Department of Computer Science
University of Manchester
Manchester M13 9PL, England

Technical Report Series
UMCS-89-7-1

Ralf Kneuper

Symbolic Execution as a Tool for Validation of Specifications

SYMBOLIC EXECUTION AS A TOOL FOR VALIDATION OF SPECIFICATIONS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE

By
Ralf Kneuper
Department of Computer Science
March 1989

DECLARATION

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Abstract

This thesis investigates symbolic execution, a method originally developed for the verification and validation of programs, and extends it to deal with (state-based) specification languages as well.

The main problems encountered are implicit specifications and the intended genericity with respect to the specification language used. Implicit specifications are handled by introducing 'description values': identifiers take as values predicates which describe their 'actual value', the value one would get by actual execution.

Language genericity is achieved by expressing the definition of symbolic execution in terms of the semantics of the specification language used. Both denotational and operational semantics of symbolic execution are discussed. The denotational semantics of symbolic execution, expressed in terms of the denotational semantics of the specification language, provide a correctness notion for symbolic execution. A description of the operational semantics of the language is used as a parameter to tailor symbolic execution to that language.

Based on these ideas, a symbolic execution system called SYMBEX is described and specified. SYMBEX is to form part of the project support environment IPSE 2.5 and help the user to validate a specification by symbolically executing it.

Finally, the achievements and limitations of this approach are discussed, and suggestions made for future work.

Acknowledgements

Above all, my thanks go to Professor Cliff Jones, my supervisor, for his advice and encouragement during the preparation of this thesis. Furthermore, I would like to thank the other members of the IPSE 2.5 project both here in Manchester and at RAL, Roy "But what does that really mean?" Simpson, John Fitzgerald, and all the other people I shared office with, for many fruitful discussions and for providing the environment (in both the technical and the non-technical sense) that made this work not just possible but even enjoyable.

I would also like to thank my parents who, for so many years, gave me the support needed to go all the way from ABC to PhD. Last not least my thanks go to Ilona for her patience and support.

This work was done as part of the IPSE 2.5 project. The IPSE 2.5 is being developed under an Alvey/SERC project by a consortium of the following member organisations: STC plc, ICL plc, University of Manchester, Dowty Defence & Air Systems Ltd., Plessey Research Roke Manor Ltd., SERC Rutherford Appleton Laboratory, British Gas plc.

Parts of this thesis are revised versions of internal project documents written by the author. An early version of §6.2 and Chapter 7 was also published as technical report [Kne87b].

Contents

Chapter 1

Introduction

Ein Mensch sitzt kummervoll und stier
Vor einem weißen Blatt Papier.
Jedoch vergeblich ist das Sitzen —
Auch wiederholtes Bleistiftspitzen
Schärft statt des Geistes nur den Stift.
Selbst der Zigarre bittres Gift,
Kaffee gar, kannenvoll geschlürft,
Den Geist nicht aus den Tiefen schürft,
Darinnen er, gemein verbockt,
Höchst unzugänglich einsam hockt.
Dem Menschen kann es nicht gelingen,
Ihn auf das leere Blatt zu bringen.
Der Mensch erkennt, daß es nichts nützt,
Wenn er den Geist an sich besitzt,
Weil Geist uns ja erst Freude macht,
Sobald er zu Papier gebracht.

Eugen Roth: Arbeiter der Stirn, in: Ein Mensch

1.1 Motivation

One major problem in producing software is the capture of the user's requirements. Although one can (at least in theory) prove the correctness of an implementation with respect to a specification, this is no help at all if the specification itself is not correct, i.e. does not match the user requirements. When analysing these informal user requirements and writing a specification, there are several problems that have to be overcome in order to produce a correct specification:

- specifier and user put different meanings to the same words
- users do not exactly know their requirements
- (formal) specifications can be difficult to understand because of
  - unfamiliar syntax

– unexpected interactions between different parts of the specification

The first of these problems can partly be overcome by using formal specification methods. Even a formal specification, however, is often not clear enough for the user to see what the system actually will do, especially for users with no previous experience with computers and formal methods. Although he or she[1] might think the specification describes her requirements, this might be seen to be false once she starts using the system. Changing the system at this stage will usually be rather expensive. It would therefore be preferable if we could help the user to check the system against her informal requirements *before* it is implemented, by giving her as much 'experience' of the final system as is possible at this stage.

One is therefore looking for ways to *validate* a specification at a very early stage in the development of a software system. Validation is the process of trying to ensure that a specification or program describes the informal user requirements. This means that one has to compare it with the informal requirements on the system. In verification, on the other hand, one looks backwards and checks that the result of a development step satisfies its specification. Hence, in verification one compares an implementation with its (formal) specification. This implies that validation is an inherently *informal* process (although it may contain formal components), while verification, at least in theory, can be made completely formal. According to [Zav84], validation is concerned with "building the right system", while verification is concerned with "building the system right".

There are a number of ways of validating a specification, including inspection, static analysis (analysis of syntax and static semantics), and animation. Animation is taken to mean any method for making the specification 'move' or 'behave' in some way in order to derive some consequences or properties of the specified software system before it is actually implemented. Possible techniques include testing and prototyping, symbolic execution, and general formal reasoning.

By animating a specification, the user should be able to experiment with it, check it in a form that she can understand, and have necessary changes made while it is still comparatively easy and cheap to do so. As a result, the user might change her mind about certain requirements once they are seen in use (which otherwise would happen *after* the system has been implemented), and perhaps even try out an alternative specification.

One particular way of animating a formal specification is to *symbolically execute* it, which is the main topic of this thesis and described in §1.3.

The following model of software development will be assumed: given a vague description of a problem, one starts off by analysing the requirements in detail. They are then formalised in the specification, which is the starting point of a (usually iterative) refinement process. In each refinement step, one starts from a specification and implements it, i.e. transforms it into a less abstract form. In the first step, this specification is the original system specification. In each following step, the result of the previous step is considered as a new specification, which is then implemented again. This process is repeated until one gets the final product which can be executed by computer.[2] The main difference between this model and "waterfall models" is the introduction of iteration into the model. This is to allow for the fact that the software developer

[1] In future I shall not make this distinction any more, but just refer to the user as either 'he' or 'she' at random.

[2] The concept of *executable* terms is discussed in §3.2. In addition to executability, the final product will usually have to satisfy other requirements, such as efficiency etc.

will often want to go back to a previous stage in the development and make some changes, for example because the specification turns out not to describe the wanted properties. On the other hand, maintenance is omitted in this model since this thesis is mainly concerned with getting the *early* steps in software development right.

In this thesis, the term 'specification' is always used in the more restricted sense of *formal* and *functional* specification of a software system, i.e. a specification describes functional properties of the final system, as opposed to performance properties etc., and does so in a formal language.

The insistence on *formal* specification languages indicates that this work is concerned with the formal methods approach to software development. The author of this thesis is aware of the criticisms of this approach [MLP79, Nau82, Fet88]. However, in the author's opinion these criticisms only point out the *limitations* of the formal methods approach which certainly cannot, in itself, guarantee the correctness of any software system with respect to the user's requirements, let alone guarantee the correctness of the result of running the software system on a computer in the physical world under all circumstances. While it is important to keep such limitations in mind, they do not invalidate the formal methods approach itself.

To some extent, this thesis addresses one of the limitations of this approach, namely the possible mismatch between the user's actual requirements on a software system and her requirements as expressed in the formal specification. Techniques such as symbolic execution can be employed to reduce this problem, but one should keep in mind that it is inherently impossible to completely solve it; no validation or verification technique, whether formal or informal, can ever hope to completely eliminate the possibility of such a mismatch.

1.2 Validation and animation of specifications

Probably the most common approach to validation of software systems today is to test the (more or less) finished product. This approach has a number of well-known disadvantages, mainly that it is very unreliable (testing can show the presence of errors, but never their absence), and errors can only be discovered fairly late in the development process, when a lot of other work may have been based on an erroneous decision, which makes its correction rather expensive.

Animation as described here is part of a larger validation process. It can be used as a validation tool both during and at the end of the specification stage, the stage when validation is both most difficult and most valuable. Although animation will not usually be used much after this stage, validation has to continue until the final system has been implemented. The following discussion is only concerned with animation and not with other validation techniques, such as static checks, including for example checking of syntax and static semantics, even though their use should obviously form part of the validation process. It will always be assumed that syntax checks have already been done and the specifications handled are syntactically correct.

Animation can be done on different levels, for example:

Formal reasoning Deriving properties of the specification using theorem proving techniques. This can be a useful technique in some cases but in general it is often not clear what should

be proven about a specification. Possible properties to derive include implementability, security with respect to some security model, or correct treatment of certain border cases.

Formal reasoning is a very general technique and can be said to include both actual and symbolic execution: execution can be viewed as deriving theorems of the form '*input* = ... $\Rightarrow$ *output* = ...'.

Actual execution (Testing) Interpreting the specification on given input values. This is discussed in more detail below.

Symbolic execution Running the specification on symbolic input, i.e. variables over the input domain or, more generally, predicates on such variables, called 'description values'. This approach is the subject of this thesis.

User interface prototyping User interface (UI) ideas can be used for animating a specification in two different contexts. First, they can be used to animate and validate the UI, as opposed to the *functionality* of the system. This usually involves building a prototype that displays only some of the functionality of the system, but basically the same UI as is intended for the final system, or at least a good graphical description of it. Such a graphical description might be most adequate for computer systems that regulate or control some other equipment, but where the user has to input some data, for example by pressing buttons.

One system that falls into this category is SPI[3], which is used for specifying/prototyping user interaction with a system by considering this interaction as a sequence of basic steps called events. Events may be themselves defined, or they may just have a name.

Second, graphics can be used to help understand the functionality of a specified system. In this case, they just provide a different front-end (or view) of the output of animation. Consider for example a specification of a lift system (this example has been used to demonstrate animation in the FOREST project [QB+87, §3.4.4]). Rather than describing with formulae and/or text that the lift moves from one floor to another, one might display a picture of a lift moving on the screen.

This second approach seems very difficult to generalise: any one system can probably only support graphical animation of a small group of similar applications.

A survey of different approaches to user interface prototyping is given in [HI88, §3.2].

Each of the methods described has got a number of drawbacks if used on its own as a tool for ensuring the correctness of a program. To a certain extent, these can be overcome by combining the different methods, and using each to check a particular aspect of the program's correctness.

Actual execution and prototyping

For some languages, actual execution of specifications will be possible directly (this is often referred to as *prototyping*). As described in [Flo84], a prototype is a system that displays some, but

[3]Developed at STC by H. Alexander, cf. [Ale87]

not all of the features of the final product. This way, one can try out some ideas, without investing the effort to build a complete system. Which features are left out depends on the particular application; a common approach is to ignore efficiency and UI questions and build a prototype which only displays (some of) the functionality of the final system. Often this can be done in a language that is higher-level than the implementation language of the final system, because the prototype does not have to be as efficient.

[Flo84] distinguishes three main classes of prototyping:

exploratory prototyping puts the emphasis on clarifying requirements and on helping communication between software developer and user. Used for discussing various alternative solutions.

experimental prototyping puts the emphasis on determining the adequacy of a proposed solution before implementing it.

evolutionary prototyping puts the emphasis on adapting the system gradually to changing requirements. In this case, the prototype gradually develops into the final product.

The following is mainly concerned with experimental prototyping, since it is assumed that a specification (and hence a 'proposed solution') already exists, or at least a high-level rudimentary version of it.

In general, prototyping is different from animation of specifications since a prototype is not usually derived directly from the specification. It usually has to be implemented separately and, since it is executable, the prototyping language cannot be as rich as a specification language might be, an aspect that is often ignored in the prototyping literature. Strictly speaking, prototyping can only be regarded as animation if the prototype itself is (part of) the specification of the final system, or at least can be derived directly from it. Languages suitable for this are often referred to as *executable specification languages*. [HJ88] gives a detailed account why it is not advisable to restrict oneself to *executable* specification languages.

Examples of executable specification languages that are used for prototyping are me too [Hen84, HM85] and EPROL [HI88]. They are both based on the executable part of VDM,[4] expressed in a functional style. In particular, this implies that implicit definitions and operations (functions with side effects) cannot be handled. In me too, specifications written in this restricted subset of VDM are then translated into a version of LISP called Lispkit. EPROL is interpreted in the EPROS prototyping system.

In general, however, a specification will contain non-executable constructs, so that testing or prototyping will not be possible directly. In this case, one has to translate (manually, semi-automatically or automatically) the specification into a suitable (programming) language. This requires a major refinement of the specification before it can be animated. Note here that the specification and programming language are not necessarily separate languages, wide-spectrum languages combine the two into a single language (e.g. CIP, see [B+87]). This allows for a gradual refinement from a specification including non-executable terms to an executable program.

[4] See §3.2 for a discussion of what is meant by the term 'executable'.

A different approach to prototyping is based on algebraic specifications, where systems are described in terms of (conditional) equations on terms. Functions are defined implicitly by giving equations describing their effects, for example $pop(push(e, st)) = st$. These equations are then directed to turn them into rewrite rules. The specification is animated by applying the resulting rewrite system. Example systems of this approach are OBJ (see [GM82]) and RAP (see [Hus85, GH85]).

As has been described, prototyping can be very useful for providing some early 'hands-on' experience of a system and help to clarify the requirements on the system. The benefits that can be gained from prototyping are discussed in some detail in [HI88, §2.5]. However, prototyping as a method for validating specifications also has a number of disadvantages, including all the usual disadvantages of testing. In particular, it is very unreliable as a tool for validation, since testing only provides results for a fairly small set of input values. Furthermore, it loses at a very early stage in the development process all the under-determinedness[5] that a good specification usually allows, since a prototype will always have to choose one out of several possible output values. Additionally, any possible non-determinacy will often not be visible to the user.

1.3 Symbolic execution

Symbolic execution is a concept that was first introduced by King (cf. [Kin76]). It is based on the idea of executing a program without providing values for its input variables. The output will then in general be a term depending on these input variables, rather than an actual[6] value. This is usually described as supplying a *symbolic* input value and returning a symbolic output value.

Symbolic execution has been used for a number of different purposes, such as program verification, validation, and test case generation (see §2.1 for more details).

In the work described here, the basic ideas of symbolic execution have been extended in order to handle specifications as well as programs. This is done by introducing so-called *description values*, in addition to the usual actual and symbolic values. Description values of (program) variables are formulae that describe (usually implicitly) the value associated with this variable.

Symbolic execution can be considered as a technique for 'executing' programs when some of the information normally needed is not available. In this sense, symbolic execution allows one to handle partial information about

- input data: the input values are not determined (or at least not uniquely); this means one has to handle a whole range of input values, rather than a single value.

[5] A specification $[\![spec]\!]$ is non-deterministic, if, given any input values, the output value of executing (an implementation of) $[\![spec]\!]$ is not uniquely defined and may be different in different executions. $[\![spec]\!]$ is under-determined if, given any input values, the output value of executing an implementation of $[\![spec]\!]$ is not uniquely defined, but for any given implementation the value is always the same. Under-determinedness is thus a property of *specifications* alone, while non-determinism is a property of both specifications and programs, see [Wie88]. The terms "execution" and "implementation" used here will be defined in §3.2 and §3.3.

[6] I shall call values in the usual sense "actual values", in order to distinguish them from "symbolic values". Similarly, I shall call the usual form of execution of programs "actual execution".

- algorithm: the algorithm for computing the output value for any given input value is not provided (or at least incomplete). In this case one usually talks about a *specification* rather than a program. So far, symbolic execution has usually only been applied to programs, only in the GIST system (see §2.1.2) and the system developed by Kemmerer (see §2.1.2) has symbolic execution been extended to specifications.

- output data: the output values are not determined uniquely by the input values and the algorithm, i.e. the program or specification is non-deterministic or under-determined.

The symbolic execution system described in this thesis (called SYMBEX) is intended to be part of the IPSE 2.5 (Integrated Project Support Environment[7], cf. §1.4); it is to be used as a tool to validate a (formal) specification against its (informal) requirements, and thus to support the first step in formal software development. SYMBEX should help the user to analyse and understand a specification *before* it is implemented, by providing suitable feedback about the specified behaviour. Note that this thesis only deals with symbolic execution as a tool for *validation*, as opposed to *verification*. Therefore, it does not deal with symbolic execution as a tool for *static* analysis, providing input to other analysis tools, as done in some other symbolic execution systems (for example the Harvard PDS system described in §2.1.2). Instead, symbolic execution is used to analyse the *dynamic* behaviour of a specification. However, many of the ideas described in this thesis, in particular the formal description given in §4.1, would apply to other forms of symbolic execution as well.

The form of symbolic execution described here is intended to be used during and after the development of a specification. Symbolic execution can actually be a useful tool even *during* the specification phase, since it can be applied to incomplete specifications. This is possible since symbolic execution can deal with *names* of functions instead of their definitions (see §7.4 for more details).

A problem with using symbolic execution for checking the correctness of a specification or a program is the danger that, even though the system might *show* a mistake, such as referencing a wrong variable, the user might not *notice* it. This can happen in particular when the results of symbolic execution look too similar to the original specification. Since the user overlooked the error there, she will probably do the same again when looking at the results of symbolically executing the specification. This puts special importance on the UI, since it has to present the information in such a way that it helps the user understand a specification.[8]

Providing a *useful* symbolic execution system is made more difficult by the fact that users are different from one another and therefore find different kinds of expressions easy to understand. Thus, a 'simplification' for one user, perhaps by folding a function definition, will make the output considerably more difficult to understand for another user, who might not know the new function introduced. This implies that the system has to be highly interactive and give the user a lot of control about the information presentation, for example what simplification to apply. For this reason, 'simplification' in the following will always mean 'simplification with respect to a certain

[7]The claim that IPSE stands for "Intelligent People Stay Elsewhere" is only a vicious rumour and should not be taken seriously (I hope).

[8]The UI of SYMBEX will be discussed in more detail in §6.2

user'. §5.2 will analyse the rôle of simplification in more detail.

1.4 Symbolic execution as part of IPSE 2.5

The work described here has been done as part of the IPSE 2.5 project, the aim of which is to build an 'Integrated Project Support Environment'. It is intended to lie between the second and third generation of IPSEs as described by the Alvey Software Engineering Strategy (see [TW83]), hence the name IPSE 2.5. For an overview over the project see [DDJ+85].

