Q]$ and by the definition of a complete set of frontiers we know that G is an instance of $R(F)$ for a frontier F in S. Thus we can apply the previous lemma (G is a ground goal, so the substitutions that take place during the derivation do not concern us here) to conclude that $G \in$ success set of S, and hence

$$[Q] \cap \text{success set of } P \subseteq [Q] \cap \text{success set of } S$$

and the proof is completed. ■

Note that the frontier theorem is valid for frontiers of *any* derivation tree, not just the Prolog derivation tree. In this way, selection methods other than Prolog's can be "compiled into" the frontier.

3.3 The transformation rules

We now describe the transformation rules introduced by Burstall and Darlington [3]. Tamaki and Sato proved that these, as well as other transformation rules useful for logic programs, preserve the soundness and completeness of logic programs [59, 60]. We discuss each of the rules, and give the soundness and completeness properties for them. We show how these rules can be used when building a frontier, and give an example of a transformation using these rules.

3.3.1 Unfolding

So far we have only used unfolding to unfold a goal in a (partial) derivation tree. Tamaki and Sato define unfolding in terms of unfolding an atom in the body of a clause [59]. Let G be an atom in the body of a clause C and let $C_1, ..., C_n$ be the

clauses of the program of which the heads unify with G. That is, each C_i, $i = 1, ..., n$, resolves with G, say, with resolvent C_i'. Then replacing C by $C_1', ..., C_n'$ is said to be the unfolding of G in C.

For example, if we unfold the **reverse** atom in the body of the second clause in the program

```
reverse( [], [] ).
reverse( [X|Xs], L ) :- reverse( Xs, R ), append( R, [X], L ).
```

(we have only shown the clauses defining the **reverse** relation here) we obtain the clauses

```
reverse( [], [] ).
reverse( [X], L ) :- append( [], [X], L ).
reverse( [X,Y|Ys], L ) :-
   reverse( Ys, R ), append( R, [Y], R1 ), append( R1, [X], L ).
```

These clauses are the same as the frontier for the partial derivation tree shown in Figure 3.1. This is because unfolding a goal in a partial derivation tree is the same as unfolding the corresponding atom in the frontier of that tree[2].

3.3.2 Defining new predicates

Certain transformations require the definition of new predicates in terms of already existing ones. For example, the reverse program shown above requires $O(n^2)$ calls to **append** to reverse a list of length n. By defining the relation **reverse1** through the clause

[2]A special case of this is unfolding the root of a partial derivation tree. Suppose G is the goal appearing in the root, then unfolding this goal corresponds to unfolding G in the body of the clause $G \leftarrow G$.

```
reverse1( L, R, A ) :- reverse( L, R1 ), append( R1, A, R ).
```

we can transform the original program so that it reverses a list of length n in $O(n)$ calls. The transformation is shown at the end of this section.

In defining a new predicate we not only change the Herbrand base of the program, but we most likely also change the minimal Herbrand model (i.e., the meaning of the program). However, as the clauses of the original program do not contain this newly defined predicate, we cannot change the definitions of the predicates appearing in the original program. Therefore we have a weak form of equivalence: the meaning of the program, restricted to the original predicates, has not changed.

The system developed by Burstall and Darlington finds definitions of new predicates that can be useful [8]. This is hard to do in general, but for certain transformation techniques it is possible. One such technique is the transformation to tail-recursive form [10, 50].

3.3.3 Folding

Folding is the inverse operation of unfolding; we replace part of a clause corresponding to the body of another clause by the head of that clause. Let C be a clause of the form $A \leftarrow B_1, \ldots, B_n$ in program P. If D is a clause in P with an instance of the atoms $B_1, \ldots, B_n$ appearing (in any order) in its body, then the result of folding D using C is the clause D', obtained from D by replacing the instances of $B_1, \ldots, B_n$ by the corresponding instance of A.

For example, if the program contains the clauses

```
reverse1( L, R, A ) :- reverse( L, R1 ), append( R1, A, R ).
```

```
reverse( [], [] ).
reverse( [X|Xs], L ) :- reverse( Xs, R ), append( R, [X], L ).
```

then we can fold the last clause using the first one, and the resulting clause is

```
reverse( [X|Xs], L ) :- reverse1( Xs, L, [X] ).
```

Clark [5] and Hogger [31] point out that folding is sound in certain cases because definitions of relations are in fact equivalences rather than one way conditionals. Tamaki and Sato proved the following result, showing that independent of the meaning of the relations being defined, the folding rule preserves the meaning of a program [59]. The *definition set* of a program P is a finite set of definitions of new predicates in terms of predicates in P.

Theorem 2 *Given a program P and clauses C and D in P, such that C is of the form $A \leftarrow B_1, \ldots, B_n$, and is in the definition set of P, and such that D has an instance of the atoms $B_1, \ldots, B_n$ appearing in its body. Then the program P', obtained from P by replacing D by the result of folding D using C, is equivalent to P if the following conditions hold:*

1. *The instance is such that the variables appearing in $B_1, \ldots, B_n$ and not in A (which are referred to as internal variables of the clause) are all mapped to distinct variables, and none of these occur elsewhere in the body or the head of D.*

2. *There are more atoms than $B_1, \ldots, B_n$ in the body of D, or D is not in the definition set.*

3.3.4 Using the functionality of a relation

If the body of a clause contains the two atoms $p(T_1, \ldots, T_n, X)$ and $p(T_1, \ldots, T_n, Y)$, and p is a function (i.e., for all ground terms $t_1, \ldots, t_n$ there is a unique value of t such that $p(t_1, \ldots, t_n, t)$ is a logical consequence of the program), then we can replace that clause by one in which we replace the two atoms by either one of them after unifying X and Y.

For example, suppose that during a transformation we obtain a clause with both the atoms `reverse( L, R1 )` and `reverse( L, R2 )` in its body. We can replace these atoms in this clause by the single atom `reverse( L, R1 )` after unifying `R1` and `R2`. This is because the reverse of a list is unique.

This transformation preserves the meaning of a program [59, 60]. The fact that a relation is a function is a property of the program. Clark suggests how we can use the clauses of the program to prove such properties [5].

3.3.5 Using laws

Other uses of properties of relations are referred to as the use of laws. Common examples are the associativity of the append relation, and the commutativity and associativity of both the sum and product relations.

For example, using the associativity of append, we can replace the clause

```
reverse1( [A|B], C, D ) :-
   reverse( B, E ), append( E, [A], F ), append( F, D, C ).
```

by the clause

```
reverse1( [A|B], C, D ) :-
   reverse( B, E ), append( [A], D, T )  append( E, T, C ).
```

Both Vasey [63] and Tamaki and Sato [60] point out that under certain conditions we can use these laws without changing the meaning of a program. One of the conditions is that we cannot "lose" the meaning of any variables. For example, we can replace the atoms `append(A,B,T)` and `append(T,C,R)`, by the atoms `append(B,C,S)` and `append(A,S,R)` as long as `T` does not appear elsewhere in the clause.

3.3.6 Rearranging clauses and atoms

The meaning of a program depends only on the existence of a successful derivation. As a result, we cannot change the meaning of a program by rearranging its clauses or the atoms inside the clauses. While this is used implicitly in most transformation systems, it is mentioned as a transformation rule only in [52].

When we execute a logic program as a Prolog program, the order of both the clauses and the atoms inside them becomes important. This is because the Prolog interpreter always selects the leftmost goal in a query, and uses the clauses in the same order as they appear in the program. This means that rearranging either the clauses or the atoms inside them can have a big influence on the efficiency of a program. And indeed, sometimes it can result in overflow instead of success. If built-in predicates are used, rearranging the atoms can result in an execution error.

One of the transformation techniques, namely transforming recursive definitions to iterative ones, is based on the idea of rearranging the atoms so that the recursive call is the last one in the body (i.e., obtaining a tail recursive definition).

3.3.7 Applying the transformation rules to frontiers

So far, we have only used unfolding when constructing frontiers. When we want to apply some of the other transformations discussed in this section, we could stop the construction of the frontier, thus obtaining a complete set of frontiers equivalent to the original program. We could then apply the transformation rule to the complete set of frontiers using Tamaki and Sato's result to guarantee equivalence. Then in another iteration we could generate a set of frontiers that is complete and nonredundant with respect to the query of interest.

Since a frontier is nothing more than a set of clauses, it would be more natural to apply the other transformations as soon as applicable, that is, to include them into the algorithm for generating complete sets of frontiers. This is done in the program transformation system that will be discussed in Section 3.4.

3.3.8 An example transformation

We show how the transformations discussed above can be used to transform the program

```
% reverse( L, R ): R is the reverse of list L
% example: reverse( [1,2,3], [3,2,1] )
reverse( [], [] ).
reverse( [X|Xs], L ) :- reverse( Xs, R ), append( R, [X], L ).

% append( U, V, W ): list W is list V appended to list U.
% example: append( [1,2], [3,4,5], [1,2,3,4,5] )
append( [], L, L ).
append( [X|Xs], Y, [X|Z] ) :- append( Xs, Y, Z ).
```

to reverse a list into a more efficient version. The above program requires $O(n^2)$ calls

to append to reverse a list of length n. We will transform it so that only $O(n)$ calls are needed.