Major innovative aspects of this project are genericity with respect to languages and development methods supported, and an emphasis on support for formal methods. The genericity aspect of IPSE 2.5 implies that the symbolic execution system described here has to be generic with respect to the specification language used. However, complete genericity obviously is extremely difficult to achieve. Therefore, the ideas described in this thesis apply mainly to a specification style based on a state-machine approach, describing a software system in terms of states and state-transformations.

The work done on supporting formal methods in IPSE 2.5 is described in [JL88]. This part of IPSE 2.5 which is concerned with formal reasoning is known unofficially as FRIPSE[9]. The main result of this work is the FRIPSE interactive formal reasoning tool that allows a user to build up theories and prove theorems in such a theory. See Appendix C for a brief description and some extracts from the specification of FRIPSE. The emphasis on support for formal methods led to the problem of validating the step from informal requirements to formal specifications, as described above. From the outset of the project, animation of specifications was therefore considered as an integral part of it which should help to overcome these problems. In [Kne87a], a number of different methods for animating specifications were discussed and compared. §1.2 included a short summary of that survey. For the purpose of validating specifications against their informal requirements, symbolic execution is considered the most useful method, since it provides the strongest results without making heavy restrictions on the language used.

Support for formal reasoning in IPSE 2.5 consists of three parts:

- The main part is FRIPSE, sometimes called the "RHS",[10] which is a formal reasoning tool. Its main functions are support for theorem proving, and the storage of theories in a structured way.
- The "LHS" is the link between FRIPSE and the world of specifications and programs. Its main functions are storage of specifications and programs and their relationships (such as *is-refinement-of*), and the generation of proof obligations or verification conditions in a form that can then be handled by FRIPSE.
- The third part is SYMBEX, as described in this thesis.

[9] FRIPSE has recently been renamed to μ_{ral} (pronounced mural)

[10] The distinction between "RHS" and "LHS" is due to an early diagram showing the structure of IPSE 2.5.

In future, when mentioning IPSE 2.5, this thesis usually only refers to the *formal reasoning part* of IPSE 2.5, i.e. the three components mentioned above.

As part of IPSE 2.5, the symbolic execution system SYMBEX should obviously be integrated with these other parts of this project, but on the other hand it can rely on some features to be provided by the project in general, rather than having to be developed specifically for symbolic execution. For example it is assumed that tools for theorem proving will be available, and do not have to be provided as part of SYMBEX itself. Chapter 5 considers symbolic execution as a formal reasoning task and the resulting relationship between SYMBEX and FRIPSE. §6.4 gives a brief summary of the requirements put on IPSE 2.5 by SYMBEX.

The genericity of IPSE 2.5 with respect to language and development method implies that the system has to be *instantiated* before it can be used, i.e. one has to provide a description of the languages and methods to be supported. When talking about 'IPSE 2.5' in the following chapters of this thesis, this usually refers to an *instantiated* version of a generic IPSE 2.5 system.

Note that, even though SYMBEX is developed as part of IPSE 2.5 and makes use of a lot of the features provided by IPSE 2.5, this does not mean that it can necessarily only be used together with IPSE 2.5. Any other formal reasoning tool with similar functionality would be equally appropriate. The exact requirements on such a formal reasoning tool are detailed in §6.4.

1.5 Some notes on scope and structure of this thesis

The ideas in this thesis are intended to apply to any specification or programming language that is based on the notions of states and state-transitions or, more precisely, whose semantics can be expressed in terms of states and state-transitions. Therefore, most of the ideas described are not appropriate for *algebraic* specification languages. However, within these restrictions the ideas described are intended to be fully generic. See §4.3 for a more detailed discussion of the range of languages covered.

Furthermore, this thesis ignores the problems arising from rounding errors in floating point arithmetic and from over- and underflow on computers with bounded storage capacity. The latter could be dealt with by introducing parameters expressing these bounds into the semantics of the language, using the 'clean termination' approach described in [CH79].

Chapter 2 describes previous work on symbolic execution. This includes a discussion of the ideas and motivations behind different approaches, as well as a survey of existing symbolic execution systems. Additionally, Chapter 2 describes two methods related to symbolic execution, namely partial evaluation and abstract interpretation.

Chapter 3 provides some of the theoretical background that will be used later. In particular, it discusses the semantics of specification or programming languages with an emphasis on denotational semantics. Based on that, the notions of execution and executability and the differences between specification and programming languages are examined. To provide some of the background and notation needed for a discussion of the *operational semantics* of symbolic execution, §3.4 gives a short summary of some concepts of term rewriting. Finally, §3.5 introduces the relevant relationships that may exist between denotational and operational semantics.

The main body of the thesis starts with Chapter 4, which gives a semantic definition of symbolic execution. By expressing it in terms of the denotational semantics of the language used, symbolic execution is defined generically over languages. This denotational description is followed in §4.2 by a description of symbolic execution from an operational semantics point of view, and a discussion of the relationship between the two. §4.2 includes a number of rules that can be used to describe symbolic execution of some common language constructs. Language genericity of this approach to symbolic execution is discussed in §4.3.

The operational semantics of the specification language used provide the basis for the language-generic symbolic execution system SYMBEX described in Chapters 5-7.

Chapter 5 discusses the issues that arise from the fact that symbolic execution is considered as a formal reasoning task, investigating the rôle played by simplification and describing how the operational semantics of a language as described in §4.2 can be structured as a collection of theories.

This is followed in Chapter 6 by a more detailed description of SYMBEX. This chapter starts off with a formal specification of the system, followed by a description of the user interface, and finishes with some brief notes on its implementation.

Chapter 7 contains a number of scenarios or examples showing the use of SYMBEX, which were used to derive some of the requirements on SYMBEX. In particular, it discusses some specific problem areas such as iteration and the handling of invariants.

Finally, Chapter 8 provides a short summary of the ideas discussed in this thesis and assesses the achievements and limitations of this approach. These are then compared with other work, and some suggestions are made for future work.

The appendix contains, apart from a few proofs that were too long to be included in the text itself, a short summary of some of the VDM-notation used, and extracts from the specification of FRIPSE. Although the latter does not actually contain any work done by the author of this thesis, it is included for ease of reference. Also included are a glossary of symbols and an index of definitions.

Chapter 2

Related work

"Would you tell me, please, which way I ought to go from here?"

"That depends a good deal on where you want to get to," said the Cat

Lewis Carroll

2.1 Symbolic execution

2.1.1 Motivation behind different approaches and general techniques

King in his Ph.D. thesis in 1969 was probably the first to introduce the concept of symbolic execution (see [Kin76]). Since then, a number of others have developed similar methods, but for several different purposes. Depending on the emphasis of their work, some authors therefore talk about *symbolic evaluation* or *symbolic testing* instead of symbolic execution.

Symbolic execution can be used for verifying programs in a number of different ways, these methods are described below. An alternative use of symbolic execution is the generation of test cases, also described below. Both of these applications are often based on path analysis:

Path analysis

All of the early systems for symbolic execution, such as EFFIGY or DISSECT, see §2.1.2, are based on the use of *path analysis*. When symbolically executing a program, one considers the different paths through the program separately. At any time during symbolic execution of the program, a path condition and a path value are associated with each path. The path value consists of the current (symbolic) values of the program variables, while the path condition consists of the condition on the input values under which this path is executed. Both path value and path condition are computed incrementally by symbolic execution of the program along the path. Whenever a statement changing the values of program variables is encountered, the path value is updated

accordingly. Whenever a branching statement is encountered, one branch is chosen and the appropriate branching condition is added to the path condition. The user may have to make an explicit choice which branch of a branching statement to take; alternatively, the symbolic execution system may automatically take all possible branches in turn. Obviously, the latter is in general not possible for loops, since they may give rise to an infinite number of paths. A particular path through a program and its associated choice of branches at branching statements is sometimes called a *test case*, for example in EFFIGY and in DISSECT (see §2.1.2).

If adding a branching condition to the path condition causes the path condition to become unsatisfiable, then that particular path will never be executed, whatever the input values, and the path can therefore be ignored in any further symbolic execution. This can be valuable information to the programmer since it might point to a mistake in the program. However, it does not necessarily mean that the appropriate branch is never taken, it may be taken when execution arrives at the same program point along a different path.

While path analysis can be a very useful approach for handling deterministic *programs*, it cannot be used in the same way to deal with non-executable[1] languages or languages allowing non-determinacy or under-determinedness since interpretation of specifications in such a language does not give rise to an execution path in the above sense.

Program verification and validation

There are a number of different verification and validation methods based on symbolic execution. When using Hoare-style inference rules for describing the semantics of a language, one can additionally provide assertions about input, output and loop invariants, and then generate verification conditions by symbolically executing the code between two assertions. [HK76] describes this approach in detail. This can actually be very similar to what is done in path analysis as just described, the main difference is that in order to handle infinite parts of the execution tree, in particular loops, one uses induction.

In this context, induction comes in two different forms. The first, as introduced by [Flo67], uses inductive assertions annotating a program. Such an inductive assertion states that *whenever* execution reaches the annotated point in the program, the assertion holds. This approach uses induction on the *computation.*

The second way of using induction was described in [Bur74], he uses induction on the *input data.* This seems most useful when dealing with recursively defined functions over data structures with several generators and allowing structural induction. In this approach, an assertion says that *there exists* a state during execution which is at the annotated point and satisfies the assertion. This interpretation allows one to express *termination* of a computation as well as correctness.

A shortcoming of many current formal methods approaches to software development is the fact that failure to verify a given program often does not provide enough information about the cause of the failure — is it due to an actually incorrect program, or is it perhaps due to inappropriate assertions (in particular loop invariants), or has an existing correctness proof just not been found yet? To some extent, this problem can be overcome by symbolic execution, which can

[1] See §3.2 for an exact definition of executable and non-executable languages

help to find the place in a program where it goes wrong, if this is indeed the case, or help to find appropriate assertions. This is why the symbolic execution system SELECT (cf. §2.1.2), for example, is described as a system for the *debugging* of programs. However, since the use of symbolic execution for debugging has been analysed in some detail before, this thesis will concentrate on its use for the validation of specifications, although part of this work, in particular the semantic definition of symbolic execution in Chapter 4, is of course independent of the motivation behind it.

Another technique to validate a program using symbolic execution is to annotate the program with *error conditions* in appropriate places and check for consistency with the path condition at these places. If the error condition is consistent with the path condition, then it is possible for this error to arise. This is for example done in ATTEST (see §2.1.2), where certain error conditions are generated automatically, for example for division by zero or array indices out of bounds. One could possibly generate such error conditions for all partial functions used in the program, and use these error conditions to show that the program is free of *semantic errors* in the sense of [CH79], which means that partial functions are only applied to elements of their domain. By introducing parameters for bounds on theoretically infinite types, such as $\mathbb{N}$, one can even model the size restriction on any practical computer and check for *clean termination* [CH79]. This check could be done by using error conditions to ensure that the size restrictions are not exceeded.

Using error conditions such as the ones described above, one can ensure the absence of certain errors, but of course there are some problems: first of all, consistency in general is undecidable, so that the method cannot always be applied. Second, one can only find errors from a finite predetermined set of errors. Thirdly, it usually only applies with respect to a certain path, but does not guarantee that the error will not arise when coming to the same point in the program, but via a different path.

A third way of using symbolic execution as a tool for verification uses *partition analysis*, as described in [CR84]. To apply this, one needs to have a specification and an implementation which can both be symbolically executed. Then the input domain is partitioned (using path analysis) into on the one hand *subspec domains* which are derived from the specification, and on the other hand into *path domains* which are derived from the program. This partitioning is done such that all elements within one such domain have the same path condition. By overlaying subspec domains and path domains one generates so-called *procedure subdomains*. For each of these one can then compute symbolic output from both the specification and the program, and check that the output from the program is a special case of the output from the specification.

Test case generation

Other researchers consider symbolic execution as a method of test case generation (e.g. [Inc87, CR84]). The usual way of generating test cases using symbolic execution relies on a heavily simplified version of symbolic execution, based on path conditions only. This approach consists of generating the path conditions, but not the symbolic values associated with them, and then selecting a particular value for each path considered, i.e. finding a solution to the path condition. One usually tries to ensure at least branch coverage (every branch at a conditional statement is

covered at least once) when choosing these paths. A number of methods have been suggested for choosing a particular solution that satisfies the path conditions, see for example [CR84, page 151]. In general, it is of course undecidable whether such a solution exists, even if the path condition contains only polynomials (by Matiyasevič's theorem[2]). This form of generating test cases can be extended by basing the selection of test cases on procedure subdomains, as generated by partition analysis (see above). This ensures that the selected test data characterize both the specification and the implementation.

One reason for testing a program after it has been symbolically executed is that it can never be symbolically executed in exactly the same environment (operating system, I/O devices, overflow and underflow etc) in which it will finally be implemented.

For symbolic execution of *specifications* there is a different reason, here we can derive test cases for the program from its specification. This might in some cases prove useful, in particular when the program has not been verified against its specification. This approach to test case generation is sometimes called a "black box" approach, since the resulting set of test cases does not depend on the structure of the program to be tested. Generation of test cases by symbolic execution of the program, on the other hand, is called code-dependent or path-oriented [Inc87].

2.1.2 Various systems

EFFIGY

Starting in 1973, King developed the system EFFIGY (see [Kin76, HK76, Kin80]) as a tool for program verification and validation. It supports symbolic execution of a simple PL/1-like language, and already includes most of the ideas used in later systems. In particular, it is based on the idea of path conditions, i.e. when symbolically executing a program, EFFIGY generates a path condition for every path traversed, and associates it with the symbolic or path value computed.

There are two ways of dealing with branching statements. In manual mode, the user has to decide at branching statements (`if-then-else` or `while`-loops) which of the possible branches to take. In this case, she may first save the state and come back to it later in order to explore a different branch. In automatic mode, on the other hand, all possible branches are explored. Loops are dealt with by allowing the user to specify a maximum number of computation steps. Only those paths are considered that reach an exit point within this number of steps. If, after the specified number of steps, no exit point has been reached, then that path is discarded and the system backtracks to the last branching statement with unexplored branches.

[2]Matiyasevič's theorem (also called MRDP-theorem) states that Hilbert's tenth problem is unsolvable (see [Dav73] for details):

Definition: A set $S \subseteq \mathbb{N}^n$ is diophantine if there exists a polynomial P with integer coefficients such that

$$\langle x_1, \ldots, x_n \rangle \in S \Leftrightarrow \exists y_1, \ldots, y_m \in \mathbb{N} \cdot P(x_1, \ldots, x_n; y_1, \ldots, y_m) = 0$$

Theorem: A set $S \subseteq \mathbb{N}^n$ is diophantine if and only if it is recursively enumerable.

This can be rephrased as

Corollary: There is no algorithm to decide whether or not an arbitrary polynomial with integer coefficients has an integer root.

EFFIGY also allows the user to provide assertions at various points in the program. These are then used to generate verification conditions for the program. Debugging is supported by providing tools for tracing, setting breakpoints , and state saving. EFFIGY distinguishes between 'logical' and 'actual' information: for example, symbolically executing a program that computes the factorial function, setting the maximum number of steps such that the loop can be executed three times, gives as a result the 'logical' information $n(n-1)(n-2)$ and the 'actual' information $n = 3$ (plus case distinctions for $n < 3$). Unfortunately, $n(n-1)(n-2)$ is immediately 'simplified' to $n^3 - n^2 - 2n^2 + 2n$ (without user intervention). This shows the need for interactive, user-dependent simplification as argued in the introduction of this thesis.

GIST

GIST is a specification/'automatic programming' system being developed by Balzer and his colleagues at USC/ISI. One of the features provided by GIST is its symbolic execution facility (see [BGW82, CSB82, Coh83]).

The overall idea of the GIST project is to build a programming environment based on transformations. First, a system is specified in the GIST specification language, in terms of objects and relations between them. It is then implemented by gradually transforming it into a LISP program. Some of the transformations used for this are done fully automatically, others require the programmer to choose from a library, e.g. "do you want this set to be implemented as a list or a tree?". If the specification is changed at a later stage, part of it can be reimplemented fully automatically.

Since GIST specifications are expressed in terms of entities and relationships, they can neither actually nor symbolically be executed in the usual sense. The notion of paths and path conditions that can be used for programming languages cannot be applied in this case. This same problem will have to be faced by SYMBEX as well, it too has to deal with specification languages for which the notion of paths is not applicable in the usual way. GIST solves this problem by considering a specification as a set of axioms in a first order temporal logic. Symbolic execution then generates simple new theorems from these axioms, using a 'Forward Inference Engine' called FIE. Using a set of heuristics (described in [Coh84]), these new theorems are then examined to decide whether they are 'interesting'. Only the interesting ones are displayed to the user.

GIST is supported by a paraphraser which translates GIST specifications into English in order to make them easier to understand (see [Swa82]). This is then extended (see [Swa83]) to explain the results of symbolically executing a GIST specification. Without the 'behavior explainer', the output from the symbolic evaluator is rather difficult to read and understand, but with it, the symbolic evaluator seems to be a very useful tool for validating specifications.

After the work described above was finished, work on GIST stopped for several years and has only recently (in 1987) started again. Unfortunately, after these recent changes some of the different parts of GIST do not fit together any more; in particular, the behavior explainer cannot now handle output from the symbolic evaluator. This recent work concentrates on support for transformations in program development, symbolic execution is no longer a central part of the system.

Work by Kemmerer, Rudnicki and Eckmann

In [Kem85], Kemmerer describes a system for symbolically executing specifications, and compares this approach with the use of prototyping tools for specification validation. A simple version of this symbolic execution tool has been built, which can be used for symbolically executing specifications in INA JO.[3] The INA JO specification language is a non-procedural language that models a system as a state machine, using the language of first-order predicate calculus. State transitions are described by post-conditions which are called transforms; these describe state transitions by specifying the values of state variables after the transition in terms of their values before the transition. State transitions cannot have pre-conditions as such, one achieves a similar effect using conditional expressions. One can define state invariants, called criterions, that have to be satisfied at any stage. Contrary to the view currently taken in VDM [Jon86], one has to *prove* that the criterions are met after each transform, this is not considered as part of the type definition.

When talking about the requirements on a system, Kemmerer distinguishes between a *formal model* which expresses the *critical* requirements on a system, and the *functional requirements*, which describe those requirements that are still necessary, but not as critical and are not themselves expressed in the specification. Instead, the functional requirements should be derivable from the specification. A functional requirement is then called valid with respect to a specification, if every possible implementation of the specification satisfies it. Similarly, a requirement may be satisfiable or unsatisfiable. 'Functional requirements' could be described as symbolic test cases, and symbolic execution is used to check whether a particular functional requirement is valid, satisfiable or unsatisfiable.

Symbolic execution of a specification given by a state transition starts from a number of initial assumptions describing the starting point of the functional requirements and results in a predicate describing the state of the system after the transition. This is essentially an instantiated version of the transform associated with the specification. One then tries to derive the result of the functional requirement from this predicate.

[Rud87] is intended as a follow-up to [Kem85], in this paper Rudnicki discusses the relationship between symbolic execution as described by Kemmerer and theorem proving as two methods for validating specifications. He emphasizes the need to prove certain properties of a specification, in particular the preservation of criterions (invariants). Since he only considers the form of symbolic execution based on path analysis but not those forms used for verification and proving theorems about a program (cf. §2.1.1), he emphasizes that symbolic execution (or symbolic testing, as he calls it) is not sufficient. In his opinion, proofs of properties of a specification should be done *by hand* rather than automatically (but supported by a proof checker), since failure to prove a statement using an automatic theorem prover does not in itself give many clues to *why* the proof failed — an argument which leaves out the possibility of an interactive theorem prover that helps the user to find a proof.

Next, Rudnicki suggests a symbolic execution strategy that seems to correspond roughly to a branch coverage strategy in testing:

> "The minimal symbolic testing of a transform consists of testing each change of the

[3] INA JO is a trademark of System Development Corporation, a Burroughs Company.

> state variables which can be caused by the transform at least once. The important thing is that each possible change of the state variable should be tested by a symbolic run starting in the initial state." [Rud87, page 192]

The initial state mentioned here is defined as part of the specification. With this strategy, one therefore has to build, for every change of state variables in a transform, a sequence of transforms starting in the initial state such that it reaches the relevant part of the transform.

In [KE85], Kemmerer and Eckmann describe a different approach to symbolic execution, which was implemented in the UNISEX-system. The original version of UNISEX was implemented by Solis as part of his master's thesis [Sol82]. UNISEX is a system for symbolically executing PASCAL programs, which provides two modes, *verify* mode and *test* mode. Symbolic execution in verify mode is based very much on the ideas described in [HK76] for verifying programs (cf. §2.1.1). To verify a PASCAL program, the user has to annotate it with a number of assertions, including at least an entry and an exit assertion, plus an assertion for each loop (the loop invariant). UNISEX then generates the relevant proof obligations by symbolically executing the code between two assertions. This can be driven either manually, in which case the user has to decide which branch to take at a branching statement, or automatically. In the latter case, UNISEX covers all branches by pushing the false branch on a stack and continuing along the true branch. When the end of the path is reached at an assert or exit statement or the end of the program or subroutine being verified, then another branch is popped from the stack and executed symbolically.

In test mode, UNISEX uses path analysis to generate path conditions and path expressions. No assertions are needed, and even if they are provided, paths end at the end of the program rather than at the next assertion. Output from UNISEX is mainly intended to show the user the behaviour of the program and thus *validate* it, rather than formally verify the program against some assertions.

In a future version, a theorem prover is to be added to UNISEX which checks at a branching statement whether the path condition implies that only one branch can possibly be taken, and which is also used to prove the proof obligations generated. In the version described in [KE85], the user has to take the place of the theorem prover and answer the relevant questions from the system.

The expression language of UNISEX used for assertions consists of the expression language of PASCAL plus the additional keywords `forall`, `exists` and `implies`. Assertions are provided as comments in the program to be executed symbolically, so that no editing is needed before the program can be compiled. The programming language supported is a sub-language of PASCAL as defined in [JW75], excluding for example subroutines with side-effects.

DISSECT

DISSECT (see [How78a]) is another system for symbolic execution which uses path conditions.[4] It tries to solve them by trying to find an actual value as "test case". DISSECT has been implemented in LISP and can be used to symbolically execute FORTRAN programs; the implementation

[4]This is the reason why the system is called DISSECT, it allows the user to 'dissect' a program and analyse the different paths separately.

of DISSECT was completed in 1976.

The DISSECT symbolic evaluator takes as input a FORTRAN source program and a file with DISSECT commands that are to be applied to the program. There are three kinds of commands:

- input commands assign actual or symbolic values to variables
- path selection commands determine which branch to take at a branching statement, or how many times to execute a loop
- output commands are used to print out the values of program variables

Additionally, these commands can be combined in various ways, for example using conditionals. Every command is associated with a particular line in the FORTRAN program.

Every symbolic evaluation then effectively explores one path (called 'test') through the program, by determining its path condition and expressing the values of program variables at any point in the program in terms of the values assigned via input commands. It is possible to combine different paths (e.g. both branches of a branching statement), but these are effectively handled as different symbolic evaluations.

DISSECT runs in batch mode rather than interactively. According to [How78a] this is a deliberate choice, since the selection of these paths or tests has to be done carefully and therefore would be difficult to do interactively.

In [How78b], Howden analyses a number of methods for the validation of programs. There he applies these methods, which include various test methodologies and symbolic execution, to six short programs in different languages (COBOL, PL/1 and ALGOL) containing bugs. His conclusion is that symbolic execution helps to find about 5% of errors in addition to those found by combining various methods of testing, and is a "natural" way of discovering errors for about 10–20% of all errors. The errors found by symbolic execution are mainly those where a wrong variable is referenced.

However, these results cannot really be applied to the work described in this thesis, since Howden only considers imperative programming languages, and only considers short programs, four out of the six programs investigated contain less than 30 lines of code, the longest one contains about 450 lines of COBOL code.

Dannenberg and Ernst's system

The paper [DE82] by Dannenberg and Ernst describes another system for symbolic execution which is also based on path conditions. It grew out of a project to design and implement a mechanical verification condition generator, but it is not clear from [DE82] whether this system has actually been implemented.

Dannenberg and Ernst use inference rules to describe the semantics of a small imperative programming language. In addition to the usual constructs, this language contains a `confirm` construct for expressing assertions and a `maintaining` argument in `while`-loops for expressing invariants, which are essentially a special case of assertions. Statements using `confirm` or `maintaining` are used as a specification of the program. The basic unit of the inference rules

is a statement of the form '$S, PC \backslash A$' which expresses the correctness of the statement list A with respect to its specification, given a state S and initial path condition PC.

Using these inference rules, Dannenberg and Ernst then propose to symbolically execute the code in between two assertions to generate verification conditions. They use symbolic execution solely as a tool for verification condition generation, but not for example for directly generating information about the state and the path condition that is then fed back to the user.

An important feature of their work is that it shows a possibility of handling functions with *side effects*. For this purpose, they introduce attribute grammars and describe the inference rules describing symbolic execution as production rules in the grammar, where the state and path condition are represented as attributes. They also give rules for handling more complicated language constructs, such as multiple-exit loops and procedures with multiple exits. Unfortunately, they do not explicitly address the problems of correctness and completeness of their rules.

ATTEST

The ATTEST system (see [Cla76, CR84]) is a symbolic execution system mainly intended for generating test data for FORTRAN programs. It uses path conditions by treating them as a system of constraints. If the path condition only contains linear constraints, then ATTEST solves it, using a linear programming algorithm. The result is then used both for identifying (un)feasible paths and for generating a set of test cases satisfying branch coverage. [Cla76] claims that this actually covers most cases, that most constraints in practice actually are linear.

ATTEST is not only meant for test data generation, but for program validation in a more general sense. This is why ATTEST actually builds up a symbolic representation of the program output associated with each path condition, which would not strictly be necessary for test data generation. ATTEST also helps to generate error conditions (e.g. for division by zero) and check them by adding them to the path condition and checking for consistency, as described in §2.1.1. However, since this consistency check is also based on the linear programming algorithm, it can only handle linear path and error conditions.

Loops are handled by trying to make their effect explicit by first expressing variable values recursively in terms of values on previous iteration (using so-called recurrence relations), and then eliminating this recursion, i.e. solving the recurrence relation. This is done by introducing a new variable denoting the number of executions of the loop. The result is then used to create a loop expression which replaces the loop itself. Unfortunately, in most cases this will not work, since many functions can only be expressed using recursion or loops. In these cases, ATTEST cannot handle the loop (see [CR84, page 147]). Instead, the user has to explicitly specify the number of iterations of the loop. Furthermore, like most systems ATTEST can only handle a single path at a time, at branching statements the user has to make an explicit choice about which branch to take.

SELECT

SELECT is a symbolic execution system developed by a group working at SRI (see [BEL75]). The main purpose of SELECT is the debugging of programs. It is intended to complement mechanical program verification and overcome some of the (theoretical and practical) problems associated

with this approach. SELECT is built from the expression-simplifier part of the SRI Program Verifier by adding facilities for symbolic execution of programs written in a subset of LISP.

Like most other systems, SELECT is based on the notion of path conditions. At branching points, the system considers all feasible branches and the appropriate predicates are added to the path conditions. For loops, the user decides on the maximum number of iterations she wants the system to take. For each path, SELECT tries to maintain an example input, i.e. a solution to the path condition. This solution can then be used as actual test data for the program. Several different algorithms have been tried for solving the path conditions, which all handle *linear* equalities and inequalities, plus a few special cases.

In addition to generating (simplified) symbolic values of program variables and actual input test data, SELECT also allows the user to annotate a program with assertions; according to [BEL75], these can serve as

- executable assertions, i.e. the assertion is a program in itself that is executed when the appropriate position in the program is reached ('Assertion' does not seem to be an adequate name for such annotations).
- constraints, which are simply added to the path condition. This enables the user to ensure that the test data generated satisfy some additional conditions.
- checkpoints; when execution reaches such a checkpoint, the negation of the assertion is added to the path condition, and the system checks the result for consistency by trying to generate a solution to it, using the general solver for path conditions.

Harvard Symbolic evaluator

The symbolic evaluator is a central part of the Harvard Program Development System PDS. PDS supports programming in the EL1 language, an imperative programming language which supports constructs such as records, pointers, recursive procedures, and several ways of sharing of variables, in addition to the usual assignments, loops etc.[5]

The basic idea behind the system is to derive semantic information about a program separately from its use in any tools such as those for exception detection, program verification and validation. The advantage of this approach is that semantic analysis is only performed once, instead of each tool performing its own analysis. Additionally, this guarantees that all tools assign the same semantics to the constructs of the programming language while, on the other hand, it is easier to adapt the tools to a different programming language, since only one analyser has to be rewritten.

This semantic analysis is based on symbolic evaluation, and the results of the analysis are then stored in a "program database". Other tools can use this information to reason about the program without having to re-analyse the program, and can add further information which they deduce to the data base. Note that with this approach symbolic evaluation is used as a *static analyser*, not a *dynamic interpreter*.

[5] See [Plo84] for a description of the rôle that symbolic evaluation plays within PDS, and [CHT79] for the technical details of the symbolic evaluator.

[CHT79] describes in some detail the simplification of expressions in the context of symbolic evaluation. However, the emphasis on symbolic evaluation providing input for further analysis by other tools, not by humans, means that a 'simpler' expression might be considerably more difficult to *read* for the human user. The simplifier generates suitable normal forms of expressions, based partly on CNF.

Loop analysis is based on different variations of solving recurrence relations. This involves identifying the number of iterations of a loop, and expressing the values of variables after any one iteration in terms of their values after the previous iteration. From this, one then tries to 'solve' the recurrence relation, i.e. transform the recursive equation into an explicit one. If this is not possible, one 'forces' a solution by creating a particular form of lambda expression, similar to description values as used in this thesis. These recurrence relations also help to generate loop invariants automatically, at least in some easier cases.

Based on the results of semantic analysis using the symbolic evaluator, a number of other tools, such as a verification condition generator and a source-to-source optimiser, have been built. However, in about 1985 the project was abandoned and replaced by a new project called E-L. The main reason was that PDS had only been implemented on a PDP-10, and rather than just port PDS to a new machine, it was decided to start again from scratch. An additional reason was that they wanted to introduce a new approach to software development, based on the use of transformations. This new system does not yet support symbolic execution, but may do so at a later stage.

REDUCE

REDUCE, as described in [AGPT85], is a system for program reduction based on symbolic execution, as introduced by [Kin80]. This is a program transformation technique for removing superfluous parts of a given program while leaving the original structure intact. For example, if a program contains a conditional with condition $x > 0$, and it subsequently becomes known that the program will only ever be used on input values such that $x > 0$, then the conditional may be replaced by its `then`-part, the `else`-part may be removed. REDUCE supports the reduction of such conditionals for a class of functional languages. Its main achievement, according to [AGPT85], is that it allows "symbolic constants which are assumed to denote subsets of data domains" which allows one to express certain constraints on the input domain. These 'symbolic constants' are predicates which are used to express input and path conditions. However, this claim seems not quite justified since other systems, such as EFFIGY or UNISEX, do allow a user to add assertions on the input domain, with a similar effect.

REDUCE does support a certain amount of genericity with respect to the language handled, it can symbolically execute a whole class of functional languages. This is achieved by defining the "symbolic semantics" of a language using predicate transformers [Dij76] on the symbolic constants.

However, in the opinion of the author of this thesis, it is not appropriate to call the technique used in REDUCE "symbolic execution" and it certainly does not fit the semantic model given in Chapter 4. This is because it does not analyse the relationship between individual input and

output values, for a set of input values. Instead, it analyses the relationship between *sets* of input values and *sets* of output values, and is therefore much closer in spirit to the technique of *abstract interpretation* as described in §2.3.

2.2 Partial evaluation

Partial evaluation (also called partial computation) and mixed computation are program transformation methods making use of partial knowledge about the input data to the program. In partial evaluation one does those computations that do not depend on *run time* values and can therefore be done at *compile time*. Another way of saying this is that partial evaluation is used for processing the *static* semantics of a program. In [EJ87], Ershov defines partial evaluation and mixed computation by splitting up the input data into two separate components d_1 and d_2, as follows:

Definition 2.2.1 (Partial evaluation, mixed computation) *Let P be programs, let D be data, and Sem: $P \times D \rightarrow D$ be the functional semantics. Then partial evaluation is a function Part: $P \times D \rightarrow P$ such that*

$$Sem(p,(d_1,d_2)) = Sem(Part(p,d_1),d_2).$$

Mixed computation is a function Mix: $P \times D \rightarrow P \times D$ such that

$$(p',d') = Mix(p,d) \Rightarrow Sem(p,d) = Sem(p',d')$$

It follows that partial evaluation is a special case of mixed computation. The idea of partial evaluation is based on Kleene's s–m–n theorem (see e.g. [Cut80, §4.4]). It has been used for a variety of different purposes, including e.g. program simplification and compiler generation.

Alternatively, partial evaluation can be described as a form of specialisation (as done in [EJ87]), since it takes as input a program p and partial description d of the input data to p and returns a specialised program p_d which is equivalent to p on all data satisfying d. An area of particular research interest is the self-application of this specialisation process, where one considers programs themselves as data. If one can program the specialisation process in a program *Special*, then *Special* is called an *autoprojector* [Ers82]. In this case, $Part(Special,p)$ is itself a program that takes a partial description d and returns the specialised version p_d of p, and $Special_p$ is a generator of specialisations of program p.

Some standard examples are

- use a general context-free parser *Parser* and a fixed context-free grammar G to get a parser $Parser_G$ for G
- let *Int* be an interpreter for a language $\mathcal{L}$. *Int* takes as input an $\mathcal{L}$-program p and some data d, and runs p on d. Then

  $$Sem(Int,(p,d)) = Sem(Part(Int,p),d)$$

  i.e. $Part(Int,p)$ is a target program, namely the compiled version of p, and $Special_{Int}$ is a compiler for $\mathcal{L}$

- let *Int*, *p* and *d* be as before. Then because of

$$Sem(Special, (Int, p)) = Sem(Part(Special, Int), p)$$

Part(*Special*, *Int*) is a compiler for the language interpreted by the interpreter *Int*, and $Special_{Special}$ is a compiler generator that transforms interpreters into compilers.

These examples are the main impetus behind the development of partial evaluation, most of the work so far has been done on compiling and compiler generation. The use of partial evaluation for compilation and compiler generation was first described in [Fut71] and has since been implemented in several different systems. The use of partial evaluation for generation of compiler generators was probably first described in [Tur79], a simple implementation of it is described in [Ses85].

Both symbolic execution and partial evaluation are characterised by the fact that, given a program and some, but not complete, information on the input data, one tries to extract as much information as possible. However, in partial evaluation this information is used *before* the program is executed, in order to transform it into a new and more efficient program, which is functionally equivalent to the original one for all input data satisfying the information given.

In symbolic execution, on the other hand, this information is only provided at run-time, too late to be used for partial evaluation. It is then used to derive as much information as possible about the output of the program (including possible side effects), but without changing the program itself. This makes it possible to add new information during execution, or to execute the program repeatedly using different information.

Of course, these advantages have to be paid for: in symbolic execution, adding new information will often slow down the system, while the program generated by partial evaluation will usually be faster the more information is given.

A large amount of work has recently been done on partial evaluation (see e.g. [BEJ87]), mainly in Japan, USSR and Denmark. So far, most work on partial evaluation has been done for simple declarative languages such as pure LISP, pure PROLOG or λ-calculus. This seems to be because these languages have a rather simple operational semantics, and most partial evaluation techniques are based on the operational semantics of the language dealt with.

2.3 Abstract interpretation

Abstract interpretation is another method for handling partial information; it is based on the idea of mapping input data into a different, 'more abstract' domain and performing the computation there. This will often be much easier to do and still provides sufficient information for certain purposes. In particular, it is often used to determine certain properties of programs without actually executing them, e.g. for type inference/type checking (see [CC77b]) or strictness analysis (see [Myc81] or [Pey87, §22]).