To start the transformation, we need to define a new predicate, **reverse1**, through the clause

```
reverse1( L, R, A ) :- reverse( L, R1 ), append( R1, A, R ).
```

This definition can be derived using the method proposed by Debray [10] for transforming programs to tail-recursive form. We first construct a frontier for the query **?reverse(X,Y)**. By unfolding the root once, we obtain a frontier that consists of the same clauses as the original definition for **reverse**. We can now fold the second clause of this definition using the clause defining **reverse1**. This gives us a frontier with the clauses

```
reverse( [], [] ).
reverse( [X|Xs], L ) :- reverse1( Xs, L, [X] ).
```

This frontier does not form a complete set of frontiers for the query **?reverse(X,Y)**, as the **reverse1** atom is not an instance of the root.

We continue by constructing a frontier for the most general version of an atom with **reverse1** as the predicate symbol, namely **reverse1(X,Y,Z)**. After unfolding the **reverse** atom in the clause defining **reverse1**, we obtain the frontier

```
reverse1( [], A, B ) :- append( [], B, A ).
reverse1( [A|B], C, D ) :-
   reverse( B, E ), append( E, [A], F ), append( F, D, C ).
```

Using the associativity of append in the second clause, we get the frontier

```
reverse1( [], A, B ) :- append( [], B, A ).
reverse1( [A|B], C, D ) :-
   reverse( B, E ), append( [A], D, T ), append( E, T, C ).
```

If we then unfold the **append** call in the first clause once, and the first **append** atom in the second clause twice, we obtain the frontier

```
reverse1( [], A, A ).
reverse1( [A|B], C, D ) :- reverse( B, E ), append( E, [A|D], C ).
```

Finally, we can fold the definition of **reverse1** in the second clause, and we end up with the frontier

```
reverse1( [], A, A ).
reverse1( [A|B], C, D ) :- reverse1( B, C, [A|D] ).
```

The two frontiers

```
reverse( [], [] ).
reverse( [X|Xs], L ) :- reverse1( Xs, L, [X] ).

reverse1( [], A, A ).
reverse1( [A|B], C, D ) :- reverse1( B, C, [A|D] ).
```

form a complete set of frontiers for the query `?reverse(X,Y)`. If we use the results of Tamaki and Sato, we know that these two frontiers are equivalent to the original program, provided that we include the definition of the **append** relation. We can use the frontier theorem to show that we do not need this relation if we are only interested in solving queries about the **reverse** relation.

3.4 Frontier transformation system

The frontier transformation system, FROST, is based on the transformation system described in [57]. The original system assists the user in building partial derivation trees and collects the answers in the frontier. This remains the same in FROST, but whereas the original system traversed the partial derivation tree depth-first, left to right, FROST allows the user to expand any part of the derivation tree. Another change is that FROST incorporates the notion of a complete set of frontiers.

3.4.1 Overview of FROST

It is very tedious to perform the transformations discussed previously by hand. To assist the user with building complete sets of frontiers we constructed FROST. Under the guidance of the user, FROST builds a frontier for a specific query. The user determines interactively what transformation rule to apply; the system then applies the rule and records the resulting frontier.

FROST consists of two levels. At the top level, the system maintains the program that is being transformed and a set of frontiers. The user can add and delete frontiers to and from this set, and ask the system if a complete set of frontiers exists for a particular query. At the second level, the user can apply transformation rules to a particular frontier (the *current frontier*) from the set of frontiers.

Unfolding, folding, and functionality are the only rules that FROST can automatically apply. All other rules, including laws, have to be defined to the system as "meta-rules." These meta-rules are rewrite rules on the bodies of the clauses in a frontier, and are discussed in Section 3.4.3.

When we build a complete set of frontiers using only unfolding, we know by the

frontier theorem that we preserve the meaning of the program. Since the user can apply other transformation rules when building a frontier using FROST, it is the user's responsibility to ensure that these also preserve the meaning of the program. Tamaki and Sato's results can be used for this purpose.

FROST can deal with all Pure Prolog programs. Depending upon the type of built-in predicates, it can also deal with some programs containing built-in predicates. Since there are no clauses in the program for the built-in predicates, goals with built-in predicates cannot be unfolded. However, certain goals, such as `X is Y + 0`, can be simplified by either "proving" these goals or by using meta-rules as will be explained in Section 3.4.3.

As was pointed out in Section 3.3.6, rearranging the goals in a clause or the clauses in a program does not change the meaning of the program. However, we can get significantly different behaviour when we execute these programs as Prolog programs. Although FROST maintains the order of the goals and clauses as much as possible, it is hard to determine what the correct order is for certain transformations (e.g., where should the head of a clause that is being folded go?). This means that the user might have to reorder the goals and clauses to make the resulting program behave as desired.

In Appendix A we show how FROST can be used to perform the transformation on the **reverse** relation shown in Section 3.3.8. FROST has been used to apply several program transformation techniques discussed in the literature, such as: loop combination [3, 50], transforming programs to tail recursive form [3, 5, 10], obtaining difference list versions of programs with append in it [24, 63], and the combination of meta and object-level code [58].

3.4.2 Program and frontier manipulation

FROST uses the module system of ALS Prolog by maintaining the program that is being transformed in the **user** module. This module is automatically created when ALS Prolog starts up and is also the default module. Thus the user can manipulate and execute the program and the clauses in it from ALS Prolog without entering FROST. FROST itself, as well as the frontiers it maintains, resides in other modules so that it does not interfere with the program that is being transformed. From within FROST, the user has commands to:

- Load a new program from a file.
- Add clauses from a file to the program.
- Display the clauses that are currently in the program.

At all times FROST maintains a set of frontiers and their associated roots. Initially this set is empty. There are commands to:

- Add a frontier for a new goal. This can only be done if there does not already exist a frontier with the same goal as its root. After adding the frontier, it becomes the current frontier and the user can apply transformations to it.
- Delete a frontier from the set.
- Start with a new set of frontiers, i.e., delete all existing frontiers.
- Display the clauses in a particular frontier.
- Save the clauses in a frontier to a file, or save the clauses of all the frontiers to a file.

- Show the roots of all the frontiers.
- Make any frontier the current one, and allow the user to apply transformations to it.
- Ask if a complete set of frontiers exists for a particular query. If so, FROST shows the frontiers that are needed to form a complete set of frontiers. If not, FROST shows how a complete set can be formed by giving frontiers that need to be included and the goals for which frontiers will have to be added. In both cases the choice of frontiers and goals is non-deterministic, in which case FROST chooses the first alternative it finds.

3.4.3 Applying transformation rules to a frontier

When transforming a particular frontier, FROST shows the clauses in the current frontier. The user can apply one of several transformation rules, or return to the top level. The transformations that can be applied are:

- Unfold a goal in a clause of the frontier.
- Prove a goal in a clause of the frontier. FROST finds all instances of this goal that it can prove in the **user** module, and unfolds this goal by treating all these instances as facts about the goal. This transformation rule can be used to combine several unfolding steps and to deal with built-in predicates.

  For example, suppose a clause contains the goal `append( [1,2], X, Y )`, and the program contains the standard definition for **append**. Proving this goal would eliminate it from the clause while unifying `Y` with `[1,2|X]`. This same step would require three applications of unfolding.

If a clause contains the goal `X is 3 + 2`, proving this goal would eliminate it while unifying `X` with `5`. In this case unfolding cannot be used since `is` happens to be a built-in predicate.

- Fold the definition of a predicate. In this case, FROST tries to fold any of the clauses in the frontier using a clause that defines this predicate in the program. It does not check if the conditions given in Theorem 2 are satisfied. It is the user's responsibility to check this.

- Use the functionality of a predicate in a clause of the frontier. To perform this transformation, the user must give the name and arity of the predicate and the argument corresponding to the return-value of the function. For example, if a clause contains the goals `reverse( [A,B], R1 )` and `reverse( [A,B], R2 )`, then FROST can use the functionality of the second argument of the `reverse` predicate.

- Apply a meta-rule to a clause. A meta-rule is a rewrite rule on the body of a clause. It consists of four parts:

  1. A unique identifier.
  2. A list of atoms that has to be replaced.
  3. A list of atoms that replaces the first list.
  4. A list of conditions (atoms) that has to be satisfied before the meta-rule can be applied. These can be used to express preconditions for the application of the meta-rule and to obtain side-effects as shown in the examples below.

  FROST applies the meta-rule by replacing goals appearing in the body of a clause that are an instance of the first list by the corresponding instance of the second list after executing the conditions in the third list. At any point in the

system the user can add new meta-rules, delete existing rules, or look at the existing ones. The meta-rules can be used to incorporate the use of laws in a transformation, and to simplify certain built-in goals.

For example, the meta-rule that replaces the list of goals

```
[append(A,B,T1),append(T1,C,ABC)]
```

by the list of goals

```
[append(B,C,T2),append(A,T2,ABC)]
```

expresses the associativity of append. This rule should only be applied if `T1` does not occur elsewhere in the clause; we assume the user checks this.

Suppose a body in a clause of the frontier contains the atom `X is Y + 0`. Proving this goal will result in an execution error because not enough variables are instantiated. We would like to simplify the clause by eliminating this goal wh. unifying `X` and `Y`. We can perform this simplification by adding a meta-rule with `[X is Y + 0]` as its first list, the empty list as its second list (thus eliminating the goal from the body of the clause), and `[X = Y]` as its list of conditions (which unifies `X` and `Y`).