Another way to view abstract interpretation is to say that it provides (usually at compile time) information about the context of a program (fragment) which enables one to guarantee

the correctness of some program transformation in this context. When viewed this way, abstract interpretation is closely related to partial evaluation: it provides the (partial) information which is then used for partial evaluation.

One of the first to describe abstract interpretations was Sintzoff (see [Sin72]), after that the subject was mainly developed by Patrick and Radhia Cousot (see [CC77a, CC77b]), who used it for analysing flowchart programs. [Myc81] extended these ideas to functional languages and mainly used them for strictness analysis. (A function

$$f: D_1 \times \ldots \times D_i \times \ldots \times D_n \to C; (d_1, \ldots, d_i, \ldots, d_n) \mapsto f(d_1, \ldots, d_i, \ldots, d_n)$$

is strict in d_i, if for all $d_1, \ldots, d_{i-1}, d_{i+1}, \ldots, d_n$: $f(d_1, \ldots, \perp, \ldots, d_n) = \perp$.) After that, abstract interpretation was developed a lot further by a number of different researchers (see e.g. [GJ85] or [AH87]), who mainly used it for optimising compilers for functional languages, in particular for strictness analysis.

Knowledge about strictness of functions is mainly used in compiling functional languages. If a function is strict, then call-by-need can safely be replaced by call-by-value. According to [Hug85], this can lead to an improvement in performance by a factor of about three to five. Additionally, it helps to make use of parallelism, since for a strict function it does not matter whether the argument is evaluated before or when it is needed.

As an example, consider the multiplication

$$(-357) * 1078 \tag{2.1}$$

Instead of actually evaluating this term, it might (for a particular application) be enough to abstract it and use the 'rule of signs'

$$(-) * (+) = (-) \tag{2.2}$$

to infer that the result of the multiplication is negative. In other words, we 'abstract away' from the specific values and perform the computation in a new universe of abstract objects (−), (+) and (0). One way of describing this is to say that abstract interpretation provides an alternative denotational semantics for a language, based on a simpler abstract domain than the standard one.

An alternative to using the rule of signs would be to use the rule

$$odd * even = even \tag{2.3}$$

to infer that the result must be even. If we want to know the number of digits of the result, we can use the rule

$$(3\text{-}digit\text{-}number) * (4\text{-}digit\text{-}number) = (6\text{- } or \text{ } 7\text{-}digit\text{-}number) \tag{2.4}$$

In this last example it is no longer possible to give an exact answer. Given the sizes (in number of digits) of the factors, it is not possible to uniquely determine the size of the product without further information.

The rules described above are often used as *checks* of the correctness of a computation. Another way of looking at them would be to consider them as rules for (sub-) type inference. In the following, we describe the general concepts of abstract interpretation in terms of lattice theory, mainly based on [Myc81].

Definition 2.3.1 (Adjoined functions) *Let $(L, \sqsubseteq_L)$ and $(M, \sqsubseteq_M)$ be complete lattices. We say that the functions Abs: $L \to M$ and Conc: $M \to L$ are* adjoined *if they are monotonic and for all $l \in L$, $m \in M$*

$$Abs(l) \sqsubseteq_M m \Leftrightarrow l \sqsubseteq_L Conc(m)$$

They are exactly adjoined, *if they are monotonic and for all $l \in L$, $m \in M$*

$$Conc(Abs(l)) \sqsupseteq_L l$$

$$Abs(Conc(m)) = m$$

The intuition behind this definition is that adjoinedness makes elements from L and M comparable under the partial orders. Exact adjoinedness ensures that taking a concrete example of an abstract element and abstracting it again returns the original abstract element.

For a partially ordered set S, let $\sqcup S$ denote the *least upper bound* of S, let $\sqcap S$ denote the *greatest lower bound* of S.

Lemma 2.3.2 *a) If Abs and Conc are adjoined then*

$$Conc(m) = \sqcup \{l \mid Abs(l) \sqsubseteq_M m\}$$

$$Abs(l) = \sqcap \{m \mid l \sqsubseteq_L Conc(m)\}$$

b) If Abs and Conc are exactly adjoined then

$$Conc(m) = \sqcup \{l \mid Abs(l) = m\}$$

c) If Abs and Conc are exactly adjoined then Abs is surjective and Conc is injective.

Given lattices L and M and a function *Abs*, this lemma provides a test whether there exists a function *Conc* s.t. *Abs* and *Conc* are (exactly) adjoined. Note that, even if *Abs* and *Conc* are exactly adjoined, neither $Abs(l) = \sqcup\{m \mid l = Conc(m)\}$ nor $Abs(l) = \sqcap\{m \mid l = Conc(m)\}$ need to hold, since l may not be in the range of *Conc*.

The first example above can be translated into this model if we let L be the powerset lattice of $\mathbb{Z}$ and M be the powerset lattice of $\{(+), (-), (0)\}$.[6] Multiplication on $\mathbb{Z}$ and on $\{(+), (-), (0)\}$ is defined in the obvious way. Then define

$$Abs(\{x\}) = \begin{cases} (+) & \text{if } x > 0 \\ (-) & \text{if } x < 0 \\ (0) & \text{if } x = 0 \end{cases}$$

$$Abs(s_1 \cup s_2) = Abs(s_1) \cup Abs(s_2)$$

Alternatively, we could define L to be the lattice containing $\{\}, \mathbb{Z}$ and all finite subsets of $\mathbb{Z}$. In this case *Abs* and *Conc* are still adjoined, but no longer *exactly* adjoined, since for example

$$Abs(Conc(\{(+), (0)\})) = Abs(\mathbb{Z}) = \{(+), (0), (-)\}$$

We can now define what it means for a function to abstract some other function:

[6] For the above example it would actually be enough to let L be the lattice containing $\{\}, \mathbb{Z}$ and all one-element subsets of $\mathbb{Z}$ (with ordering $\subseteq$), and similarly let M be the lattice containing $\{\}, \{(+)\}, \{(-)\}, \{(0)\}, \{(+), (-), (0)\}$ (also with ordering $\subseteq$).

Definition 2.3.3 (Abstraction) *a) Given two lattices L and M, we say that M abstracts L if there exist functions Abs: $L \to M$ and Conc: $M \to L$ which are exactly adjoined.*

b) Let L_1, L_2 be two lattices, and let M_1, M_2 be lattices abstracting L_1, L_2 respectively via functions Abs_i and $Conc_i$. Let g be a function $g: L_1 \to L_2$ and h be a function $h: M_1 \to M_2$. We say that h abstracts g (or, strictly speaking, (h, M_1, M_2) abstracts (g, L_1, L_2) w.r.t. Abs_i and $Conc_i$) if for all $l \in L_1$

$$Conc_2 \circ h \circ Abs_1(l) \sqsupseteq_{L_2} g(l)$$

Note that one does not require equality between $Conc_2 \circ h \circ Abs_1$ and g, only inclusion, since otherwise h might be non-computable. One therefore demands this *safety* property instead, which says that the result of applying the abstracted function is allowed to be too large, but never too small.

How can one find the abstraction h of a function g, given the lattices L_1, L_2, M_1, M_2 and the functions Abs_i, $Conc_i$ as described in the above definition? To do this, one has to replace all constants and basic functions in the body of the definition of g by their abstract versions. Variables remain unchanged.

If g is recursively defined as $g = F(\ldots, g, \ldots)$, then this replacement results in a recursive definition of h, namely $h = F^{\#}(\ldots, h, \ldots)$, where $F^{\#}$ denotes the abstracted version of the functional F. If the functional $F^{\#}$ is continuous, then h can be defined as its fixpoint $\sqcup\{(F^{\#})^i(\ldots, \bot, \ldots) \mid i \in \mathbb{N}\}$.

A standard example of the application of abstract interpretation is strictness analysis: for this one uses the abstraction (cf. [Pey87, §22])

$$\begin{aligned} abs\,\bot &= 0 \\ abs\,x &= 1 \quad \text{if } x \neq \bot \end{aligned}$$

Then f is strict iff $Eval[\![f(\bot)]\!] = \bot$, which is true iff $abs(Eval[\![f(\bot)]\!]) = 0$.

One now has to find a suitable abstract interpretation $Eval^{\#}$ of $Eval$. This should satisfy the *safety* property

$$abs(Eval[\![E]\!]) \sqsubseteq Eval^{\#}[\![E]\!]$$

for any expression E. In this case, the safety property demands that abstract interpretation should never suggest that a function is strict if this is not the case.

Let f be defined as $f(p,q,r) \triangleq$ **if** $p = 0$ **then** $q+r$ **else** $q+p$. Since $a+b$ is defined iff both a and b are defined, we have $+^{\#} = \cdot$. Similar for = and **if-then-else**. 0 is always defined and therefore $0^{\#} = 1$. Then a suitable abstract interpretation would be

$$\begin{aligned} f^{\#}(p,q,r) &= Eval^{\#}[\![\textbf{if } p = 0 \textbf{ then } q + r \textbf{ else } q + p]\!] \\ &= \textbf{if}^{\#} p =^{\#} 0^{\#} \textbf{ then}^{\#} q +^{\#} r \textbf{ else}^{\#} q +^{\#} p \\ &= \textbf{if}^{\#} p \cdot 1 \textbf{ then}^{\#} q \cdot r \textbf{ else}^{\#} q \cdot p \\ &= (p \cdot 1) \cdot ((q \cdot r) \vee (q \cdot p)) \\ &= p \cdot q \end{aligned}$$

where $a \vee b$ is defined as 1 if at least one of a and b is 1, and 0 otherwise.

Then

$$\begin{aligned} f^{\#}(0,1,1) &= 0 \\ f^{\#}(1,0,1) &= 0 \\ f^{\#}(1,1,0) &= 1 \end{aligned}$$

and f is strict in p and q.

Chapter 3

Theoretical background

> They gave me courses on love, on intelligence, most precious, most precious. They also taught me to count, and even to reason. Some of this rubbish has come in handy on occasions, I don't deny it, on occasions which would never have arisen if they had left me in peace.
>
> *Samuel Beckett: The Unnamable*

3.1 Language semantics

Consider a specification or programming language $\mathcal{L}_1$. Unless otherwise stated, I shall in future consider a program as a special kind of specification. The differences between the two will be discussed in §3.3. Specifications are a certain class of $\mathcal{L}_1$-terms, usually containing free variables called input and output variables and state variables. §3.3 will say more about what distinguishes specifications.

For simplicity, I shall from now on assume that specifications only use a single state variable, but no other input or output variables. Since the state variable might be of arbitrarily complex type, this is no real restriction of generality.

The semantics of a language describe the 'meaning' of terms of the language in some way. Here the following definition of meaning will be used:

Definition 3.1.1 (Meaning of specifications) *The meaning of a specification* $[\![spec]\!]$[1] *is a relation R between input and output states.*

Non-termination or abortion is described by the output state $\perp$, *i.e.* $R(\sigma, \perp)$ *describes the fact that, starting from state* σ, *execution of* $[\![spec]\!]$ *may not terminate or abort. We require that* $R(\perp, \perp)$, *and* $R(\perp, \sigma)$ *only if* $\sigma = \perp$.

[1] Terms in the object (or specification or programming) language are written in Strachey brackets $[\![...]\!]$, in order to distinguish them from terms in the (meta-) language used for describing the semantics of the term

We also require that for every state σ_1, there exists at least one state σ_2 (possibly $\perp$) s.t. $R(\sigma_1, \sigma_2)$.

For deterministic programs, R will actually be a function from states to states rather than a relation.

The requirement $R(\perp, \perp) \wedge (R(\perp, \sigma) \Rightarrow \sigma = \perp)$ was introduced to ease the description of composition. It essentially describes the fact that if a specification never starts to be interpreted[2] because the one interpreted previously fails to terminate, then interpretation of this specification also will not terminate. Equivalently this could be expressed as $R(\perp, \sigma) \Leftrightarrow \sigma = \perp$.

The reason for using $\perp$ as a state in its own right, rather than for example introducing a termination set T of states, as done in VDM, is that this will make it easier to describe symbolic execution and to distinguish between non-termination of symbolic execution itself and termination with the result "execution does not terminate". Additionally, composition is easier to describe this way. However, provided two relations R are considered equivalent if they agree on all pairs of states whose first element is in T, then this is only a matter of taste and the two models are isomorphic: given R as above, (R_1, T_1) can be defined as $R_1(\sigma_1, \sigma_2) = R(\sigma_1, \sigma_2) \wedge \sigma_2 \neq \perp$ and $T_1 = \{\sigma \mid \neg R(\sigma, \perp)\}$. Conversely, given (R_1, T_1) we can define $R(\sigma_1, \sigma_2) = R_1(\sigma_1, \sigma_2) \vee (\sigma_1 \notin T_1 \wedge \sigma_2 = \perp)$.

There are a number of different ways of describing the semantics of a programming or specification language, the most common ones are (cf. [Sch86, pages 3f] or [Sto77, §2])

operational semantics: the meaning of a construct of the language is given by explicitly stating its effect, the operation that it evokes (see for example [Plo81]). Given an input state for a specification, the operational semantics of the language provides an algorithm to find the appropriate output state.

Another way of describing operational semantics views only input variables as free variables of a specification. In this case one substitutes the input data for the free variables of the specification term, and then rewrites the resulting ground term into normal form in a rewrite system which is given by the semantics of the programming language (cf. [Sch86, §10.7]). This normal form is then the output from executing the specification. For example, the semantics of λ-calculus can be given this way, using β-reduction etc.

denotational semantics: the meaning of a construct of the language is described by giving it a 'denotation', i.e. by translating it into a different structure and modelling the effect of statements of the language there. This different structure is usually, but not necessarily, based on domain theory as introduced by Scott (see [Sto77, Sch86]). One possible alternative is to express the denotations in the language of predicate calculus, this is called *predicative* semantics (as introduced in [Heh84]).

axiomatic semantics: the meaning of a language is described by axioms that can be used to prove theorems about (specification) terms in the language. These axioms act as constraints on

[2]From now on, I shall distinguish between *interpreting* a specification and *executing* it. Interpreting a specification transforms one state into another according to the meaning of the specification. Executing it additionally requires that one has an algorithm for performing this transformation. These issues will be discussed in more detail in §3.2.

the relation between input and output variables. The usual style for such axioms is Hoare logic (cf. [Hoa69, Apt81]), using input and output assertions. A similar approach is the use of predicate transformers and weakest preconditions, as introduced by Dijkstra in [Dij76].

The following is mainly based on denotational semantics. By giving a denotational semantics to a language $\mathcal{L}_1$, one translates it into another language $\mathcal{L}_2$, called semantic language, which is considered to be 'understood', i.e. the meaning of its constructs is known. In other words, one explains the semantics of $\mathcal{L}_1$ in terms of the semantics of $\mathcal{L}_2$. The translation is given by a recursive function from terms in the language $\mathcal{L}_1$ to terms in $\mathcal{L}_2$. This translation is called *valuation* function. Usually, $\mathcal{L}_2$ will have some theory associated with it, in that case it will be more adequate to say that we understand the *theory* associated with $\mathcal{L}_2$, rather than the language itself. Common choices for $\mathcal{L}_2$ with an associated theory are the languages of Scott's domain theory, of predicate calculus, of partial recursive functions, or of λ-calculus. In the following, the language of LPF (cf. Appendix B.2), the logic of partial functions, is used.

Definition 3.1.2 (Valuation functions) *A valuation function $\mathcal{M}$ maps terms of a language $\mathcal{L}$ to their meaning (denotation), an element or set of elements of the abstract or semantic domain. We require that valuation functions are defined structurally, i.e. the meaning of a term is defined in terms of the meaning of its subterms.*[3]

The valuation function may map to a *set* of elements of the abstract domain in order to handle non-determinism and under-determinedness. Alternatively, one might introduce power domains instead.

A valuation function may also take additional arguments such as the environment or continuations, in order to handle more complicated language constructs. This will in the following be handled by 'currying' the valuation function and turning the denotation of the construct itself into a function. In particular, the denotation of a program term is usually defined as a function from states to states. Variations are used for non-deterministic programs, whose meaning is often given as a binary relation between states or, equivalently, a function from states to sets of states, and for under-determined programs, whose meaning is often given as a set of functions from states to states.

Usually, one introduces several different valuation functions for different classes of terms, such as commands, Boolean expressions etc. The valuation function for terms in class C will be written as $\mathcal{M}_C$. Figure 3.1 describes the valuation functions on predicates and specifications. Specifications are terms in the language that denote a binary relation on states. The definitions in Figure 3.1 only give some properties of $\mathcal{M}_{Pred}$ and $\mathcal{M}_{Spec}$ that will be needed later. Obviously, for any given language one will want to define these functions in much more detail, and *Pred* and *Spec* should probably allow expression of non-recursive functions as well. The conditions on these two valuation functions ensure that the languages of predicates and specifications are 'reasonably expressive', at least they should allow one to express all (partial) recursive functions of the appropriate type, for example by expressing a suitable recursion scheme for defining the function.

[3]This property is sometimes called the 'denotational rule'.

Given a set *Name* of identifiers (names) and a set *Val* of values, a state is a map of type

$$\Sigma = \text{map } Name \text{ to } Val_{\perp}$$

Define

$$\Sigma_{\perp} = \Sigma \cup \{\perp\}$$

The valuation function on predicates (over states) is some function

$$\mathcal{M}_{Pred}: Pred \to \Sigma_{\perp} \to \mathbb{B}$$

such that

$$\forall f: \{\text{partial recursive functions } \Sigma_{\perp} \to \mathbb{B}\} \cdot \exists [\![\varphi]\!]: Pred \cdot \mathcal{M}_{Pred}[\![\varphi]\!] = f$$

The valuation function on specifications *Spec* is some function that satisfies

$$\mathcal{M}_{Spec}\ ([\![spec]\!]: Spec)\ R: (\Sigma_{\perp} \times \Sigma_{\perp}) \to \mathbb{B}$$
$$\text{post } \forall \sigma: \Sigma_{\perp} \cdot [R(\perp, \sigma) \Rightarrow \sigma = \perp] \wedge \exists \sigma': \Sigma_{\perp} \cdot R(\sigma, \sigma')$$

and

$$\forall f: \{\text{partial recursive functions } \Sigma_{\perp} \times \Sigma_{\perp} \to \mathbb{B}\} \cdot \exists [\![spec]\!]: Spec \cdot \mathcal{M}_{Spec}[\![spec]\!] = f$$

Figure 3.1: Valuation functions for specifications and predicates

The valuation function $\mathcal{M}_{Spec}$ for specifications maps a specification to a binary relation on states that is interpreted as the input/output relation induced by the specification. Since the investigations in this thesis are restricted to specification languages that are based on the notions of state and state transitions (cf. §1.5), this seems the most appropriate approach.