Chapter 4

Transforming SCS Specifications

The SCS specifications presented in Chapter 2 can be executed as Prolog programs. The main concern when writing these specifications, however, is clarity and not efficiency. As a result, they are often inefficient when executed as Prolog programs. In Chapter 3 we showed how logic-based program transformation can make logic programs more efficient while preserving their meaning. In this chapter we examine how we can use program transformation to optimize SCS specifications so that they become efficient enough to serve as implementations.

In the SCS method the state of the module is represented by result-terms. While this allows the specifications to be fully abstract, it is a source of inefficiency when executing these specifications as Prolog programs. In the first section we show how we can obtain a more efficient implementation by replacing the result-terms by more appropriate terms to represent the state of the module. This transformation from abstract to concrete data objects is also used in VDM [34], where it is called *data reification.* Its use in logic programming is discussed in [38].

When writing SCS specifications, clarity of the specification is the most important

issue. This means that specifications contain definitions of relations that are obviously correct but possibly very inefficient. In the second section we show how some of these definitions can be transformed into more efficient algorithms. This *algorithm reification* is also discussed in [38]. It is similar to the *operation decomposition* of VDM [34], although in our case the language of the specification and the language of the implementation are the same.

4.1 Data reification

We use an example to show how we can apply data reification to SCS specifications by replacing the fully abstract result-terms by a more appropriate representation for the state of the module. We discuss the problems we encountered when using this method on several modules.

4.1.1 The counter module

To illustrate the ideas behind the transformation and some of the problems we encountered, we use a simple example. Figure 4.1 shows an SCS specification of an integer counter module. The access program `s_init` initializes the counter to `0`. There is an access program to increment the counter, and one to check its value. When an attempt is made to increment the counter above `100`, the `overflow` exception is raised.

After counting to `100`, the result-term that represents the state of the module consists of `s_init` followed by `100` calls to `s_inc`. Moreover, the second clause defining the `holds` relation has to be used `100` times to get the value of the counter. If we represented the state of a module by an integer rather than by result-terms, the

```
%
% INTRODUCTION
%
%    The counter module provides an integer counter.
%
% SYNTAX
%
%   s_init.
%   s_inc.              %overflow%
%   g_val -> integer.

%
% EFFECTS
%
% Initially:
holds( g_val(0), [s_init] ).
% After s_inc:
holds( g_val(N1), [s_inc|S] ) :- holds( g_val(N), S ), N1 is N+1.

%
% EXCEPTIONS
%
% overflow
exception( overflow, s_inc, S ) :- holds( g_val(100), S ).
```

Figure 4.1: SCS specification of counter module

amount of storage needed and the time needed to "calculate" the value of the counter is much less and remains constant when the value of the counter increases.

4.1.2 Implementation of the counter module

If we use an integer to represent the state of the module, we can redefine the `holds` and `exception` relations using the clauses

```
holds( g_val(N), N ).
exception( overflow, s_inc, 100 ).
```

The first clause expresses that the value of the counter is the same as the value of the state. The second one expresses that the exception `overflow` is raised for the call `s_inc` when the state of the module has the value `100`.

These clauses by themselves do not give us an implementation of the counter module. For example, we cannot use them together with the interface shown in Figure 2.8. This is because the above clauses do not describe what the initial state of the module is, and how the set-calls change the state of the module. For SCS specifications, the initial state of the module is the result-term consisting of the call to initialize the module, and if `T` is the result-term representing the state before set-call `C`, then `[C|T]` is the result-term representing the state after that call.

We can obtain an implementation by defining the initial state of the module and stating how set-calls affect the state of the module. We use the relation `initstate` to define the initial state of the module given the call to initialize the module. The `newstate` relation is used to define how each set-call affects the state of the module. For the counter module this can be done using the clauses

```
% state( I ) :- integer( I ).
%-------------
initstate'( s_init, 0 ).
newstate'( s_inc, N, N1 ) :- N1 is N+1.
%-------------
holds'( g_val(N), N ).
%-------------
exception'( overflow, s_inc, 100 ).
```

Figure 4.2: Implementation of counter module

```
initstate( s_init, 0 ).
newstate( s_inc, N, N1 ) :- N1 is N+1.
```

The initial state is 0, and `s_inc` increments the value of the state by 1.

We can now use the interface shown in Figure 2.8 to execute the above implementation. We assume the clauses

```
initstate( Call, [Call] ).
newstate( Call, State, [Call|State] ).
```

from Figure 2.8 have been eliminated, since the definition of the `initstate` and `newstate` relations are part of the implementation.

4.1.3 From result-terms to concrete states

Our goal is to derive the implementation shown in Figure 4.2 starting from the specification shown in Figure 4.1. We have added a quote to the predicate names to avoid naming conflicts. In comments, we add the definition of the unary `state` predicate, which succeeds for all syntactically correct states.

Such a transformation from one data structure to another is discussed in [24], and we use the same approach here. We first define a mapping function `m`, which shows how the old data structure maps to the new one. In our case, it defines the mapping from result-terms to the representation of the state we choose. For example, for the counter module the clauses

```
m( [s_init], 0 ).
m( [s_inc|T], N1 ) :- m( T, N ), N1 is N+1.
```

define the mapping function as a binary relation. Clearly, the choice of `m` greatly influences the efficiency of the implementation we can derive from it, and for certain choices of `m` we cannot derive an implementation at all. However, for all the modules we looked at, it was easy to find an appropriate `m`.

The second step is to define the new relations in terms of the old ones and the mapping relation through the clauses:

```
initstate'( C, S ) :- initstate( C, T ), m( T, S ).
newstate'( C, S, S1 ) :-
   newstate( C, T, T1 ), m( T, S ), m( T1, S1 ).

holds'( C, S ) :- holds( C, T ), m( T, S ).

exception'( E, C, S ) :- exception( E, C, T ), m( T, S ).
```

The new relations define the same relations as the old ones, except that they use the new representation, rather than result-terms, for the state of the module. For example, the first clause expresses the fact that `S` is the new representation of the initial state given initialization call `C`, if `T` is the result-term representing the initial state for the same call, and `m` maps `T` to `M`. Using the clauses

```
initstate'( C, S ) :- m( [C], S ).
newstate'( C, S, S1 ) :- m( T, S ), m( [C|T], S1 ).

holds'( C, S ) :- holds( C, T ), m( T, S ).

exception'( E, C, S ) :- exception( E, C, T ), m( T, S ).
```

Figure 4.3: Definition of the new predicates for the data-structure mapping

```
initstate( Call, [Call] ).
newstate( Call, State, [Call|State] ).
```

we can simplify the first two clauses so that we obtain the definitions shown in Figure 4.3.

We want to transform the clauses in Figure 4.3 (which remain the same for all modules) and the specification from Figure 4.1 into the implementation shown in Figure 4.2. This is the order in which we would like to do things. However, in practice, we always wrote the implementation first, and then attempted to show it could be obtained by transforming the specification. We discuss how to perform the transformation for each of the relations in this example.

The `initstate'` and `newstate'` relations. Using unfolding and the functionality of the mapping `m` we obtain the clauses

```
initstate'( s_init, 0 ).
newstate'( s_inc, A, B ) :- m( _, A ), B is A+1.
```

The first of these clauses is what we want, but the second one contains the additional condition `m( _, A )` in its body.

This condition restricts the **newstate'** relation so that it only holds when the second argument is a state corresponding to a result-term (this condition is also contained in the original definition of **newstate'**). There is no need to add this condition to the implementation in Figure 4.2, since we assume that the second argument is a valid state when **newstate'** is called. Thus although we cannot derive the desired result directly, we can derive a clause with an additional condition in the body which can be omitted if we assume the **newstate'** predicate is called with the second argument instantiated to a valid state.

The holds' relation. After unfolding several times, and folding the definition of the **holds'** relation, we derive the clauses

```
holds'( g_cnt(0), 0 ).
holds'( g_cnt(A), B ) :- holds'( g_cnt(C), D ), A is C+1, B is D+1.
```

We want to derive the clause

```
holds'( g_cnt(N), N ).
```

which states that the relation holds for any value of N. However, the first two clauses can only be satisfied if the second argument is an integer, i.e., represents a valid state. This problem is similar to the problem with the **newstate'** relation, where we can only derive the desired result if we add a condition to the body of the clause. We can do the same thing here, and thus we would like to show that the original two clauses are equivalent to

```
holds'( g_cnt(N), N ) :- m( _, N ).
```

We can transform this last clause using unfolding and folding and we obtain the clauses

```
holds'( g_cnt(0), 0 ).
holds'( g_cnt(A), A ) :- holds'( g_cnt(B), B ), A is B+1.
```

As shown in Appendix B, using the method discussed in [16], these clauses have the same minimal model as the above two clauses. Therefore we can conclude that the clauses

```
holds'( g_cnt(0), 0 ).
holds'( g_cnt(A), B ) :- holds'( g_cnt(C), D ), A is C+1, B is D+1.
```

have the same meaning as the clause

```
holds'( g_cnt(N), N ) :- m( _, N ).
```

Unfortunately, in order to use such a method we have to know the clause we want to derive in advance. Thus we are no longer transforming the specification, but we are verifying that a certain implementation is equivalent to the specification.