For example in VDM, if $[\![spec]\!]$ contains a pre-condition and σ_1 is a state not satisfying this pre-condition, then $\mathcal{M}_{Spec}[\![spec]\!](\sigma_1, \sigma_2)$ is true for all $\sigma_2: \Sigma_{\perp}$. If σ_1 does satisfy the pre-condition, then $\mathcal{M}_{Spec}[\![spec]\!](\sigma_1, \perp)$ is false.

Note that the specification of Σ with function $\mathcal{M}_{Spec}$ is *sufficiently abstract* in the sense of [Jon86, page 233].[4]

The definition of $\mathcal{M}_{Spec}$ in Figure 3.1 can easily be extended to cover *sequences* of specifications:

$$\mathcal{M}_{Spec}[\![[spec]]\!] \triangleq \mathcal{M}_{Spec}[\![spec]\!]$$

$$\mathcal{M}_{Spec}[\![[spec_1, \ldots, spec_{n-1}, spec_n]]\!] \triangleq \mathcal{M}_{Spec}[\![[spec_1, \ldots, spec_{n-1}]]\!] \circ \mathcal{M}_{Spec}[\![spec_n]\!]$$

using overloading of $\mathcal{M}_{Spec}$.

[4] A model is sufficiently abstract if, for any two different states, one can find a sequence of operations that distinguishes them. This is very similar to full abstraction of the denotational semantics of a language in relation to its operational semantics.

3.2 Execution and executability

We now define what exactly is meant by the word 'language':

Definition 3.2.1 (Languages) *A (programming or specification) language $\mathcal{L}$ is described by a syntax*[5] *and a semantics. The syntax is given in terms of production rules and possibly some additional restrictions. The semantics of a language may be given in any of the styles described in §3.1.*

In order to emphasize the fact that a term has been formed according to the production rules and *it satisfies the additional restrictions on the syntax of $\mathcal{L}$, it will be called a* well-formed *term.*

The additional restrictions mentioned in this definition are used to 'weed out' special cases of terms, when this would not (or at least not easily) be possible using only production rules. For example, most languages demand that a variable is declared before it can be used. These additional restrictions are often called context conditions, and they can be considered as part of the *static semantics* of the language. However, in this context it is more convenient not to do so but to consider them as part of the syntax of the language.

Definition 3.2.2 (Execution and interpretation) *Given a specification $[\![spec]\!]$, interpretation of $[\![spec]\!]$ is a state transformation from a state σ to a state σ' s.t. $\mathcal{M}_{Spec}[\![spec]\!](\sigma, \sigma')$. Here "transformation into state $\perp$" denotes non-termination or abortion of the interpretation. interpret is defined as an arbitrary function*

$$interpret: Spec \rightarrow \Sigma_\perp \rightarrow \Sigma_\perp$$

that satisfies

$$\mathcal{M}_{Spec}[\![spec]\!](\sigma, interpret[\![spec]\!]\sigma)$$

If this state transformation is given by a (partial) recursive algorithm, then interpretation of $[\![spec]\!]$ is called execution.

Note that in general the result $interpret[\![spec]\!]\sigma$ is not defined uniquely by this definition, since $[\![spec]\!]$ may be under-determined or non-deterministic. When dealing with such a specification, one has to force the interpretation of $[\![spec]\!]$ into choosing a particular value out of the set of possible values.

Definition 3.2.3 (Executable languages) *A language $\mathcal{L}$ is executable if every specification term in $\mathcal{L}$ can be executed in the sense of Definition 3.2.2.*

An $\mathcal{L}$-term is executable if it is a term of an executable sub-language of $\mathcal{L}$.

A special case of this is a language $\mathcal{L}$ with operational semantics. In this case, the recursive algorithm is given explicitly by the operational semantics, and the language is therefore executable.

A difficulty that arises in this context is that even though recursive functions do model the hardware operations within a computer to a certain extent, they do not take into account time

[5] Strictly speaking, one has to distinguish between abstract and concrete syntax. Since the distinction is irrelevant in this context, it will be ignored in the following.

and space restrictions that apply to any computer in the real world. Strictly speaking, almost all language constructs are therefore non-executable on real computers, since for sufficiently large arguments, the capacity of any computer will be exceeded. However, Definition 3.2.3 ignores such restrictions and says that a language is executable if it is executable given a large enough computer and unlimited (but finite) time.

3.3 Specifications and specification languages

The reason for writing a specification instead of starting with the implementation is that the specification may be more abstract in one or more of the following ways:

Data In specifications one may use abstract data types (e.g. sets) that are not provided by the implementation language. This implies that in a specification it is easier to describe objects in terms of the underlying problem domain rather than in terms of the computer technology used. Some programming languages do support some more abstract data types, e.g. me too [Hen84, HM85] and SETL [SDDS86], but they can never be as general and expressive as is possible for specification languages.

'What' vs 'How' A specification does not have to state *how* a certain goal is going to be achieved, its purpose is to state *what* the program is supposed to do. To a certain extent, this is also achieved using high-level programming languages, but again, specification languages can go much further towards reaching this goal than programming languages can.

Property oriented In a specification, we can describe a system in terms of its properties. This seems to be the most important aspect of abstraction in specifications and can be achieved using

- pre-/post-conditions: operations can be specified using pre- and post-conditions. These can best be described in some logic notation, for example using Horn clauses, or first order predicate calculus (classical or LPF, see Appendix B.2), or temporal logic.
- algebraic representation: the system is described by building an algebraic model, where a data structure is defined in terms of the operations that can act on it.

In each of these, programming languages can go some way towards reaching these goals, but they can never go as far as specification languages can. This is because of both efficiency and theoretical reasons.

In the following, I am going to describe some principles that may be used to distinguish programming and specification languages.

Definition 3.2.3 raises the question whether there is any language construct which is not executable but

- would be useful in a programming language, or
- is used in any (programming or specification) language

Imagine for example the function V: *Formula* $\rightarrow \mathbb{B}$ that returns true if its argument, a formula of second-order predicate calculus, is valid, and false otherwise. Obviously V is not recursive and therefore not executable in the sense of Definition 3.2.3. However, for some formulae ϕ it is possible to determine $V(\phi)$, using a recursive algorithm. One might therefore want to include the function V in the programming language. In this case, it would be up to the programmer to ensure that V is only applied to such arguments as can be handled by the recursive algorithm supplied, other arguments must be filtered out beforehand.

Although such a construct is conceivable as part of a programming language, this particular one has never been used as such. On the other hand, there is at least one construct available in most programming languages which is not in general executable, namely the `while`-statement. Execution of this is usually defined using a recursion equation, but it is not in general decidable whether any given state is in the domain of the least fixed point of this equation. It is the programmer's responsibility to only apply the construct in states in the domain of the recursion equation, i.e. ensure that the loop terminates.

For `while`-loops, the problem of executability is thus solved by providing a clear and natural operational semantics, and making the programmer responsible for ensuring the executability of any terms using `while`. For specification languages, on the other hand, one does not need to be quite as strict. Here one may allow terms and constructs that are non-executable, or for which at least one cannot decide in advance whether they are executable or not, as long as they are considered useful. It is then up to the specifier to ensure that a specification denotes a recursive function. (This is expressed as the "implementability proof obligation" in VDM.)

I have just mentioned the concept of *implementability*. The idea behind this is that even if a specification $[\![spec_1]\!]$ is not executable itself, it may still be possible to achieve the effect of $[\![spec_1]\!]$ with an executable program $[\![spec_2]\!]$. Since $[\![spec_1]\!]$ does not have to be deterministic, it can be implemented by selecting particular output values out of the set of possible ones. As a result, the meaning of $[\![spec_2]\!]$ may be included in that of $[\![spec_1]\!]$, rather than the two being equal.

Let the domain of a relation on states be given by

$$\mathsf{dom} : ((\Sigma_\perp \times \Sigma_\perp) \rightarrow \mathbb{B}) \rightarrow \Sigma$$

$$\mathsf{dom}\, R \triangleq \{\sigma \in \Sigma \mid \exists \sigma' \in \Sigma \cdot R(\sigma, \sigma')\}$$

Definition 3.3.1 (Satisfaction and Implementability) *Satisfaction of a specification is defined by*

$$satisfies : Spec \times Spec \rightarrow \mathbb{B}$$

$$satisfies([\![spec_1]\!], [\![spec_2]\!]) \triangleq$$
$$\quad \mathsf{dom}\, \mathcal{M}_{Spec}[\![spec_2]\!] \subseteq \mathsf{dom}\, \mathcal{M}_{Spec}[\![spec_1]\!]$$
$$\quad \wedge\, \mathcal{M}_{Spec}[\![spec_1]\!] \subseteq \mathcal{M}_{Spec}[\![spec_2]\!]$$

This will usually be written in infix notation as $[\![spec_1]\!]satisfies[\![spec_2]\!]$.

If $[\![spec_1]\!]satisfies[\![spec_2]\!]$ *and is* executable, *then we call* $[\![spec_1]\!]$ *an* implementation *of* $[\![spec_2]\!]$ *and* $[\![spec_2]\!]$ *a* specification *of* $[\![spec_1]\!]$. $[\![spec_1]\!]$ *is called* implementable *if there exists an implementation of it.*

It follows that a specification $[\![spec]\!]$ is implementable iff it denotes a recursive function. In particular, every executable program is implementable. Note that $[\![spec_1]\!]$ and $[\![spec_2]\!]$ are not required to be terms in the same language, quite often $[\![spec_2]\!]$ will actually be written in a non-executable (specification) language, while $[\![spec_1]\!]$ is written in an executable (programming) language. On the other hand, $\mathcal{L}_2$ might be an executable language as well (and even equal to $\mathcal{L}_1$). This is often done if $[\![spec_2]\!]$ has some other disadvantage, for example inefficiency. $[\![spec_2]\!]$ might be written in a high-level but inefficient language and used as a prototype.

The definition of *satisfies* can be interpreted as demanding that $[\![spec_2]\!]$ is both *at least as defined* and *at least as determined* as $[\![spec_1]\!]$. Due to our definition of dom, the second actually implies that $[\![spec_2]\!]$ cannot be *more* defined than $[\![spec_1]\!]$, so that one might equivalently demand that the domains of $[\![spec_1]\!]$ and $[\![spec_2]\!]$ are equal. An alternative interpretation of the second conjunct in the definition of *satisfies* is that the transformation from a specification to its implementation is non-correctness-preserving under both parsings of the word: if a pair $(\sigma_1, \sigma_2) \in \Sigma\times\Sigma$ describes a non-correct input-output behaviour w.r.t. the specification, this behaviour will also be non-correct w.r.t. the implementation, i.e. non-correctness is preserved. Correctness, on the other hand, is not in general preserved under this transformation. The first conjunct then restricts the amount of correctness that can be 'lost' by the transformation.

The approach to satisfaction used here is different from that used, for example, in VDM, where the implication $\mathcal{M}_{Spec}[\![spec_1]\!](\sigma, \sigma') \Rightarrow \mathcal{M}_{Spec}[\![spec_2]\!](\sigma, \sigma')$ would only be required to hold if $\sigma \in \mathrm{dom}\, \mathcal{M}_{Spec}[\![spec_2]\!]$, and dom R would be defined as $\{\sigma \in \Sigma \mid \exists \sigma' \in \Sigma \cdot R(\sigma, \sigma') \wedge \neg R(\sigma, \perp)\}$. In other words, if a specification is allowed not to terminate for some input state $\sigma: \Sigma$, then *any* output satisfies the specification for input σ. The approach used here has the advantage that it makes it possible to describe the specification of *partially* correct behaviour. While this is not needed in VDM since there one is not interested in specifying partial correctness and insists on total correctness over the domain of interest (i.e. when the pre-condition holds), the framework described here is intended to be more general and cater for other specification languages as well. The version of correctness presented here is sometimes called *loose* correctness, while the version of correctness used in VDM is called *robust* correctness (cf. [Bro85]). Note however that the model of symbolic execution described later will *not* depend on this notion of satisfaction, satisfaction will only be used to *describe properties* of the model.

Another approach to implementing specifications is the use of a wide-spectrum language which is not itself executable, but which contains an executable sub-language. One then writes a specification using the full language and implements it in the executable sub-language. This approach is often combined with transformational programming, as for example in the CIP project [B+87].

3.4 Term rewriting

This section has been included to provide some of the background and notation for the discussion of operational semantics in Plotkin's SOS-style (cf. §3.1).

3.4.1 Basic concepts

The standard reference article on this subject is the survey [HO80]. The following is mainly based on that article and on [Der85].

Definition 3.4.1 *Let $\mathcal{T}$ be some set of terms, let $v(t)$ denote the set of free variables in the term $t \in \mathcal{T}$, and let $\vec{x} = (x_1, \ldots, x_n)$. A* rewrite rule *over $\mathcal{T}$ is a directed equation $l[\vec{x}] \longrightarrow r[\vec{x}]$ where $l, r \in \mathcal{T}$ and $v(r) \subseteq v(l) \subseteq \{x_1, \ldots, x_n\}$. A* (term) rewriting system **R** *is a set of such rewrite rules, and $=_{\mathbf{R}}$ is the equality generated by the set of equations* **R**, *ignoring their directions. We say that* **R** *is* finite *if it contains a finite number of rewrite rules.*

A rewrite rule $l[\vec{x}] \longrightarrow r[\vec{x}]$ can be applied to a term $t \in \mathcal{T}$ if t contains a subterm s that matches l, i.e. for some $\vec{a} \in \mathcal{T}^n$ we get $s = l[\vec{x}/\vec{a}]$. Here $l[\vec{x}/\vec{a}]$ denotes the result of replacing the variables $\vec{x} = (x_1, \ldots, x_n)$ in the term l by $\vec{a} = (a_1, \ldots, a_n)$.

Applying the rule to t then means replacing s in t by $r[\vec{x}/\vec{a}]$, which yields some new term t'. We denote this by $t \Rightarrow t'$, or, to stress the dependence on **R**, by $t \Rightarrow_{\mathbf{R}} t'$. Repeated application of rules from **R** is denoted by $t \overset{*}{\Rightarrow} t'$ (transitive-reflexive closure, any number of repetitions) or $t \overset{+}{\Rightarrow} t'$ (transitive closure, at least one repetition).

Definition 3.4.2 *A term t is* irreducible *or in* normal form *(with respect to* **R**), *iff no rule from* **R** *can be applied to t.*
R *is* confluent *or* uniquely terminating *or* convergent *iff*

$$\forall t_1, t_2, s \in \mathcal{T} \cdot s \overset{*}{\Rightarrow} t_1 \text{ and } s \overset{*}{\Rightarrow} t_2 \text{ implies } \exists p \in \mathcal{T} \cdot t_1 \overset{*}{\Rightarrow} p \text{ and } t_2 \overset{*}{\Rightarrow} p.$$

R has the Church-Rosser property *iff*

$$\forall t_1, t_2 \in \mathcal{T} \cdot t_1 =_{\mathbf{R}} t_2 \text{ iff } \exists s \in \mathcal{T} \cdot t_1 \overset{*}{\Rightarrow} s \text{ and } t_2 \overset{*}{\Rightarrow} s.$$

R *is* nœtherian *or* (finitely) terminating *iff for no $t \in \mathcal{T}$ there exists an infinite chain $t \Rightarrow t_1 \Rightarrow t_2 \Rightarrow \ldots$.*
R *is a* complete *or* canonical *rewrite system iff it is finite, confluent and nœtherian.*

Lemma 3.4.3 *a) If* **R** *is nœtherian, then every term has a normal form.*
b) If **R** *is confluent, then the normal form of any term is unique, if it exists.*
c) **R** *is confluent* iff *it has the Church-Rosser property.*

Lemma 3.4.4 (Decidability) *a) For general rewrite systems, confluence and termination are undecidable.*
b) If **R** *is nœtherian, then the confluence of* **R** *is decidable.*
c) If **R** *is complete, then the equational theory $=_{\mathbf{R}}$ is decidable.*

From these lemmas it can be seen that an important question arising when dealing with a rewrite system **R** is to decide whether it is nœtherian. If it is, then most of the interesting properties of **R** are decidable.

3.4.2 Termination

Termination of a rewrite system is usually proven using monotonic well-founded orderings.

A strict partial ordering $\prec$ over a set of terms T is *monotonic* if $t_1 \prec t_2 \Rightarrow f(\ldots, t_1, \ldots) \prec f(\ldots, t_2, \ldots)$, where $f(\ldots, t, \ldots)$ denotes any term which contains the subterm t. A monotonic ordering $\prec$ is a *simplification ordering* if for all terms from T, $f(\ldots, t, \ldots) \succ t$ holds. $\prec$ is *well founded* if there is no infinite descending sequence $t_1 \succ t_2 \succ t_3 \succ \ldots$.

Given some monotonic well-founded ordering $\succ$, we say that a rewrite rule $l \longrightarrow r$ is a *reduction*, if $l \succ r$ for any substitution of ground terms for its variables.

Lemma 3.4.5 *a) A rewrite system* **R** *over a set T of terms is nœtherian* iff *there exists a monotonic well-founded ordering $\prec$ of T such that all rules $l \longrightarrow r$ in* **R** *are reductions.*

b) If there exists a simplification ordering $\prec$ over T such that for any rule $l \longrightarrow r$ in **R** *and any substitution of terms for its variables, $l \succ r$ holds, then* **R** *is nœtherian.*

[HO80] describes a number of possible orderings of terms.

3.5 Relationship between denotational and operational semantics

§3.1 discussed different styles of describing the semantics of a specification language. Now given two such descriptions of the same language, but in different styles, what is their relationship? How can one ensure that they do indeed describe the same language, or at least do not contradict each other? In this context, I am only interested in the relationship between denotational and operational semantics, since these are the two styles that I will use for describing symbolic execution. The following description is based on [Sch86, §10.7].

The operational semantics of a language can be considered as an *interpreter* for the language, consisting of a set of (interpreter) configurations and a binary relation on configurations (called evaluation or reduction relation). A configuration consists of a state σ and a sequence of specifications. The denotation of such a configuration is the state that results from interpreting the sequence of specifications starting from σ, or the set of such states if the specifications are non-deterministic or under-determined. This interpretation should be mirrored by the reduction relation of the operational semantics, which should transform configurations into configurations with a simpler (usually shorter) sequence of specifications, until finally a configuration with an empty sequence is reached. Such a configuration is called *final*.

Now we can define a notion of soundness, called faithfulness of an operational semantics, considered as a set of transitions from configurations to configurations, with respect to a denotational semantics $\mathcal{M}$:

Definition 3.5.1 (Faithfulness) *a) A transition $c_1 \hookrightarrow c_2$ is faithful with respect to $\mathcal{M}$, if it implies $\mathcal{M}[\![c_1]\!] = \mathcal{M}[\![c_2]\!]$, or $\mathcal{M}[\![c_1]\!] \supseteq \mathcal{M}[\![c_2]\!]$ if $\mathcal{M}$ returns a* set *of valuations.*

b) An operational semantics $\_ \hookrightarrow \_$ is faithful to a denotational semantics $\mathcal{M}$ if for all configurations c_1 and c_2, $c_1 \hookrightarrow c_2$ implies $\mathcal{M}[\![c_1]\!] = \mathcal{M}[\![c_2]\!]$, or $\mathcal{M}[\![c_1]\!] \supseteq \mathcal{M}[\![c_2]\!]$ if $\mathcal{M}$ returns a set *of valuations.*

For the operational semantics to be useful, it is not enough to be just faithful, however, since even an empty reduction relation is faithful. One therefore needs a suitable notion of completeness. This should express that the reduction relation does indeed reduce configurations to final configurations.

Similarly, there are a number of other important properties describing the relationship between denotational and operational semantics, for example full abstraction. However, while faithfulness is a property of *individual* transitions, these other properties discuss operational semantics as a whole. Since this thesis does not give the complete operational semantics of any language, but only a few transitions describing some important language constructs, these other properties are not relevant in this context.

Chapter 4

Semantics of symbolic execution

"I don't know what you mean by 'glory,' " Alice said

Humpty Dumpty smiled contemptuously. "Of course you don't — till I tell you. I meant 'there's a nice knock-down argument for you!' "

"But glory doesn't mean 'a nice knock-down argument,'" Alice objected.

"When I use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean — neither more nor less."

"The question is," said Alice, "whether you can make words mean so many different things."

"The question is," said Humpty Dumpty, "which is to be master — that's all."

Lewis Carroll: Through the Looking Glass

4.1 Denotational semantics of symbolic execution

4.1.1 The semantic model

The following description starts off with some possible semantic models that turn out to contain insufficient information and were therefore rejected, but provided stepping stones on the way to the construction of the model that will be used.

As a first attempt at a formal description of symbolic execution, one might try to base it on the observation that in symbolic execution, the input to a specification $[\![spec]\!]$ can be considered as a set $S \subseteq \Sigma_\perp$ of states. As output, it returns the set of states that can be reached from a state in S via $\mathcal{M}_{Spec}[\![spec]\!]$. However, for describing the denotational semantics of symbolic execution this is not sufficient, since it would lose all the information about the relationship between input

and output states themselves, as opposed to the relationship between the *sets* of these states.[1] For example, given the specification $x = 0 \vee x = \overleftarrow{x} + 1$, symbolic execution would map $\mathbb{N}$ to $\mathbb{N}$ and not really provide sufficient information. To get more useful information, one would have to restrict the set S, in this case $\{\sigma \mid \sigma(x) \in \mathbb{N}\}$, to a small subset, which would be contrary to the ideas of symbolic execution and lead towards 'testing' of specifications.

Now consider the following improved attempt at a definition of symbolic execution: again, the input consists of a set $S \subseteq \Sigma_\perp$ of states. Then the output of symbolic execution of a specification $[\![spec]\!]$ has as denotation the function $\lambda\sigma \in S \cdot \{\sigma_1 \in \Sigma_\perp \mid \mathcal{M}_{Spec}[\![spec]\!](\sigma, \sigma_1)\}$, which maps a state σ in S to the set of all states that may result from interpreting $[\![spec]\!]$ starting in state σ.

The reason for starting symbolic execution with a set $S \subseteq \Sigma_\perp$, rather than $\Sigma_\perp$ itself, is that the model should support restrictions on the original set of parameters. This means that the result of applying symbolic execution may not be defined for every parameter state $\sigma: \Sigma_\perp$, only for those in the original set. In other words, the output of symbolic execution of a specification denotes a *partial* function. The reason for introducing such restrictions is that the output terms resulting from symbolic execution can get very complicated, partly because of conditionals that have to be introduced to describe the effect of branching statements. By introducing suitable restrictions on the input domain, the user can ensure that only one branch can be taken and the conditional can be eliminated from the output expression. For example, with a specification `if` $x \geq 0$ `then` $[\![spec_1]\!]$ `else` $[\![spec_2]\!]$ one might at first only want to consider the case $x \geq 0$, and therefore start from the set $S_1 = \{\sigma: \Sigma_\perp \mid \sigma(x) \geq 0\}$, considering $x < 0$ separately.

The main problem with this improved description of symbolic execution is that it does not allow composition of two (or more) symbolic execution steps, input and output do not even have the same type. One way to overcome this problem is to introduce the initial starting state σ as a parameter to the sets of states as described above. Rather than starting symbolic execution off with a *set* of states, one already starts with a map from states to sets of states. In the first step, this will essentially be the identity map on the input set of states; after one or more symbolic execution steps it will be the map from the original parameter state to the set of states that can be reached from the parameter state via the appropriate sequence of operations. Thus after n steps one gets

$$\{\sigma \mapsto \{\sigma_n \mid \exists \sigma_1, \ldots, \sigma_{n-1} \cdot \mathcal{M}_{Spec}[\![spec_1]\!](\sigma, \sigma_1) \wedge \ldots \wedge \mathcal{M}_{Spec}[\![spec_n]\!](\sigma_{n-1}, \sigma_n)\} \mid \sigma \in S\}$$

One might therefore consider such a map as a partial non-deterministic state transition function. A sequence of symbolic execution steps is then described by the sequence of state transition functions generated in each step.

Such a collection of parameterised sets of states can often be expressed as a term containing a free (input) variable x, or, more general, a predicate containing the free variables x and y, where y denotes the output variables of $[\![spec]\!]$.

The model of symbolic execution as described would be enough if one only had to deal with symbolic execution itself. However, it does not support intermediate **assume** or **believe** commands (as described in §6.1.2). **assume** assumes that a given logical expression is true, and

[1] For other purposes it can still be very useful to consider these sets; this is essentially what is done in *abstract interpretation*, cf. §2.3.

thus restricts the set of possible execution sequences. The model described above allows for such restrictions only that restrict the set of *initial input states*, but not for any restrictions on later states. Although in most cases it only makes sense to introduce such later restrictions if at some stage the negations of these restrictions are dealt with as well, one still needs this possibility to ensure that result expressions do not get too complicated.

believe also assumes that a logical expression is true. The difference to **assume** is that a **believed** logical expression is considered as a *proof obligation* to be discharged later, the belief has to be *justified*. It thus does not restrict the set of parameter states, but only expresses a condition that holds already but has not been proven yet.

There are a number of different ways of overcoming this problem of supporting intermediate **assume** or **believe** commands. Instead of modelling each step in a sequence of symbolic executions by a map from the *initial* starting state to the set of reachable states, one could use maps from the starting state *of this step* to the set of reachable states. The sequence of symbolic executions would then essentially be described by the sequence of meaning relations of executed specifications. This does model restrictions on parameter states, but only if they are expressed using two consecutive states (or only a single one), other restrictions cannot in general be modelled in this way.

For this reason, the model of symbolic execution selected is based on a 'symbolic execution state' called *SEStateDen* which contains sets of *sequences* of states. The definition of *SEStateDen* is given in Figure 4.1. The name *SEStateDen* is a shorthand for *S*ymbolic *E*xecution *State* as used for *Den*otational semantics. Similarly, §4.2 will introduce *SEStateOp* for states in operational semantics.

In addition to the set of sequences of states, *SEStateDen* contains a field *LEN* which stores the number of symbolic execution steps performed, plus 1 for the initial state (see Figure 4.1). At the same time, this is the number of *actual* execution steps modelled in any sequence of states in the field *SEQS* plus 1, which leads to the first conjunct in the invariant. In this model, **assume** or **believe** restrictions are modelled by "cutting off" as much as necessary from the end of all sequences of states until the condition is satisfied. This intuition explains the second conjunct on the invariant on *SEStateDen*, which demands that no sequence in *SEStateDen* is an initial segment of another such sequence.

As a convention, τ will be used to denote elements of *SEStateDen*, while σ denotes elements of $\Sigma_\perp$, as before.

Symbolic execution of a specification is modelled by adding another state to all those sequences that have not been "cut off", see Figure 4.2. Just as interpretation or execution, given a specification, maps states to states, so symbolic execution, given a specification, maps *SEStateDen*s to *SEStateDen*s.

Doing symbolic execution in the way described here and storing *all* possible sequences of states allowed by a sequence of specifications requires a fairly rich language for expressing the results of symbolic execution, which might not always be available. For example, the result of executing a `while`-loop will often not be expressible in the language available. Therefore, in addition to such *full* symbolic execution Figure 4.2 also defines *weak* symbolic execution, where the result *includes* the set of all possible sequences of states. This ensures that the properties one

A state as used in symbolic execution is given by

$$SEStateDen :: SEQS : \mathcal{P}(\text{seq of } \Sigma_\perp)$$
$$LEN : \mathbb{N}$$

where

$$inv\text{-}SEStateDen(mk\text{-}SEStateDen(set, l)) \triangleq$$
$$\forall \sigma\text{-}seq \in set \cdot \text{len } \sigma\text{-}seq \leq l$$
$$\wedge \forall \sigma\text{-}seq_1, \sigma\text{-}seq_2 \in set \cdot \forall \sigma\text{-}seq: \text{seq of } \Sigma_\perp \cdot$$
$$\sigma\text{-}seq_1 = \sigma\text{-}seq_2 \frown \sigma\text{-}seq \Rightarrow \sigma\text{-}seq = [\,]$$

Here $\frown$ denotes concatenation of sequences.
A set $S \subseteq \Sigma_\perp$ of states (or, similarly, a predicate on states) can be represented by the *SEStateDen*

$$\tau(S) \triangleq mk\text{-}SEStateDen(\{[\sigma] \mid \sigma \in S\}, 1)$$

The function *yield* extracts the input/output relationship from the sequences in *SEStateDen*.

$$yield(\tau) \triangleq \lambda\sigma: \Sigma_\perp \cdot \{\sigma': \Sigma_\perp \mid \exists \sigma\text{-}seq \in SEQS(\tau) \cdot$$
$$\text{hd } \sigma\text{-}seq = \sigma \wedge \text{last } \sigma\text{-}seq = \sigma'$$
$$\wedge \text{len } \sigma\text{-}seq = LEN(\tau)\}$$

Figure 4.1: Denotational semantics of symbolic execution — State

gets as a result of weak symbolic execution still hold for the denotation of the full result, they just do not in general give a complete description.

Since in many cases one is really interested in the relationship between input and output states and less in the intermediate states, a function called *yield* for extracting this relationship from an *SEStateDen* is also provided (in Figure 4.1). This can be considered as extracting from an *SEStateDen* the map from initial states to possible resulting states, the possible data model for symbolic execution rejected above. It thus is quite similar to the *yield* operator $^+$ introduced in [dBZ82].

Note that there is a distinction between symbolic execution of the composition of specifications and the composition of symbolic executions. As Lemma 4.1.7 will show, they give rise to *SEStateDens* that describe the same relationship between initial and final states, but the *SEStateDens* themselves are different. They lead to *SEStateDens* of different lengths, since symbolic execution of the composition of specifications is considered as a single step, while a sequence of symbolic executions in general consists of several steps.

It is not immediately obvious that *symbolic-ex* as defined is a *total* function. Although a result is constructed for any input values, this result might not be of type *SEStateDen*. The following lemma shows that this case does not arise.

Lemma 4.1.1 *symbolic-ex is total.*

(Full) symbolic execution is given by the functions

$$symbolic\text{-}ex : Spec \to SEStateDen \to SEStateDen$$

$$symbolic\text{-}ex[\![spec]\!]\tau \triangleq$$
$$mk\text{-}SEStateDen($$
$$\{\sigma\text{-}seq \mid \text{len } \sigma\text{-}seq = LEN(\tau) + 1 \wedge \text{front } \sigma\text{-}seq \in SEQS(\tau)$$
$$\wedge \mathcal{M}_{Spec}[\![spec]\!](\text{last front } \sigma\text{-}seq, \text{last } \sigma\text{-}seq)$$
$$\vee \text{len } \sigma\text{-}seq < LEN(\tau) \wedge \sigma\text{-}seq \in SEQS(\tau)\},$$
$$LEN(\tau) + 1)$$

and

$$symbolic\text{-}ex\text{-}s : \text{seq of } Spec \to SEStateDen \to SEStateDen$$

$$symbolic\text{-}ex\text{-}s[\![spec\text{-}seq]\!]\tau \triangleq$$
$$\text{if } spec\text{-}seq = [\,]$$
$$\text{then } \tau$$
$$\text{else } symbolic\text{-}ex\text{-}s[\![\text{tl } spec\text{-}seq]\!](symbolic\text{-}ex[\![\text{hd } spec\text{-}seq]\!]\tau)$$

Weak symbolic execution is a function

$$w\text{-}symbolic\text{-}ex\ ([\![spec]\!]: Spec, \tau_1: SEStateDen)\ \tau_2: SEStateDen$$
$$\text{post } SEQS(\tau_2) \supseteq SEQS(symbolic\text{-}ex[\![spec]\!]\tau_1)$$
$$\wedge LEN(\tau_2) = LEN(symbolic\text{-}ex[\![spec]\!]\tau_1)$$

with a similar function for sequences of specifications.

Figure 4.2: Denotational semantics of symbolic execution — Functions

Proof We have to show that for any $[\![spec]\!]: Spec$, $\tau: SEStateDen$

$$inv\text{-}SEStateDen(symbolic\text{-}ex[\![spec]\!]\tau)$$

The first condition is obviously true. Now let

$$\sigma\text{-}seq_1, \sigma\text{-}seq_2 \in SEQS(symbolic\text{-}ex[\![spec]\!]\tau)$$

and $\sigma\text{-}seq$: seq of $\Sigma_\perp$, $\sigma\text{-}seq \neq [\,]$ be such that

$$\sigma\text{-}seq_1 = \sigma\text{-}seq_2 \frown \sigma\text{-}seq$$

The definition of *symbolic-ex* then implies that $\sigma\text{-}seq_2 \in SEQS(\tau)$.

Case 1: $\text{len } \sigma\text{-}seq_1 = LEN(\tau) + 1$

Then

$$\sigma\text{-}seq_2 \frown \text{front } \sigma\text{-}seq = \text{front } \sigma\text{-}seq_1 \in SEQS(\tau)$$

and *inv-SEStateDen*(τ) implies that front σ-*seq* = [ ], i.e. len σ-*seq* = 1. But then len σ-*seq*$_2$ = *LEN*(τ), therefore σ-*seq*$_2$ cannot be in

SEQS(*symbolic-ex*[[*spec*]]τ) — contradiction.

Case 2: len σ-*seq*$_1$ < *LEN*(τ)

In this case σ-*seq* = [ ] follows immediately from *inv-SEStateDen*(τ).

□

4.1.2 Some properties of symbolic execution

After an example, this subsection contains some lemmata describing properties of this model of symbolic execution. Most of these properties are those that one would 'obviously' expect to hold, they thus serve to validate our model.

Example 4.1.2 Let *Name* = $\{x, y\}$. We want to symbolically execute the operation

OP_1
ext wr x : $\mathbb{Z}$
  wr y : $\mathbb{N}$
pre $x \geq 0$
post $y^2 \leq \overleftarrow{x} \wedge x = \overleftarrow{x} + 1$

Then

$\mathcal{M}_{Spec}[\![OP_1]\!](\sigma, \sigma_1) \Leftrightarrow$ if $\sigma(x) \geq 0$ then $\sigma_1(y)^2 \leq \sigma(x) \wedge \sigma_1(x) = \sigma(x) + 1$ else true

Now the user **assumes** that the pre-condition of OP_1 is true. This means that OP_1 is to be symbolically executed in the *SEStateDen* τ_1 which represents the predicate $x \geq 0$:

$$\begin{aligned} \tau_1 &= \textit{mk-SEStateDen}(\{[\sigma] \mid \mathcal{M}_{Pred}[\![x \geq 0]\!]\sigma\}, 1) \\ &= \textit{mk-SEStateDen}(\{[\sigma] \mid \sigma(x) \geq 0\}, 1) \end{aligned}$$

Then symbolic execution of the specification OP_1 starting in the *SEStateDen* τ_1 results in the *SEStateDen*

$$\begin{aligned} &\textit{symbolic-ex}[\![OP_1]\!]\tau_1 \\ &= \textit{mk-SEStateDen}(\{\sigma\text{-}\textit{seq} \mid \text{len } \sigma\text{-}\textit{seq} = \textit{LEN}(\tau_1) + 1 \wedge \text{front } \sigma\text{-}\textit{seq} \in \textit{SEQS}(\tau_1) \\ &\qquad \wedge \mathcal{M}_{Spec}[\![OP_1]\!](\text{last front } \sigma\text{-}\textit{seq}, \text{last } \sigma\text{-}\textit{seq}) \\ &\qquad \vee \text{len } \sigma\text{-}\textit{seq} < \textit{LEN}(\tau_1) \wedge \sigma\text{-}\textit{seq} \in \textit{SEQS}(\tau_1)\}, \textit{LEN}(\tau_1) + 1) \\ &= \textit{mk-SEStateDen}(\{\sigma\text{-}\textit{seq} \mid \text{len } \sigma\text{-}\textit{seq} = 2 \wedge \sigma\text{-}\textit{seq}[1](x) \geq 0 \\ &\qquad \wedge \mathcal{M}_{Spec}[\![OP_1]\!](\sigma\text{-}\textit{seq}[1], \sigma\text{-}\textit{seq}[2])\}, 2) \\ &= \textit{mk-SEStateDen}(\{\sigma\text{-}\textit{seq} \mid \text{len } \sigma\text{-}\textit{seq} = 2 \wedge \sigma\text{-}\textit{seq}[1](x) \geq 0 \\ &\qquad \wedge \sigma\text{-}\textit{seq}[2](y)^2 \leq \sigma\text{-}\textit{seq}[1](x) \\ &\qquad \wedge \sigma\text{-}\textit{seq}[2](x) = \sigma\text{-}\textit{seq}[1](x) + 1\}, 2) \end{aligned}$$

Strictly speaking, Op_1 is the *name* of the operation (or specification) rather than the operation itself. For the time being, I shall use names of specifications to denote both the name itself and the specification referred to by it, until in §4.2.3 a mapping from specification names to specifications is introduced.