The `exception'` relation. After unfolding and folding the definition of the `holds'` relation from Figure 4.3, we obtain the clause

```
exception'( overflow, s_inc, A ) :- holds'( g_cnt(100), A ).
```

We can unfold the `holds'` condition using the new definition for `holds'` and we get the clause

```
exception'( overflow, s_inc, 100 ).
```

which is the desired result.

4.1.4 Problems with the transformation

The example above shows that for a simple module we can derive the implementation from the specification and the mapping relation, although we need more powerful theorem-proving techniques than the unfold/fold transformation rules. We tried using the method on several other modules with various degrees of success. The modules we used are: character counter, stack, queue, traversing stack (a stack in which all elements can be accessed, although they can only be added and deleted from the top), tree, and a lexical scanner. We discuss the problems we encountered for each of the relations.

The `initstate'` and `newstate'` relations. It was straightforward to transform the `initstate'` relation. Depending upon how the mapping relation `m` was defined (as explained below), the transformation for the `newstate'` relation was either straightforward or caused some problems. However, in each case the derived definition of the `newstate'` relation contained an additional condition in the body expressing the fact that the second argument should be a valid state, which was dropped in the actual implementation.

There are two different ways of defining the mapping relation `m` for specifications with state equivalences, such as the stack module discussed in Section 2.5. One way is to first define the mapping for the result-terms in canonical form, and then to use the equality relation to define the mapping for all result-terms.

Consider the stack specification shown in Figure 2.7. An appropriate representation for the state of the module is a pair consisting of the depth of the stack and a list representing the current contents of the stack. The mapping relation can thus be defined by the clauses

```
m( T, S ) :- equal( T, NT ), m1( NT, S ).

m1( [s_init], (0,[]) ).
m1( [s_push(I)|T], (N1,[I|S]) ) :- m1( T, (N,S) ), N1 is N + 1.
```

The first clause defines **m** for all result-terms using the equality relation, and the last two clauses define the mapping for canonical result-terms (ones consisting of a number of calls to **s_push** followed by a single call to **s_init**). Using this definition we were unable to derive the desired clauses

```
newstate'( s_push(I), (N,L), (N1,[I|L]) ) :- N1 is N + 1.
newstate'( s_pop, (N,[_|L]), (N1,L) ) :- N1 is N - 1.
```

for the **newstate'** relation.

Since result-terms are equivalent when they represent the same abstract state, it seems that we should always be able to define **m** only for canonical result-terms. However, it can be advantageous for efficiency reasons to have more than one concrete representation for the same abstract state. For example, for a module implementing a set, we typically would have a different representation of the same set when elements are added in a different order.

In this case we define **m** for all result-terms without using the equality relation. For the stack example, this can be done using the clauses

```
m( [s_init], (0,[]) ).
m( [s_push(I)|T], (N1,[I|S]) ) :- m( T, (N,S) ), N1 is N + 1.
m( [s_pop|T], (N1,S) ) :- m( T, (N,[I|S]) ), N1 is N - 1.
```

With this definition we can transform the definition of the **newstate'** relation to

```
newstate'( s_push(I), (N,L), (N1,[I|L]) ) :-
   m( _, (N,L) ), N1 is N + 1.
newstate'( s_pop, (N,[I|L]), (N1,L) ) :-
   m( _, (N,[I|L]) ), N1 is N - 1.
```

which is what we want.

When using the first method to define the mapping relation, we were unable to derive the desired definition for the **newstate'** relation. However, when using a method proposed by Hoffman [28] to organize the specifications based on a canonical form and a single-call extension we were able to derive the desired results for some of the calls. When defining the mapping relation for all result-terms, it was easy to derive the desired definition for the **newstate'** relation. This is no surprise, since in this case the definition of the mapping relation already expresses how each set-call affects the state.

The holds' relation. The counter module shown above was the only module for which we formally proved that the desired result was equivalent to the original definition. For other modules we could transform the definition far enough so that, using an informal argument, we could conclude that the resulting clauses were equivalent to the desired implementation. For some of the modules, however, we were not able transform the definition so that it even seemed possible to show that the desired implementation was equivalent to the resulting clauses.

It could be that using a theorem-proving system such as discussed in [16] we could verify that some of the implementations have the same meaning as the original definitions. However, this means that we would have to know the desired implementation. In the cases of the modules above this was the case, however, for more complicated modules this might not be true.

The `exception'` relation. When the exceptions can be defined in terms of the `holds` relation (as is the case for the counter module above, and the character counter and stack modules shown in Chapter 2) the transformation of the `exception'` relation is straightforward. For some of the exceptions in the other modules this was not the case. For these we could still derive the desired results for some exceptions, but for others we were not able to do so.

4.2 Algorithm reification

After changing the representation for the state, the resulting program can still contain inefficient algorithms. In this section we look at an example and show how it can be transformed so that it is more efficient.

The example is the token module from [27], which supplies a simple lexical scanner. The INTRODUCTION, SYNTAX, and part of the TYPES section of the specification are shown in Figure 4.4. Besides `s_init`, which initializes the module, there are two other set-calls: `s_str` loads the string to be scanned and `s_next` advances the module to the next token. The get-calls `g_pos`, `g_toktyp`, and `g_tokval` give the position in the string (before any token, at a token, or after all tokens in the string), the type of the current token, and the value of the token respectively. The types `pos` and `toktyp` are used to represent the position in the string and the type of the token respectively. Both are enumerated types, and both are implemented as unary predicates with the type as the predicate name.

We represent the state of the module using a quadruple

(position,token type,token value,remainder of input string)

```
%
% INTRODUCTION
%
%   The token module extracts typed tokens, one at a time, from
%   a string supplied by the user.

%
% SYNTAX
%
%   s_init.
%   s_str( string ).            %maxstrlen%
%   s_next.                     %after%
%   g_pos -> pos.
%   g_toktyp -> toktyp.
%   g_tokval -> string.         %notok%

%
% TYPES
%
pos( Pos ) :- member( Pos, [before,tok,after] ).
toktyp( Type ) :- member( Type, [badtok,idtok,inttok,realtok] ).
```

Figure 4.4: INTRODUCTION, SYNTAX and TYPES of token module

With this representation we can transform the specification and obtain the implementation of the token module shown in Figure 4.5. For brevity, the definitions of some relations, such as **wschar** and **toktyp**, are left out.

The implementation in Figure 4.5 finds the next token using the relations **next** and **next1**. For an arbitrary string, **next** gives the first token in that string and the remainder of the string after that token. It uses **next1** to do the same for a string starting with a non-whitespace character. Both relations use the **append** relation to decompose a string. This increases the readability of the predicates, but makes it less efficient than a recursive definition.

Using the unfold/fold transformation rules, we can transform these definitions to the clauses:

```
next( [], [], [] ).
next( [X|Xs], T, R ) :- not wschar( X ), next1( [X|Xs], T, R ).
next( [X|Xs], T, R ) :- wschar( X ), next( Xs, T, R ).

next1( [], [], [] ).
next1( [X|Xs], [], [X|Xs] ) :- wschar( X ).
next1( [X|Xs], [X|T], R ) :- not wschar( X ), next1( Xs, T, R ).
```

These definitions are tail-recursive and do not use any of the **append**, **whitespace**, or **nonwhitespace** relations and are therefore more efficient than the original definitions.

Similarly, we can transform the definition of the **newstate** relation so that we no longer need the definition for **next**. After these transformations we obtain the implementation shown in Figure 4.6.

For both transformations above, we need a more general form of folding than the one presented in Chapter 3. This involves the folding of the definition of a predicate that is defined with more than one clause. We expect that, with certain restrictions

```
initstate( s_init, (before,badtok,_,"") ).

newstate( s_str(S), _, (before,badtok,_,S) ).
newstate( s_next, (_,_,_,S), (after,badtok,_,[]) ) :-
   whitespace( S ).
newstate( s_next, (_,_,_,S), (tok,Type,Token,Rest) ) :-
   not whitespace( S ), next( S, Token, Rest ),
   toktyp( Token, Type ).
%------------
holds( g_pos(P), (P,_,_,_) ).
holds( g_toktyp(T), (_,T,_,_) ).
holds( g_tokval(T), (_,_,T,_) ).
%------------
exception( maxstrlen, s_str(S), _ ) :-
   length( S, L ), maxstrlen(M), L > M.
exception( after, s_next, (after,_,_,_) ).
exception( notok, g_tokval(_), (_,badtok,_,_) ).
%------------
% next( I, T, R ) - T is the first token in I, with remainder R
next( Input, Token, Rest ) :-
   append( Blanks, [T | Ts], Input ),
   whitespace( Blanks ), not wschar( T ),
   next1( [T | Ts], Token, Rest ).
next( Input, [], [] ) :- whitespace( Input ).
% next1( I, T, R ) - T is the first token in input I, whose first
%   character is not a whitespace, and the remainder of I is R
next1( Input, Token, [R | Rs] ) :-
   append( Token, [R | Rs], Input ),
   nonwhitespace( Token ), wschar( R ).
next1( Input, Input, [] ) :- nonwhitespace( Input ).
% whitespace( S ) - string S contains whitespace characters only.
whitespace( [] ).
whitespace( [W | Ws] ) :- wschar( W ), whitespace( Ws ).
% nonwhitespace( S ) - string S contains non-whitespace
%   characters only.
nonwhitespace( [] ).
nonwhitespace( [NW | NWs] ) :-
   not wschar( NW ), nonwhitespace( NWs ).
```