The following properties of the model of symbolic execution use the function *yield*. The reason for this is that these properties relate different ways of getting the same mapping from initial states to sets of final result states. Although they arrive at the same such mapping, the execution sequence used to get there may be different, in particular they may have different lengths.

Lemma 4.1.3 *Let* $[\![spec]\!]$: *Spec, let* τ_1, τ_2: *SEStateDen be such that* $symbolic\text{-}ex[\![spec]\!]\tau_1 = \tau_2$. *Then for all* $\sigma, \sigma_1 : \Sigma_\perp$

$$\sigma_1 \in yield(\tau_1)(\sigma) \Rightarrow interpret[\![spec]\!]\sigma_1 \in yield(\tau_2)(\sigma)$$

In particular if τ_1 *represents a set S of states, i.e.* $\tau_1 = mk\text{-}SEStateDen(\{[\sigma] \mid \sigma \in S\}, 1)$, *then for all* $\sigma \in S$

$$[\sigma, interpret[\![spec]\!]\sigma] \in SEQS(\tau_2)$$

This lemma states that the result of interpreting a specification in a state σ can also be achieved by symbolically executing the specification in a *SEStateDen* τ which represents a set of states including σ, and then selecting a sequence starting with σ in the result.

Lemma 4.1.4 *Let* τ: *SEStateDen. Then*

$$yield(symbolic\text{-}ex[\![\texttt{skip}]\!]\tau) = yield(\tau)$$

Lemma 4.1.5 *If* $[\![spec_1]\!]satisfies[\![spec_2]\!]$, *then for all* τ: *SEStateDen*

$$SEQS(symbolic\text{-}ex[\![spec_1]\!]\tau) \subseteq SEQS(symbolic\text{-}ex[\![spec_2]\!]\tau)$$

Lemma 4.1.6 *Given some* $[\![spec]\!]$: *Spec and* τ: *SEStateDen. Then*

$$\begin{aligned} &yield(symbolic\text{-}ex[\![spec]\!]\tau) \\ &= \{\sigma \mapsto \{\sigma_1 \mid \exists \sigma_2 \cdot \sigma_2 \in yield(\tau)(\sigma) \wedge \mathcal{M}_{Spec}[\![spec]\!](\sigma_2, \sigma_1)\}\} \end{aligned}$$

Proof

$$\begin{aligned} &yield(symbolic\text{-}ex[\![spec]\!]\tau) \\ &= \lambda\sigma \cdot \{\sigma_1 \mid \exists \sigma\text{-}seq \in SEQS(symbolic\text{-}ex[\![spec]\!]\tau) \cdot \\ &\qquad \mathsf{hd}\ \sigma\text{-}seq = \sigma \wedge \mathsf{last}\ \sigma\text{-}seq = \sigma_1 \\ &\qquad \wedge \mathsf{len}\ \sigma\text{-}seq = LEN(symbolic\text{-}ex[\![spec]\!]\tau)\} \\ &= \lambda\sigma \cdot \{\sigma_1 \mid \exists \sigma\text{-}seq' \cdot \exists \sigma_2 \cdot \mathsf{len}\ \sigma\text{-}seq' = LEN(\tau) \wedge \sigma\text{-}seq' \in SEQS(\tau) \\ &\qquad \wedge \mathcal{M}_{Spec}[\![spec]\!](\mathsf{last}\ \sigma\text{-}seq', \sigma_2) \wedge \mathsf{hd}\ \sigma\text{-}seq' = \sigma \wedge \sigma_2 = \sigma_1\} \\ &= \lambda\sigma \cdot \{\sigma_1 \mid \exists \sigma\text{-}seq' \cdot \mathsf{len}\ \sigma\text{-}seq' = LEN(\tau) \wedge \sigma\text{-}seq' \in SEQS(\tau) \wedge \mathsf{hd}\ \sigma\text{-}seq' = \sigma \\ &\qquad \wedge \mathcal{M}_{Spec}[\![spec]\!](\mathsf{last}\ \sigma\text{-}seq', \sigma_1)\} \\ &= \lambda\sigma \cdot \{\sigma_1 \mid \exists \sigma_2 \cdot \sigma_2 \in yield(\tau)(\sigma) \wedge \mathcal{M}_{Spec}[\![spec]\!](\sigma_2, \sigma_1)\} \end{aligned}$$

□

4.1.3 Composition of specifications

Let ; denote sequential composition of specifications, and let ∘ denote function composition. Then

Lemma 4.1.7 (Composition) *For all specifications* $[\![spec_1]\!], [\![spec_2]\!]: Spec$,

$$yield \circ symbolic\text{-}ex[\![spec_1; spec_2]\!] = yield \circ symbolic\text{-}ex\text{-}s[\![[spec_1, spec_2]]\!]$$

Proof See Appendix A.1. □

Note that we do *not* have

$$symbolic\text{-}ex[\![spec_1; spec_2]\!] = symbolic\text{-}ex\text{-}s[\![[spec_1, spec_2]]\!]$$

since $[\![spec_1; spec_2]\!]$ is regarded as a single specification, while $[\![[spec_1, spec_2]]\!]$ is a sequence of two specifications. Therefore symbolic execution of the two leads to *SEStateDens* of different lengths.

Example 4.1.8 Given the operation specification

OP_2
ext wr $x : \mathbb{Z}$
rd $y : \mathbb{N}$
pre $-100 \le x \le +100$
post $\exists z: \mathbb{Z} \cdot y * z + x = \overleftarrow{x} \wedge 0 \le x < y$

we want to symbolically execute OP_2 starting in the *SEStateDen* τ_2 resulting from symbolically executing OP_1, as given in Example 4.1.2. From the specification it follows that

$$\begin{aligned}\mathcal{M}_{Spec}[\![OP_2]\!](\sigma, \sigma_1) \Leftrightarrow\ & \text{if } -100 \le \sigma(x) \le +100 \\ & \text{then } \exists z: \mathbb{Z} \cdot \sigma_1(y) * z + \sigma_1(x) = \sigma(x) \\ & \quad \wedge\, 0 \le \sigma_1(x) < \sigma_1(y) \wedge \sigma_1(y) = \sigma(y) \\ & \text{else true}\end{aligned}$$

Then symbolic execution of OP_2 starting in τ_2 results in the τ_3: *SEStateDen* with $LEN(\tau_3) = 3$ and

$$\begin{aligned}& SEQS(symbolic\text{-}ex[\![OP_2]\!]\tau_2) \\ &= \{\sigma\text{-}seq \mid \text{len } \sigma\text{-}seq = LEN(\tau_2) + 1 \wedge \text{front } \sigma\text{-}seq \in SEQS(\tau_2) \\ &\qquad \wedge \mathcal{M}_{Spec}[\![OP_2]\!](\text{last front } \sigma\text{-}seq, \text{last } \sigma\text{-}seq) \\ &\qquad \vee \text{len } \sigma\text{-}seq < LEN(\tau_2) \wedge \sigma\text{-}seq \in SEQS(\tau_2)\} \\ &= \{\sigma\text{-}seq \mid \text{len } \sigma\text{-}seq = 3 \\ &\qquad \wedge \sigma\text{-}seq[1](x) \ge 0 \wedge \sigma\text{-}seq[2](y)^2 \le \sigma\text{-}seq[1](x) \\ &\qquad \wedge \sigma\text{-}seq[2](x) = \sigma\text{-}seq[1](x) + 1\end{aligned}$$

$$\begin{aligned}
&\wedge \mathcal{M}_{Spec}[\![OP_2]\!](\sigma\text{-}seq[2], \sigma\text{-}seq[3])\} \\
= \ &\{\sigma\text{-}seq \mid \mathsf{len}\ \sigma\text{-}seq = 3 \\
&\quad \wedge \sigma\text{-}seq[1](x) \geq 0 \wedge \sigma\text{-}seq[2](y)^2 \leq \sigma\text{-}seq[1](x) \\
&\quad \wedge \sigma\text{-}seq[2](x) = \sigma\text{-}seq[1](x) + 1 \\
&\quad \wedge \mathsf{if}\ -100 \leq \sigma\text{-}seq[2](x) \leq +100 \\
&\quad\quad \mathsf{then}\ \exists z{:}\,\mathbb{Z} \cdot \sigma\text{-}seq[3](y) * z + \sigma\text{-}seq[3](x) = \sigma\text{-}seq[2](x) \\
&\quad\quad\quad \wedge 0 \leq \sigma\text{-}seq[3](x) < \sigma\text{-}seq[3](y) \\
&\quad\quad\quad \wedge \sigma\text{-}seq[3](y) = \sigma\text{-}seq[2](y) \\
&\quad\quad \mathsf{else\ true}\}
\end{aligned}$$

Note that the restriction on the set of starting states for the resulting set of state sequences (i.e. $\sigma\text{-}seq[1](x) \geq 0$) was explicitly introduced by the user, using the **assume** command, before symbolically executing OP_1. This is the reason why, in spite of the second pre-condition $-100 \leq x \leq +100$, the result still considers all $\sigma\text{-}seq$ s.t. $\sigma\text{-}seq[1](x) \geq 0$. Instead, the result itself contains a conditional. It is only for practical reasons that the user will often **assume** that the pre-condition is true, so as to keep the resulting expression reasonably simple.

4.1.4 Non-determinism and under-determinedness

In symbolic execution, the effects of under-determinedness and non-determinism are captured by the *state* rather than by making symbolic execution itself non-deterministic — the reason being that one wants to check that *all* outputs allowed by the specification or program are correct, and not just one of them.

As an example, consider the command (from Dijkstra's language of guarded commands [Dij76])

$$IF \ \triangleq \ \mathtt{if}\ b_1 \rightarrow spec_1 \ \square\ b_2 \rightarrow spec_2\ \mathtt{fi}$$

The meaning of IF is given by

$$\begin{aligned}
\mathcal{M}_{Spec}[\![IF]\!](\sigma_1, \sigma_2) \ \Leftrightarrow \ &\mathcal{M}_{Pred}[\![b_1]\!]\sigma_1 \wedge \mathcal{M}_{Spec}[\![spec_1]\!](\sigma_1, \sigma_2) \\
&\vee \mathcal{M}_{Pred}[\![b_2]\!]\sigma_1 \wedge \mathcal{M}_{Spec}[\![spec_2]\!](\sigma_1, \sigma_2)
\end{aligned}$$

Since we are interested in the non-deterministic case, we let τ_1: *SEStateDen* represent

$$\{\sigma{:}\,\Sigma_\perp \mid \mathcal{M}_{Pred}[\![b_1]\!]\sigma \wedge \mathcal{M}_{Pred}[\![b_2]\!]\sigma\}$$

i.e.

$$\tau_1 = \textit{mk-SEStateDen}(\{[\sigma] \mid \mathcal{M}_{Pred}[\![b_1]\!]\sigma \wedge \mathcal{M}_{Pred}[\![b_2]\!]\sigma\}, 1)$$

Then $LEN(\textit{symbolic-ex}[\![IF]\!]\tau_1) = 2$ and

$$\begin{aligned}
&SEQS(\textit{symbolic-ex}[\![IF]\!]\tau_1) \\
&\quad = \{\sigma\text{-}seq \mid \mathsf{len}\ \sigma\text{-}seq = 2 \wedge \mathsf{front}\ \sigma\text{-}seq \in SEQS(\tau_1)
\end{aligned}$$

$$
\begin{aligned}
& \wedge \mathcal{M}_{Spec}[\![IF]\!](\sigma\text{-}seq[1], \sigma\text{-}seq[2])\} \\
= \; & \{\sigma\text{-}seq \mid \mathsf{len}\, \sigma\text{-}seq = 2 \wedge \mathcal{M}_{Pred}[\![b_1]\!]\sigma\text{-}seq[1] \wedge \mathcal{M}_{Pred}[\![b_2]\!]\sigma\text{-}seq[1] \\
& \wedge (\mathcal{M}_{Pred}[\![b_1]\!]\sigma\text{-}seq[1] \wedge \mathcal{M}_{Spec}[\![spec_1]\!](\sigma\text{-}seq[1], \sigma\text{-}seq[2]) \\
& \vee \mathcal{M}_{Pred}[\![b_2]\!]\sigma\text{-}seq[1] \wedge \mathcal{M}_{Spec}[\![spec_2]\!](\sigma\text{-}seq[1], \sigma\text{-}seq[2]))\} \\
= \; & \{\sigma\text{-}seq \mid \mathsf{len}\, \sigma\text{-}seq = 2 \wedge \mathcal{M}_{Pred}[\![b_1]\!]\sigma\text{-}seq[1] \wedge \mathcal{M}_{Pred}[\![b_2]\!]\sigma\text{-}seq[1] \\
& \wedge (\mathcal{M}_{Spec}[\![spec_1]\!](\sigma\text{-}seq[1], \sigma\text{-}seq[2]) \\
& \vee \mathcal{M}_{Spec}[\![spec_2]\!](\sigma\text{-}seq[1], \sigma\text{-}seq[2]))\}
\end{aligned}
$$

and the non-determinism has been transferred inside the *SEStateDen* τ_2.

4.2 Operational semantics of specifications as used for symbolic execution

This section describes a model of symbolic execution based on the operational semantics approach. The style of operational semantics used is based on that of Plotkin's "Structured Operational Semantics" [Plo81], but of course some of the transitions themselves are rather different since they describe *symbolic* rather than actual execution. However, if there is no danger of confusion, I shall in future not explicitly mention that I am dealing with the particular version of operational semantics used for symbolic execution, but just talk about operational semantics.

The following discussion starts off with the underlying data structure used, and then shows a number of transitions and rules for various language constructs.

There is an important difference between the descriptions of the denotational and operational semantics of symbolic execution. While it is possible to explicitly define the denotational semantics of symbolic execution itself by expressing them in terms of the denotational semantics of the language used, this is not possible for the operational semantics. Instead, one here has to provide a different version of the operational semantics of the language, specifically for symbolic execution. This paper does not try to provide the complete operational semantics for any language, but shows the rules for a number of important language constructs instead.

4.2.1 The data structure

States as used on the operational level will be called *SEStateOps* — *S*ymbolic *E*xecution *States* as used for *Op*erational semantics. In *SEStateOps*, the information derived by symbolic execution should get associated with those identifiers whose values are described by it. For this reason, *SEStateOps* use maps from *Name* to the relevant information. The easiest way to model this relevant information seems to be to model it as predicates. These predicates must be predicates on *sequences* of states rather than single states, since they should model the relationship between different states. Such predicates are introduced as *PredS* below. These are the predicates the user should actually get to see as description values of variables at any stage in the symbolic execution. A *PredS* then is any expression whose semantics can be given as

$\mathcal{M}_{PredS}$: *PredS* → *StateSeq* → $\mathbb{B}$

where *StateSeq* is defined as

StateSeq = seq of $\Sigma_\perp$ | *StateSeq*

StateSeq is defined recursively rather than just as a sequence of states in order to be able to handle blocks and loops, as described below. This decision does not seriously affect the definition of *PredS*.

The language of *PredS* has to include constant symbols true and false, and operator symbols for $\wedge$, $\Rightarrow$, $\Leftrightarrow$ (all with their standard interpretation), and a conditional provided-then (as defined in §4.2.4).

The only condition on the internal structure of *PredS* is that it must be possible to define a function

mentions: *PredS* → set of *Name*

which collects the identifiers mentioned in a given *PredS* into a set. No other conditions are needed since symbolic execution itself makes almost no use of the information contained in the *PredS*, only *simplification* needs to know about the syntax and semantics of *PredS*. (In particular, it needs to know when two *PredS* are equivalent.) The definitions of the syntax and semantics of *PredS* are therefore given in a theory which is used to instantiate symbolic execution for a particular specification language (and thus for a particular language of *PredS*), but are not used in the model of symbolic execution itself. These simplification theories are described in §5.3.2.

Since allowing *sets of PredS* rather than only individual *PredS* as description values makes it easier to combine different *PredS* and, when needed (for example for simplification), split the result again to get its components, *SEStateOp*s are modelled using maps from *Name* to set of *PredS*.

An additional complication arises because each symbolic execution step gives rise to a new predicate on sequences of states, and obviously each such predicate may provide valuable information that should be associated with the appropriate identifier and the appropriate execution step. Therefore, *SEStateOp*s will be defined as *sequences* of maps from identifiers to sets of predicates on sequences of states. An *SEStateOp* thus stores a *history* of the results of symbolic execution.

In this history a loop should be considered as a single step, even though it may really consist of any number of steps (including 0). Therefore, the result of the loop is modelled as an *SEStateOp* itself, which is then considered as one step in the original *SEStateOp*. Similarly, blocks should be considered as a single step and are therefore also modelled as an *SEStateOp* themselves. This leads to the recursive definition of *SEStateOp* given in Figure 4.3. One might thus consider an *SEStateOp* as a tree, where the leaves of the tree are maps and the inner nodes are *SEStateOp*s. Pre-order traversal of this tree describes the execution sequence modelled by the (root) *SEStateOp*.