Figure 4.5: Token module after data reification

```
% state( (Pos,Type,Val,Rem) ) :-
%     pos( Pos ), toktyp( Type ), string( Val ), string( Rem ).
%------------
initstate( s_init, (before,badtok,_,"") ).

newstate( s_str(S), _, (before,badtok,_,S) ).
newstate( s_next, (_,_,_,[]), (after,badtok,_,[]) ).
newstate( s_next, (_,_,_,[C|Cs]), NS ) :-
   wschar( C ), newstate( s_next, (_,_,_,Cs), NS ).
newstate( s_next, (_,_,_,[C|Cs]), (tok,Type,[C|Ts],Rest) ) :-
   not wschar( C ), next1( Cs, Ts, Rest ), toktyp( [C|Ts], Type ).
%------------
holds( g_pos(P), (P,_,_,_) ).
holds( g_toktyp(T), (_,T,_,_) ).
holds( g_tokval(T), (_,_,T,_) ).
%------------
exception( maxstrlen, s_str(S), _ ) :-
   length( S, L ), maxstrlen(M), L > M.
exception( after, s_next, (after,_,_,_) ).
exception( notok, g_tokval(_), (_,badtok,_,_) ).
%------------
next1( [], [], [] ).
next1( [X|Xs], [], [X|Xs] ) :- wschar( X ).
next1( [X|Xs], [X|T], R ) :- not wschar( X ), next1( Xs, T, R ).
```

Figure 4.6: Implementation of token module

(e.g., unfolding the resulting clause should produce the original clauses), this folding also preserves the minimal model of the program, but this still remains to be proven.

We have shown one example of algorithm reification. The type of transformation that can be performed depends on the algorithms used in the specification. For that reason the transformation process cannot be as systematic as for data reification. As explained in Section 3.4.1, FROST can be used to apply a wide variety of transformation techniques to obtain an efficient implementation.

Chapter 5

Concluding Remarks

5.1 Contributions

The SCS method. We introduced the SCS method for writing module interface specifications in logic. This method, which can be regarded as a combination of the SCR method [27] and the trace method [2], allows the user to write precise interface specifications that can be executed as Prolog programs.

The SCS method obtains precision in both syntax and semantics. The semantics of the SCS method are based on the established semantics of first-order predicate logic. This means that statements about the module's observable behaviour are logical consequences of the specification, when regarded as a theory of first-order predicate logic.

SCS specifications can be executed as Prolog programs. There are two advantages to having an executable specification:

1. We can check that it behaves as was intended. The test cases used for this purpose can later be applied to test an implementation of the module.

2. It can serve as an oracle to test an implementation. This allows the tester to define a large number of test cases without having to specify the correct behaviour for each test case.

The PROTEST system [29] allows us to both test specifications and to use a specification as an oracle to test an implementation.

Complete sets of frontiers. The frontier theorem shows that a complete set of frontiers for a query has the same minimal model as the original program with respect to that query. Thus we can specialize a program to a particular query without having to eliminate redundant clauses. We showed how FROST incorporates this idea into an interactive transformation system. Besides the unfold transformation, which is the only transformation rule the frontier theorem deals with, FROST also allows the application of other transformation rules in the construction of a frontier.

A complete set of frontiers has three advantages over the unfolding as proposed by Tamaki and Sato [59]:

1. It allows us to specialize a program for an instance of a goal. The result by Tamaki and Sato only applies to the most general form of a goal. To make their result applicable for an instance of a goal we have to add a definition to the program.

2. There is no need to eliminate redundant clauses from the program. After unfolding, certain clauses in the program are no longer needed. A syntactic analysis can be used to determine the redundant clauses, but with a complete set of frontiers there is no need for this.

3. In Section 3.2.2 we showed that the definition of a complete set of frontiers naturally suggests an algorithm for specializing a program to a particular query.

Lloyd and Shepherdson [40] prove a result similar to the frontier theorem using a different criterion for completeness. Although this result also deals with programs containing negated atoms, it does not suggest a method for guiding the transformation process.

From specification to implementation in logic. We have shown, using a simple example, how we can derive an efficient Prolog implementation by transforming an SOS specification. In general, this transformation involves two steps. During data reification we change the data structure to represent the state of the module from an abstract representation (the result-term) to a more concrete one. The algorithm reification step consists of transforming the easy to understand, but possibly inefficient, algorithms used in the specification to more efficient, and probably less clear, versions.

The steps involved in part of the data reification are similar for each module. For certain predicates in the specification, the transformation is simple and systematic enough so that it could be automated. For others, however, the unfold/fold transformations are not powerful enough to obtain the desired definitions. In these cases we have to rely on other theorem-proving techniques, such as the system presented in [16], if we want to derive the implementation from the specification. The transformations that can be used during algorithm reification depend on the algorithms used in the specifications, and as such, the algorithm reification cannot be as systematic as the data reification.

5.2 Conclusions

Abstract specifications. The SCS method is abstract in that it describes the state of the module only in terms of the sequence of access program calls on the module. This allows the designer of the specification to focus on the requirements of the module and does not bias the implementor towards a particular implementation.

However, it turns out that it is sometimes very hard to write specifications using result-terms. For example, for the tree module shown in Appendix C it was a lot harder to write an SCS specification than it was to write a state-based specification using a Prolog term to represent the tree. As a result, the abstract specification is less clear than the state-based version. Another disadvantage of the more complicated abstract specification is that in this case we were unable to derive the state-based version from the original specification.

When an abstract specification is a lot less clear than the corresponding state-based version, or it takes a lot more effort to write the former, it becomes advantageous to write the state-based version rather than the abstract one. This can eliminate the need for data reification. However, the representation of the state in a state-based version is chosen to make the specification as clear as possible. This means that there can be yet another representation that would make the specification more efficient as a Prolog program. The data reification step then consists of transforming from the first representation to the more efficient one.

Transformation, verification, and testing. Program transformation is an attractive technique for obtaining a correct and efficient implementation from a specification. The implementation is derived from the specification so that the implementation need not be known beforehand, as is the case with program verification and

testing. It has the advantage over testing that it proves the program is correct, rather than showing that it behaves correctly for a finite number of test cases.

In most cases we could not use program transformation alone to derive the implementation from the specification. For these cases, we tried to use other theorem-proving techniques to show that the implementation behaves according to the specification, i.e., we performed program verification. There are several problems with this. The theorems that need to be proved are often complicated. Therefore it is advantageous to use program transformation as much as possible to simplify the theorem-proving task. Another problem is that it is hard to prove the theorems manually. Theorem-proving systems such as described in [16, 20] could eliminate this problem, but even these systems require assistance.

Considering the problems with program transformation and program verification, the need for testing becomes obvious[1]. With the PROTEST system [29] we can use the same test cases that were used to test the specification to test the implementation. Testing techniques to thoroughly test modules and techniques for simplifying the testing task when a runnable specification is available are discussed in [29].

5.3 Future work

The method we propose to obtain efficient implementations from specifications has been tried on a small number of modules. More experience is needed to see if the same method can be applied to other modules.

It might be the case that it is hard to write SCS specifications for some of these modules. For example, we have not addressed the issue of how to specify non-

[1] Even with fully automated program transformation/verification systems there is the problem of having to verify the system itself, the software it runs on, etc., and the need for testing remains.

deterministic modules, and indeed, the language of the SCS method might not be adequate to deal with such modules. A second problem is the cost associated with writing abstract and/or executable specifications. We have already noted that, in certain cases, it is too costly to write abstract specifications. Similarly, the cost of making the specifications executable can become too high. We then have to decide if we can still use the SCS method, perhaps in combination with the use of prose (as Hoffman proposes for the SCR method [27]).

For even the simple modules we have looked at so far, the unfold/fold transformation rules are not powerful enough to transform the specification into the desired implementation. Perhaps a theorem-proving system such as mentioned in [16] could help us derive the implementations for such cases. If this is not the case, we need to look at testing techniques such as discussed in [29] to check that an implementation behaves according to the specification.

The frontier theorem currently deals with the unfold transformation rule. We would like to include other transformation rules, such as folding, in the construction of a set of frontiers, as is being done in the FROST system. We also need to prove that the more general form of folding used in the example in Section 4.2 preserves the minimal model of a logic program. We expect this is the case, whenever the result of unfolding the resulting clause using the old definition of the predicate produces the original clauses.

Bibliography

[1] R. Balzer, N. Goldman, and D. Wile. On the transformational implementation approach to programming. In *Second International Conference on Software Engineering*, pages 337–344, 1976.

[2] W. Bartussek and D.L. Parnas. Using assertions about traces to write abstract specifications for software modules. In *Second Conference of the European Cooperation in Informatics*, pages 211–236. Springer-Verlag, 1978.

[3] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[4] S.H. Caine and E.K. Gordon. PDL—a tool for software design. In *Proceedings AFIPS National Computer Conference 1975*, pages 271–276, 1975.

[5] K. Clark. The synthesis and verification of logic programs. Research report, Imperial College, 1978.

[6] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[7] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1984.

[8] J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16(1):1–46, 1981.

[9] J. Darlington. Program transformation. *Byte*, pages 201–216, 1985.

[10] S.K. Debray. Optimizing almost tail-recursive Prolog programs. In J-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 204–219. Springer-Verlag, Berlin, 1985.

[11] R. DeMillo, R. Lipton, and A. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, 1979.

[12] E. Dijkstra. Programming considered as a human activity. In E.N. Yourdon, editor, *Classics in Software Engineering*, pages 3–9. Yourdon Press, New York, 1979.

[13] E. Dijkstra. Structured programming. In E.N. Yourdon, editor, *Classics in Software Engineering*, pages 43–50. Yourdon Press, New York, 1979.

[14] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[15] J.K. Dixon, J. McLean, and D.L. Parnas. Rapid prototyping by means of abstract module specifications written as trace axioms. *ACM SIGSOFT Software Engineering Notes*, 7(5):45–49, 1982. Working Papers ACM SIGSOFT Rapid Prototyping Workshop 1982.

[16] C. Elkan and D. McAllester. Automated inductive reasoning about logic programs. In R.A. Kowalski and K.A. Bowen, editors, *Proc. of the Fifth International Conference on Logic Programming*, pages 876–892. MIT Press, 1988.

[17] M.E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[18] M.S. Feather. Mappings for rapid prototyping. *ACM SIGSOFT Software Engineering Notes*, 7(5):17–24, 1982. Working Papers ACM SIGSOFT Rapid Prototyping Workshop 1982.

[19] R.W. Floyd. Assigning meaning to programs. In *Proc. Symposia in Applied Mathematics (Vol XIX)*, pages 19–32, 1967.

[20] L. Fribourg. Equivalence-preserving transformations of inductive properties of Prolog programs. In R.A. Kowalski and K.A. Bowen, editors, *Proc. of the Fifth International Conference on Logic Programming*, pages 893–908. MIT Press, 1988.

[21] J.A. Goguen and J.J. Tardo. An introduction to OBJ: A language for writing and testing formal algebraic program specifications. In *Proc. IEEE Conference on Specification for Reliable Software*, pages 170–189, 1979.

[22] S.J. Goldsack. *Ada for Specification: Possibilities and Limitations.* Cambridge University Press, 1985.

[23] J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):27–52, 1978.

[24] A. Hansson and S.-A. Tärnlund. Program transformation by data structure mapping. In K.L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 117–122. Academic Press, London, 1982.

[25] D.M. Hoffman. Software inspection, verification, and testing: A hybrid approach. To appear in: Proceedings of the Pacific Northwest Software Quality Conference 1990.

[26] D.M. Hoffman. A CASE study in module testing. In *Proceedings of the Conference on Software Maintenance*, pages 100–105, 1989.

[27] D.M. Hoffman. Practical interface specification. *Software - Practice and Experience*, 19(2):127–148, 1989.

[28] D.M. Hoffman and R. Snodgrass. Trace specifications: Methodology and models. *IEEE Transactions on Software Engineering*, 14(9):1243–1252, 1988.

[29] D.M. Hoffman and P.A. Strooper. Automated module testing in Prolog. Technical Report DCS-134-IR, University of Victoria, July 1990.

[30] D.M. Hoffman and Y. Wang. Executable prototypes of trace specifications. In *CIPS Edmonton*, pages 176–184, 1987.

[31] C.J. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, 1981.

[32] G. Huet and D.C. Oppen. Equations and rewrite rules: A survey. In R.V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, 1980.

[33] C.B. Jones. Program specification and verification in VDM. Technical Report UMCS-86-10-5, University of Manchester, 1986.

[34] C.B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, 1986.

[35] H.J. Komorowski. Synthesis of programs in the framework of partial deduction. Technical Report Ser. A, No. 81, Abo Akademi, Finland, 1989.

[36] R.A. Kowalski. Predicate logic as programming language. In *Information Processing 74*, pages 569–574, 1974.

[37] R.A. Kowalski. *Logic for Problem Solving.* North-Holland, New York, 1979.

[38] F. Lin and F.E. Hunt. LCD-reification: A formal method for developing prolog programs. In *Proc. of 5th Int. Workshop on Software Specification and Design*, pages 249–256, 1989.

[39] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1987.

[40] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic. Technical Report CS-87-09, Bristol University, 1987.

[41] J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[42] J. McLean. A formal method for the abstract specification of software. *Journal of the ACM*, 31:600–627, 1984.

[43] J. McLean, D. Weiss, and C. Landwehr. Executing trace specifications using Prolog. Technical Report 8940, Naval Research Laboratory, 1986.

[44] C.A. Meadows. A method for automatically translating trace specifications into Prolog. Technical Report 9131, Naval Research Laboratory, 1988.

[45] M.L. Minsky. *Computation.* Prentice Hall, 1967.

[46] D.L. Parnas. Use of abstract interfaces in the development of software for embedded systems. Technical Report 8047, Naval Research Laboratory, June 1977.

[47] D.L. Parnas and P.C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, 1986.

[48] D.L. Parnas and Y. Wang. The trace assertion method of module interface specification. Technical Report 89-261, Queen's University, October 1989.

[49] H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Computing Surveys*, 15(3):199–236, 1983.

[50] A. Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *ACM Symposium on LISP and Functional Programming*, pages 273–281, 1984.

[51] B. Robinet, editor. *Program Transformation.* Dunod, Paris, 1978.

[52] H. Seki and K. Furukawa. Compiling control by a program transformation approach. Technical Memorandum TM-0240, ICOT, 1986.

[53] E.Y. Shapiro. A subset of concurrent Prolog and its interpreter. In E.Y. Shapiro, editor, *Concurrent Prolog*, pages 27–83. MIT Press, 1987.

[54] J.C. Shepherdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers, 1988.

[55] J.M. Spivey, editor. *Understanding Z.* Cambridge University Press, 1988.

[56] S.L. Squires, M. Zelkowitz, and M. Branstad. Rapid prototyping workshop: Overview. *ACM SIGSOFT Software Engineering Notes*, 7(5):2, 1982. Working Papers ACM SIGSOFT Rapid Prototyping Workshop 1982.

[57] P.A. Strooper. A transformation system for logic programs. Technical Report DCS-106-IR, University of Victoria, February 1989.

[58] A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to meta programming. Technical Report TR-126, ICOT, 1985.

[59] H. Tamaki and T. Sato. A transformation system for logic programs which preserves equivalence. Technical Report TR-018, ICOT, 1983.

[60] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Proc. of the Second International Logic Programming Conference*, pages 127–138, 1984.

[61] U.S. Department of Defense, Washington, D.C. *Reference Manual for the Ada Programming Language*, 1815 ACM AdaTec special publication edition, 1982.

[62] M.H. van Emden and K. Yukawa. Logic programming with equations. *Journal of Logic Programming*, 4(4):265–288, 1987.

[63] P. Vasey. Qualified answers and their application to transformation. In *Proc. of the Third International Logic Programming Conference*, pages 425–432, 1986.

[64] C.M. White. *ALS Prolog User's Guide and Reference Manual.* Applied Logic Systems, Inc., 1988.

[65] N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1983.

Appendix A

An Example Transformation Using FROST

We show how we can perform the transformation discussed in Section 3.3.8 using FROST. We assume that a program with the clauses

```
reverse1( L, R, A ) :- reverse( L, R1 ), append( R1, A, R ).

% reverse( L, R ): R is the reverse of list L
% example: reverse( [1,2,3], [3,2,1] )
reverse( [], [] ).
reverse( [X|Xs], L ) :- reverse( Xs, R ), append( R, [X], L ).

% append( U, V, W ): list W is list V appended to list U.
% example: append( [1,2], [3,4,5], [1,2,3,4,5] )
append( [], L, L ).
append( [X|Xs], Y, [X|Z] ) :- append( Xs, Y, Z ).
```

is contained in the file **rev.pro**. We have added the definition for **reverse1** to this file for simplicity.

The following terminal session shows how we perform the transformation. The user input is preceded by >. FROST uses integers to represent the frontiers in the set of frontiers and the clauses inside a frontier. These integers must be used by the user to identify particular frontiers and clauses in the commands to FROST. For example, the command

```
> unfold(1,1).
```

tells FROST to unfold the first goal in the first clause of the current frontier.

```
Enter command:
> loadprog(rev).
Consulting rev...
rev consulted.

Enter command:
> showprog.
% Clauses for reverse1/3.
reverse1(A,B,C) :- reverse(A,D), append(D,C,B).
% Clauses for reverse/2.
reverse([],[]).
reverse([A | B],C) :- reverse(B,D), append(D,[A],C).
% Clauses for append/3.
append([],A,A).
append([A | B],C,[A | D]) :- append(B,C,D).

Enter command:
> addfront(reverse,2).

Current frontier for reverse(A,B):
1. reverse([],[]).
2. reverse([A | B],C) :- reverse(B,D), append(D,[A],C).

Enter command:
> fold(reverse1,3).
```

```
Current frontier for reverse(A,B):
1. reverse([],[]).
2. reverse([A | B],C) :- reverse1(B,C,[A]).

Enter command:
> top.

1. reverse(A,B)

Enter command:
> complete(reverse,2).
*** To get a complete set of frontiers for reverse(A,B),
you can use frontiers [1],
and need frontiers for the goals:
[reverse1(A,B,[C])]

1. reverse(A,B)

Enter command:
> addfront(reverse1,3).

Current frontier for reverse1(A,B,C):
1. reverse1(A,B,C) :- reverse(A,D), append(D,C,B).

Enter command:
> unfold(1,1).

Current frontier for reverse1(A,B,C):
1. reverse1([],A,B) :- append([],B,A).
2. reverse1([A | B],C,D) :-
      reverse(B,E), append(E,[A],F), append(F,D,C).

Enter command:
> prove(1,1).

Current frontier for reverse1(A,B,C):
1. reverse1([],A,A).
2. reverse1([A | B],C,D) :-
      reverse(B,E), append(E,[A],F), append(F,D,C).
```

```
Enter command:
> showmeta.
comm1: [sum(A,B,C)] --> [sum(B,A,C)]
comm2: [mul(A,B,C)] --> [mul(B,A,C)]
ass1: [sum(A,B,C),sum(C,D,E)] --> [sum(B,D,F),sum(A,F,E)]
ass2: [mul(A,B,C),mul(C,D,E)] --> [mul(B,D,F),mul(A,F,E)]
ass3: [append(A,B,C),append(C,D,E)] --> [append(B,D,F),append(A,F,E)]
dist: [sum(A,B,C),mul(C,D,E)] --> [mul(A,D,F),mul(B,D,G),sum(F,G,E)]

Enter command:
> meta(2,ass3).

Current frontier for reverse1(A,B,C):
1. reverse1([],A,A).
2. reverse1([A | B],C,D) :-
      append([A],D,E), append(F,E,C), reverse(B,F).

Enter command:
> prove(2,1).

Current frontier for reverse1(A,B,C):
1. reverse1([],A,A).
2. reverse1([A | B],C,D) :- append(E,[A | D],C), reverse(B,E).

Enter command:
> fold(reverse1,3).

Current frontier for reverse1(A,B,C):
1. reverse1([],A,A).
2. reverse1([A | B],C,D) :- reverse1(B,C,[A | D]).

Enter command:
> top.

1. reverse(A,B)
2. reverse1(A,B,C)

Enter command:
```

```
> complete(reverse,2).
A complete set of frontiers for reverse(A,B)is:
[1,2]

1. reverse(A,B)
2. reverse1(A,B,C)

Enter command:
> saveall('rev.out').

1. reverse(A,B)
2. reverse1(A,B,C)

Enter command:
> halt.
cs14% more rev.out

% Frontier for reverse(A,B):
reverse([],[]).
reverse([A | B],C) :- reverse1(B,C,[A]).

% Frontier for reverse1(A,B,C):
reverse1([],A,A).
reverse1([A | B],C,D) :- reverse1(B,C,[A | D]).
```

Appendix B

Proving Programs Equivalent

We show that the clauses

```
C1: holds( g_cnt(0), 0 ).
C2: holds( g_cnt(A), A ) :- holds( g_cnt(B), B ), A is B+1.
```

have the same minimal Herbrand model as

```
C3: holds( g_cnt(0), 0 ).
C4: holds( g_cnt(A), B ) :-
       holds( g_cnt(C), D ), A is C+1, B is D+1.
```

using the method discussed in [16].

We first translate the above clauses to

```
C1': holds( g_cnt(0), 0 ).
C2': holds( g_cnt(s(A)), s(A) ) :- holds( g_cnt(A), A ).

C3': holds( g_cnt(0), 0 ).
C4': holds( g_cnt(s(A)), s(B) ) :- holds( g_cnt(A), B ).
```

where we have replaced addition with the more succinct *successor* notation for natural numbers. In this notation, `0` represents zero and if `X` represents the number `N`, then `s(X)` represents `N+1`. For example, `s(s(0))` represents the number two. The use of the successor notation not only simplifies the notation, but it also avoids having to deal with several properties of the addition relation in the proof.

Since `C2'` is an instance of `C4'`, both `C1'` and `C2'` are a logical consequence of `C3'` and `C4'`. Similarly, `C3'` is a logical consequence of `C1'` and `C2'`. Thus we only need to prove that `C4'` holds in the minimal model of `C1'` and `C2'`. Note that `C4'` is not a logical consequence of `C1'` and `C2'`, as these have a model which is not a model for `C4'`.

We use some of the steps used in the theorem-proving system proposed by Elkan and McAllester to prove this result [16]. This system performs automated inductive reasoning on logic programs. We present the steps needed to prove the above result; the justification of these steps can be found in [16].

In [16], a transformation is first applied to the program clauses and the formula to be proven so that the system can apply induction in the proof. Since we can prove our result without using induction, we do not apply this transformation here. The proof proceeds by applying a sequence of tactics.

We first apply the *definition expansion* tactic, which, in this case, is the same as the unfold transformation rule. This tactic leaves us with two new clauses to be proven.

The first clause is

```
holds( g_cnt(s(0)), s(0) ).
```

This clause can be proven using the *first-order reasoning* tactic, which involves proving a goal using a logic program. In this case, we need to prove the goal

```
?- holds( g_cnt(s(0)), s(0) ).
```

with the program consisting of `C1'` and `C2'`. Clearly, this query succeeds.

For the second sub-proof, we need to prove that the clause

```
holds( g_cnt(s(s(A))), s(s(A)) ) :- holds( g_cnt(A), A ).
```

holds in the minimal model of `C1'` and `C2'`. We can also prove this using the first-order reasoning tactic. We first add the clause

```
holds( g_cnt(a), a ).
```

to the program. This clause is the skolemized version (`a` is a new constant symbol) of the goal in the body of the clause to be proven. Using this program, we solve the query with as its only goal the skolemized version of the head of the clause we want to prove, which is

```
?- holds( g_cnt(s(s(a)), s(s(a)) ).
```

This query succeeds, thus finishing the proof of the original clause.

Appendix C

Specification and Implementation of Tree

In this section, we show an SCS specification and an implementation of a binary tree module similar to the one presented in [30] (exceptions are added). This module allows the user to build and traverse a binary tree. At any time, there is a *current* node in the tree. There are calls to add and delete nodes, to set and examine the value of the current node, to move up or down the tree, and to check if nodes exist above or below the current node.

C.1 Specification

The SCS specification is based on the methodology for writing trace specifications proposed in [28]. The specification is very similar to the trace specification shown in [30]. The canonical form for the result-terms consists of the calls required to build the tree listed in pre-order, followed by the calls required to move from the root of

the tree to the current node. Using this canonical form, the **equal** relation defines how each set-call affects the canonical form of the tree. It uses several relations to simplify the specification.

The **parse** predicate splits up a result-term in canonical form into four parts:

1. The calls to create the part of the tree before the current node.
2. The calls to create the sub-tree rooted at the current node.
3. The calls to create the part of the tree after the current node.
4. The calls to move from the root of the tree to the current node.

The **tree** predicate further splits up the calls to create a tree in the calls to create the left and right sub-trees and the call (if any) to set the value of the current node.

Further details of the specification are explained in [30]. Note that the specification is fairly complicated. Part of this complication is due to the complexity of the module itself. However, when we compare the specification to the implementation shown in the next section, we see that the use of result-terms to represent the state of the module also complicates the specification.

```
%
% INTRODUCTION
%
%    The binary tree module provides access to a binary tree.  Each
%    node has a value field which can be undefined.  You can add
%    and delete nodes, determine the value of a node, and examine
%    whether it has a parent, left or right child.  After initiali-
%    sation the root node is the current node.  Deleting a node
%    deletes the entire sub-tree hanging from it and makes the
%    parent the current node.  Deleting the root deletes the entire
%    tree except for a now undefined root, which becomes the current
```

```
%    node.  Adding a left or right child makes that child the
%    current node.
%
% SYNTAX
%
%   s_init.
%   s_current( integer ).
%   g_current -> integer.          %noval%
%   s_mvleft.                      %nonode%
%   s_mvright.                     %nonode%
%   s_mvparent.                    %nonode%
%   s_mkleft.                      %exnode%
%   s_mkright.                     %exnode%
%   s_del.
%   g_exleft -> boolean.
%   g_exright -> boolean.
%   g_exparent -> boolean.
%
% RELATIONS
%
parse( Trace, Mkbef, Mkcurr, Mkaft, Mvs ) :-
   app3( Mvs, Mks, [s_init], Trace ),
   allmvs( Mvs ),
   tree( Mks, _, _, _ ),
   match( Mks, Mvs, Mkbef, Mkcurr, Mkaft ).

unparse( Trace, Mkbef, Mkcurr, Mkaft, Mvs ) :-
   app3( Mkcurr, Mkbef, [s_init], Beg ),
   app3( Mvs, Mkaft, Beg, Trace ).

tree( Tree, Root, LTree, RTree ) :-
   app3( RTree, LTree, Root, Tree ),
   root( Root ), ltree( LTree ), rtree( RTree ).
root( [] ).
root( [s_current(_)] ).
ltree( [] ).
ltree( T ) :-
   append( [s_mvparent|T1], [s_mkleft], T ), tree( T1, _, _, _ ).
rtree( [] ).
```

```
rtree( T ) :-
   append( [s_mvparent|T1], [s_mkright], T ), tree( T1, _, _, _ ).

match( Curr, [], [], Curr, [] ).
match( Mks, Mvs, Bef, Curr, Aft ) :-
   append( Mvs1, [s_mvleft], Mvs ),
   tree( Mks, Root, LT, RT ),
   append( [s_mvparent|Mks1], [s_mkleft], LT ),
   match( Mks1, Mvs1, Bef1, Curr, Aft1 ),
   append( Bef1, [s_mkleft|Root], Bef ),
   append( RT, [s_mvparent|Aft1], Aft ).
match( Mks, Mvs, Bef, Curr, [s_mvparent|Aft] ) :-
   append( Mvs1, [s_mvright], Mvs ),
   tree( Mks, Root, LT, RT ),
   append( [s_mvparent|Mks1], [s_mkright], RT ),
   match( Mks1, Mvs1, Bef1, Curr, Aft ),
   app3( Bef1, [s_mkright|LT], Root, Bef ).

allmvs( L ) :-
   count( s_mvleft, L, N1 ), count( s_mvright, L, N2 ),
   N is N1 + N2, length( L, N ).

app3( P1, P2, P3, T ) :- append( P1, P23, T ), append( P2, P3, P23 ).

replcurr( Curr, C, NewCurr ) :-
   not append( _, [s_current(_)], Curr ),
   append( Curr, [C], NewCurr ).
replcurr( Curr, C, NewCurr ) :-
   append( Curr1, [s_current(_)], Curr ),
   append( Curr1, [C], NewCurr ).

nonempty( [], false ).
nonempty( [_|_], true ).

%
% EFFECTS
%
% Equalities
equal( [s_init], [s_init] ).
```

```
equal( [s_current(I)|T1], T3 ) :-
   equal( T1, T2 ),
   parse( T2, Mkbef, Mkcurr, Mkaft, Mvs ),
   replcurr( Mkcurr, s_current(I), Mkcurr1 ),
   unparse( T3, Mkbef, Mkcurr1, Mkaft, Mvs ).
equal( [s_mvleft|T1], [s_mvleft|T2] ) :- equal( T1, T2 ).
equal( [s_mvright|T1], [s_mvright|T2] ) :- equal( T1, T2 ).
equal( [s_mvparent|T1], T2 ) :- equal( T1, [_|T2] ).
equal( [s_mkleft|T1], T3 ) :-
   equal( T1, T2 ), parse( T2, Bef, Curr, Aft, Mvs ),
   tree( Curr, Root, _, RT ),
   append( [s_mkleft|Root], Bef, NewBef ),
   app3( Aft, RT, [s_mvparent], NewAft ),
   unparse( T3, NewBef, [], NewAft, [s_mvleft|Mvs] ).
equal( [s_mkright|T1], T3 ) :-
   equal( T1, T2 ), parse( T2, Bef, Curr, Aft, Mvs ),
   tree( Curr, Root, LT, _ ),
   app3( [s_mkright|LT], Root, Bef, NewBef ),
   append( Aft, [s_mvparent], NewAft ),
   unparse( T3, NewBef, [], NewAft, [s_mvright|Mvs] ).
equal( [s_del|T1], [s_init] ) :-
   equal( T1, T2 ), parse( T2, [], _, [], [] ).
equal( [s_del|T1], T3 ) :-
   equal( T1, T2 ),
   parse( T2, [_|Bef], Curr, Aft, [_|Mvs] ),
   append( NewAft, [_], Aft ),
   unparse( T3, Bef, [], NewAft, Mvs ).
% Equivalence axiom:
holds( C, S1 ) :- equal( S1, S2 ), holds1( C, S2 ).
% Value of g_current
holds1( g_current(I), W ) :-
   parse( W, _, Curr, _, _ ), append( _, [s_current(I)], Curr ).
% Value of g_exleft
holds1( g_exleft(Bool), W ) :-
   parse( W, _, Curr, _, _ ), tree( Curr, _, LeftST, _ ),
   nonempty( LeftST, Bool ).
% Value of g_exright
holds1( g_exright(Bool), W ) :-
   parse( W, _, Curr, _, _ ), tree( Curr, _, _, RightST ),
```

```
   nonempty( RightST, Bool ).
% Value of g_exparent
holds1( g_exparent(Bool), W ) :-
   parse( W, _, _, _, Mvs ), nonempty( Mvs, Bool ).
%
% EXCEPTIONS
%
% exnode
exception( exnode, s_mkleft, S ) :- holds( g_exleft(true), S ).
exception( exnode, s_mkright, S ) :- holds( g_exright(true), S ).
% nonode
exception( nonode, s_mvleft, S ) :- holds( g_exleft(false), S ).
exception( nonode, s_mvright, S ) :- holds( g_exright(false), S ).
exception( nonode, s_mvparent, S ) :- holds( g_exparent(false), S ).
% noval
exception( noval, g_current(_), S ) :-
   equal( S, S1 ), parse( S1, _, Curr, _, _ ),
   not append( _, [s_current(_)], Curr ).
```

C.2 Implementation

The implementation of the tree module represents the state of the module as a pair (`Tree`,`Path`), where:

- `Tree` is a term representing the content of the tree. `nil` represents an empty tree, and a triple (`L`,`C`,`R`) represents a tree with value `C` and left and right subtrees `L` and `R` respectively. When the value of a node has not yet been defined, the value field of this node is `undef`.
- `Path` is a list of "`l`"s and "`r`"s representing the path from the root of the tree to the current node.

Several predicates are defined to simplify the implementation. Given a tree and the path to the current node, `curr` returns the sub-tree rooted at the current node.

Given a tree, the path to the current node, and a value, **setcurr** returns a new tree with the value of the current node set to the given value. Given a tree, the path to the current node, and a direction (left or right), **addnode** adds a new node under the current one in the given direction. Similarly, **delnode** deletes a subtree under the current node.

```
% state( (Tree,Path) ) :- tree( Tree ), path( Path ).
% tree( nil ).
% tree( (L,C,R) ) :- tree( L ), node( C ), tree( R ).
% node( undef ).
% node( I ) :- integer( I ).
% path( [] ).
% path( [D|Ds] ) :- dir( D ), path( Ds ).
% dir( l ).
% dir( r ).
%--------------
initstate( ((nil,undef,nil),[]) ).

newstate( s_current(I), (T1,P), (T2,P) ) :- setcurr( P, I, T1, T2 ).
newstate( s_mvleft, (T,P), (T,NP) ) :- append( P, [l], NP ).
newstate( s_mvright, (T,P), (T,NP) ) :- append( P, [r], NP ).
newstate( s_mvparent, (T,P), (T,NP) ) :- append( NP, [_], P ).
newstate( s_mkleft, (T,P), (NT,NP) ) :-
   append( P, [l], NP ), addnode( P, l, T, NT ).
newstate( s_mkright, (T,P), (NT,NP) ) :-
   append( P, [r], NP ), addnode( P, r, T, NT ).
newstate( s_del, (_,[]), ((nil,undef,nil),[]) ).
newstate( s_del, (T,P), (NT,NP) ) :-
   append( NP, [Dir], P ), rmnode( NP, Dir, T, NT ).
%--------------
holds( g_current(I), (T,P) ) :- curr( P, T, (_,I,_) ).
holds( g_exleft(B), (T,P) ) :-
   curr( P, T, (L,_,_) ), nonempty( L, B ).
holds( g_exright(B), (T,P) ) :-
   curr( P, T, (_,_,R) ), nonempty( R, B ).
holds( g_exparent(false), (_,[]) ).
holds( g_exparent(true), (_,[_|_]) ).
```

```
%--------------
exception( exnode, s_mkleft, S ) :- holds( g_exleft(true), S ).
exception( exnode, s_mkright, S ) :- holds( g_exright(true), S ).
exception( nonode, s_mvleft, S ) :- holds( g_exleft(false), S ).
exception( nonode, s_mvright, S ) :- holds( g_exright(false), S ).
exception( nonode, s_mvparent, (_,[]) ).
exception( noval, g_current(_), (T,P) ) :- curr( P, T, (_,undef,_) ).
%--------------
nonempty( nil, false ).
nonempty( (_,_,_), true ).

curr( [], T, T ).
curr( [l|P], (Lt,_,_), C ) :- curr( P, Lt, C ).
curr( [r|P], (_,_,Rt), C ) :- curr( P, Rt, C ).

setcurr( [], I, (Lt,_,Rt), (Lt,I,Rt) ).
setcurr( [l|P], I, (Lt,C,Rt), (NLt,C,Rt) ) :-
   setcurr( P, I, Lt, NLt ).
setcurr( [r|P], I, (Lt,C,Rt), (Lt,C,NRt) ) :-
   setcurr( P, I, Rt, NRt ).

addnode( [], l, (_,C,Rt), ((nil,undef,nil),C,Rt) ).
addnode( [], r, (Lt,C,_), (Lt,C,(nil,undef,nil)) ).
addnode( [l|P], D, (Lt,C,Rt), (NLt,C,Rt) ) :-
   addnode( P, D, Lt, NLt ).
addnode( [r|P], D, (Lt,C,Rt), (Lt,C,NRt) ) :-
   addnode( P, D, Rt, NRt ).

rmnode( [], l, (_,C,Rt), (nil,C,Rt) ).
rmnode( [], r, (Lt,C,_), (Lt,C,nil) ).
rmnode( [l|P], D, (Lt,C,Rt), (NLt,C,Rt) ) :-
   rmnode( P, D, Lt, NLt ).
rmnode( [r|P], D, (Lt,C,Rt), (Lt,C,NRt) ) :-
   rmnode( P, D, Rt, NRt ).
```