In addition to the sequence described above, *SEStateOp* contains a field *INDEX* which stores the index or position of this *SEStateOp* in the recursive definition — this will be needed to get the right description values in the *SEStateOp*. Since these express properties of sequences of states, they need to know which sequence of states they should refer to. This issue should become clearer in the discussion of simplification in §4.2.4 and with the example transition for VDM-operations given in §4.2.6, where *INDEX* will actually be needed. The invariant on *SEStateOp*

Define

$Index = \text{seq of } \mathbb{N}_1$

A state as used for describing the operational semantics of a language for symbolic execution is defined recursively as

$SE\text{-}map = \text{map } Name \text{ to set of } PredS$

$SE\text{-}elem = SE\text{-}map \mid SEStateOp$

$$\begin{aligned} SEStateOp :: \quad & SEQ : \text{seq of } SE\text{-}elem \\ & INDEX : Index \end{aligned}$$

where

$$\begin{aligned} & inv\text{-}SEStateOp(mk\text{-}SEStateOp(Seq, ix)) \triangleq \\ & \quad Seq \neq [\,] \\ & \quad \wedge\, \text{hd } Seq : SE\text{-}map \\ & \quad \wedge\, \forall k \leq \text{len } Seq \cdot \\ & \qquad Seq[k] : SEStateOp \Rightarrow INDEX(Seq[k]) = \text{cons}(k, ix) \end{aligned}$$

The denotation of an *SEStateOp* is given by

$$\mathcal{M}_{SEStateOp} : SEStateOp \to SEStateDen$$

$$\begin{aligned} & \mathcal{M}_{SEStateOp}[\![S]\!] \triangleq \\ & \quad mk\text{-}SEStateDen(\{[\sigma] \mid satisfies\text{-}all\text{-}restrictions([\sigma], S, 1)\}, 1) \\ & \text{pre len } SEQ(S) = 1 \end{aligned}$$

$$\begin{aligned} & \mathcal{M}_{SEStateOp}[\![mk\text{-}SEStateOp(Seq \curvearrowright e, ind)]\!] \triangleq \\ & \quad \text{let } S_1 = mk\text{-}SEStateOp(Seq \curvearrowright e, ind) \text{ in} \\ & \quad \text{let } S_2 = mk\text{-}SEStateOp(Seq, ind) \text{ in} \\ & \quad mk\text{-}SEStateDen\Big(\\ & \qquad \{\sigma\text{-}seq : \text{seq of } \Sigma_\perp \mid \\ & \qquad\quad satisfies\text{-}all\text{-}restrictions(\sigma\text{-}seq, S_1, \text{len } Seq + 1) \\ & \qquad\quad \wedge \big(\text{front } \sigma\text{-}seq \in SEQS(\mathcal{M}_{SEStateOp}[\![S_2]\!]) \\ & \qquad\qquad \wedge\, \text{len } \sigma\text{-}seq = \text{len } Seq + 1 \\ & \qquad\quad \vee\, \sigma\text{-}seq \in SEQS(\mathcal{M}_{SEStateOp}[\![S_2]\!]) \\ & \qquad\qquad \wedge\, \text{len } \sigma\text{-}seq = \text{len } Seq \\ & \qquad\qquad \wedge\, \neg\exists \sigma : \Sigma_\perp \cdot satisfies\text{-}all\text{-}restrictions(\\ & \qquad\qquad\qquad\qquad \sigma\text{-}seq \curvearrowright \sigma, S_1, \text{len } Seq + 1) \\ & \qquad\quad \vee\, \sigma\text{-}seq \in SEQS(\mathcal{M}_{SEStateOp}[\![S_2]\!]) \\ & \qquad\qquad \wedge\, \text{len } \sigma\text{-}seq < \text{len } Seq\big)\}, \\ & \qquad \text{len } Seq + 1\Big) \end{aligned}$$

Figure 4.3: Operational semantics of symbolic execution — State

ensures that every *SEQ*(*S*) has a first element which defines the allowed parameter states. An *SEStateOp* itself would not be allowed as first element because it should only arise *as a result of* symbolically executing a specification (usually a loop or block). Additionally, the invariant ensures that *SEStateOp* describes the intuition behind *INDEX* as described above — the *INDEX* of any *SEStateOp* which is the k-th element of *SEQ* of the *SEStateOp* S is the *INDEX* ix of S with k added at the front, or cons(k, ix).

The valuation function $\mathcal{M}_{SEStateOp}$ maps an *SEStateOp* to an *SEStateDen*, where the resulting *SEStateDen* contains those sequences of states that satisfies all the predicates in the *SEStateOp*. This is expressed using the following notation and auxiliary functions:

$\curvearrowright$ denotes adding of an element to the end of a sequence: $seq \curvearrowright e \triangleq seq \frown [e]$.

satisfies-restriction takes a sequence of states σ-*seq* and a *PredS* *ps* and checks whether σ-*seq* satisfies *ps*. Any restriction on a state σ-*seq*[k] where len σ-*seq* $< k$ is considered as satisfied. This function will have to be defined formally by recursion over the syntax of *PredS*. $\tilde{\sigma}$ returns a name for the value of an identifier *nm* at some stage k in an actual execution sequence and is used to refer to that value in a *PredS* (cf. §4.2.4).

satisfies-restriction : seq of $\Sigma_\perp \times$ *PredS* $\times$ *Index* $\rightarrow$ B

satisfies-restriction(σ-*seq*, *ps*, *ix*) $\triangleq$
1. Replace any $\tilde{\sigma}$(cons(k, *ix*), *nm*) in *ps* by σ-*seq*[k](*nm*) ($k \leq$ len σ-*seq*)
2. In the result, replace any atomic formula still containing $\tilde{\sigma}$ by true
   and evaluate

The following function checks that σ-*seq* satisfies all the restrictions imposed by S at level i:

satisfies-restrictions : seq of $\Sigma_\perp \times$ *SEStateOp* $\times \mathbb{N}_1 \rightarrow$ B

satisfies-restrictions(σ-*seq*, S, i) $\triangleq$
if *SEQ*(S)[i]: *SE-map*
then $\bigwedge_{n \in \text{dom } SEQ(S)[i]} \bigwedge_{ps \in SEQ(S)[i](n)}$
satisfies-restriction(σ-*seq*, *ps*, *INDEX*(S))
else $\exists \sigma\text{-}seq' \cdot$ *satisfies-all-restrictions*(
σ-*seq*′, *SEQ*(S)[i], len *SEQ*(*SEQ*(S)[i]))
$\wedge$ len σ-*seq*′ = len *SEQ*(*SEQ*(S)[i])
$\wedge$ σ-*seq*[$i-1$] = hd σ-*seq*′ $\wedge$ σ-*seq*[i] = last σ-*seq*′

pre $i \leq$ len *SEQ*(S)

The function *satisfies-all-restrictions* is defined below. As will be seen, the two functions are mutually recursive.

The function *is-legal-sequence* arises from the conditions on $\mathcal{M}_{Spec}$ and is a (very weak) check that σ-*seq* can actually arise from a sequence of executions.

is-legal-sequence(σ-*seq*) $\triangleq$
$\forall i <$ len σ-*seq* $\cdot$ σ-*seq*[i] = $\perp$ $\Rightarrow$ σ-*seq*[$i+1$] = $\perp$

Finally, *satisfies-all-restrictions* checks that *all* restrictions up to level j imposed by S are satisfied and the sequence 'is legal':

$$satisfies\text{-}all\text{-}restrictions : \text{seq of } \Sigma_\perp \times SEStateOp \times \mathbb{N}_1 \to \mathbb{B}$$

$$satisfies\text{-}all\text{-}restrictions(\sigma\text{-}seq, S, j) \triangleq \bigwedge_{i=1}^{j} satisfies\text{-}restrictions(\sigma\text{-}seq, S, i) \wedge is\text{-}legal\text{-}sequence(\sigma\text{-}seq)$$

$$\text{pre } j \leq \text{len } SEQ(S)$$

We now discuss some of the properties of $\mathcal{M}_{SEStateOp}$. The first one follows immediately from the definitions:

Lemma 4.2.1 *For all S: SEStateOp*

$$LEN(\mathcal{M}_{SEStateOp}[\![S]\!]) = \text{len } SEQ(S)$$

Theorem 4.2.2 *The valuation function $\mathcal{M}_{SEStateOp}$ is total.*

Proof One needs to show that, for any S: *SEStateOp*, $\mathcal{M}_{SEStateOp}[\![S]\!]$: *SEStateDen* exists. To do so, one needs to show that $\mathcal{M}_{SEStateOp}[\![S]\!]$ satisfies the invariant *inv-SEStateDen*. This is done in Appendix A.2. □

In Figure 4.1 we described how an *SEStateDen* can represent a predicate on states (expressed there as a set of states). Similarly, one can represent such predicates by *SEStateOp*s. Given φ: *Pred*, let Φ be the *PredS*

$$\mathcal{M}_{Pred}[\![\varphi]\!]\{n \mapsto \bar{\sigma}([1], n) \mid n: Name\}$$

and let

$$S(\varphi) \triangleq mk\text{-}SEStateOp([\{n \mapsto \{\Phi\} \mid n: Name\}], [\,])$$

Then $\mathcal{M}_{SEStateOp}[\![S(\varphi)]\!]$ is the *SEStateDen* that represents φ, and we say that $S(\varphi)$ is the *SEStateOp* that represents φ. Of course, Φ does not have to be associated with each *Name* n, one could alternatively only associate it with those n that are mentioned in Φ, or even only with one arbitrary n.

The valuation function of *SEStateOp*, like the others defined before, could also be considered as a *retrieve* function [Jon86, pages 204ff]. In this case, it has an adequacy proof obligation associated with it.[2] If *Val* is finite, then it depends on the expressiveness of *PredS* whether $\mathcal{M}_{SEStateOp}$ satisfies this obligation. For infinite *Val*, however, there are uncountably many sets of state sequences and therefore uncountably many *SEStateDen*. On the other hand, there are only countably many *SEStateOp* and therefore *SEStateOp* cannot be adequate w.r.t. $\mathcal{M}_{SEStateOp}$.

So far we have always allowed the *PredS*-conditions inside an *SEStateOp* to refer to *any* element of a state sequence, including future ones. This will cause some problems when adding

[2] A representation *Rep* is *adequate* with respect to a retrieve function $retr: Rep \to Abs$ iff $\forall a \in Abs \cdot \exists r \in Rep \cdot retr(r) = a$

another element to the sequence *SEStateOp*, so for example in the transition for VDM-operations in §4.2.6. Earlier conditions on the current state σ may destroy the faithfulness (see Definition 3.5.1) of that transition. We therefore define the following property which ensures that this problem does not arise. The definition uses the auxiliary function

$$\textit{highest-index}: \textit{PredS} \to \textit{Index}$$

which finds the highest index *ix* s.t. for some *n*: *Name* and some *ix-seq*: seq of $\mathbb{N}$ with hd *ix-seq* = *ix*, $\tilde{\sigma}(\textit{ix-seq}, n)$ occurs in a *PredS*. This function has to be defined recursively over the syntax of *PredS*.

Definition 4.2.3 *mk-SEStateOp(Seq, ind) is* well-behaved *iff*

$$\forall i \leq \text{len}\, \textit{Seq} \cdot \textit{Seq}[i]: \textit{SE-map} \Rightarrow \bigwedge_{n \in \text{dom}\, \textit{Seq}[i]} \bigwedge_{ps \in \textit{Seq}[i](n)} \textit{highest-index}(ps) \leq i$$

The main motivation for the definition of well-behaviour is captured by the following lemma:

Lemma 4.2.4 *Let* len σ-*seq* $\geq i$. *If S*: *SEStateOp is well-behaved, then for all* $\sigma: \Sigma_\perp$

$$\textit{satisfies-restrictions}(\sigma\textit{-seq}, S, i) \Leftrightarrow \textit{satisfies-restrictions}(\sigma\textit{-seq} \frown \sigma, S, i)$$

Proof "$\Rightarrow$" Follows from the well-behaviour of *S*

"$\Leftarrow$" Follows directly from the definition of *satisfies-restrictions*. □

We now introduce the function *collect-preds*, which collects into a set all the *PredS* in a given *SEStateOp*, up to a certain element (given as argument *ix*) in the execution sequence of *S*. If *ix* is empty, then *all PredS* in the *SEStateOp* are collected:

$$\textit{collect-preds} : \textit{SEStateOp} \times \textit{Index} \to \text{set of } \textit{PredS}$$

$$\begin{aligned}
&\textit{collect-preds}(\textit{mk-SEStateOp}(S, \textit{ind}), \textit{ix}) \triangleq \\
&\quad \text{let } \textit{ix}' = \text{if } \textit{ix} = [\,] \text{ then } [\text{len}\, \textit{SEQ}(S)] \text{ else } \textit{ix} \text{ in} \\
&\quad \bigcup_{i=1}^{\text{last } \textit{ix}'} \Big(\text{if } \textit{SEQ}(S)[i]: \textit{SE-map} \\
&\qquad \text{then } \bigcup_{n \in \text{dom } \textit{SEQ}(S)[i]} \textit{SEQ}(S)[i](n) \\
&\qquad \text{else if } i = \text{last } \textit{ix}' \\
&\qquad\quad \text{then } \textit{collect-preds}(\textit{SEQ}(S)[i], \text{front } \textit{ix}') \\
&\qquad\quad \text{else } \textit{collect-preds}(\textit{SEQ}(S)[i], [\,]) \Big)
\end{aligned}$$

$$\begin{aligned}
&\text{pre if } \textit{ix} \neq [\,] \\
&\quad \text{then last } \textit{ix} \leq \text{len } \textit{SEQ}(S) \\
&\qquad \wedge \text{ if } \textit{SEQ}(S)[\text{last } \textit{ix}]: \textit{SEStateOp} \\
&\qquad\quad \text{then } \textit{pre-collect-preds}(\textit{SEQ}(S)[\text{last } \textit{ix}], \text{front } \textit{ix}) \\
&\qquad\quad \text{else front } \textit{ix} = [\,] \\
&\quad \text{else true}
\end{aligned}$$

4.2.2 A syntactic view of symbolic execution

It is not immediately clear from the above how *SEStateOps* relate to the conventional concept of symbolic execution, where identifiers take symbolic values. Consider an identifier **int** x. Possible kinds of values of x include

actual values (ground terms): these are the 'usual' values as used in actual execution. The identifier x has value c for some $c \in Val$ at stage i of the *SEStateDen* τ iff

$$\forall \sigma\text{-}seq \in SEQS(\tau) \cdot \sigma\text{-}seq[i](x) = c$$

Accordingly, this is represented by the *PredS* $\tilde{\sigma}([i], x) = c$. In the appropriate S: *SEStateOp* we then get that the *PredS* $\tilde{\sigma}([i], x) = c$ is in $S[i](x)$.

symbolic values (terms containing symbols denoting identifiers): for example $x = 2 * y - 1$ is a possible symbolic value of the identifier x. Symbolic values denote a whole range of input values but possibly restricted to those of a particular form (odd numbers for x in the above example). They are distinguished by the fact that they express the value of an identifier (x in the above example) as an explicit function of the values of other identifiers (y in the example).

The identifier x has value $f(y)$ at stage i of the *SEStateDen* τ iff

$$\forall \sigma\text{-}seq \in SEQS(\tau) \cdot \sigma\text{-}seq[i](x) = f(\sigma\text{-}seq[i](y))$$

These two kinds of values in symbolic execution are the ones used in most symbolic execution systems. However, they are too restricted for dealing with specifications, since they cannot deal with values that are defined implicitly, or underdefined. Therefore we introduce

description values: a variable, and in particular the output variable, may have a *predicate* as a value, which describes the value implicitly, rather than a term describing it explicitly. Such a predicate is called a description value, it may describe a set of states as associated with an identifier in a *SEStateDen*.

These description values are general predicates of type *PredS*, while both actual values and symbolic values can be considered as special cases of description values and therefore of *PredS*.

The most general results would be achieved by letting S: *SEStateOp* describe the results of (actual) execution starting with the set $\Sigma_\perp$ of all states. For practical reasons, however, one will usually have to cut down the complexity of the output term by (interactively) restricting the admissible universes of the variables used, in extreme cases even restricting it to just one element, i.e. to mix symbolic and actual execution. In S, such a restriction just has the effect of adding another constraint *ps*: *PredS* at the last element of S. Although in theory it does not matter for which n: *Name ps* is added to $S[i](n)$, in practice one will probably want to add it to all those n which are mentioned in the constraint *ps*.

In some cases, it might be more useful to show only part of the information gained from symbolic execution, for example to ignore a more general description such as an invariant and

only show those parts of the information about the output that arise from the execution itself. In this case, the information that is not shown should be hidden behind "...", so that the user can always get to it again and "unhide" it. Eliminating the information rather than just hiding it would lead to *weak* symbolic execution. §6.2.5 describes hiding of information in more detail.

4.2.3 Transitions and rules

In the following I am going to define the kind of transitions and rules used for describing the operational semantics of language constructs in general, and then give the appropriate transitions and rules for various constructs. In many cases (e.g. the rule for `if-then-else`), the transitions and rules of the operational semantics of various language constructs are defined by translating them into an equivalent construct in the language used for describing the results, and then simplifying the result whenever possible. This simplification will hopefully help to eliminate the construct from the description.

From the point of view of their purpose, one can therefore distinguish three different kinds of transitions:

- Transitions describing (state-changing) specifications, like the one in §4.2.6 describing VDM-operations. Since such operations actually lead to a new state, they are described by transitions that extend a S: *SEStateOp* by adding another element to the sequence *SEQ*(S).
- Transitions that eliminate combinators for specifications by translating them into equivalent constructs used inside *PredS* expressions. As an example, consider the rule for `if-then-else` given in §4.2.7.
- Simplification transitions derived from the theory for *PredS*, as discussed in §4.2.4. The transition $S_1 \hookrightarrow S_2$ is allowed if S_2 can be derived from S_1 by simplification of *PredS* only.

We now define the various components that are needed to express transitions. *SpecName* is the type of specification names, and *SpecMap* associates specification names with specifications:

SpecMap = map *SpecName* to *Spec*

Configurations consist of a sequence of *SpecName*s (which may be empty) and an *SEStateOp*:

Conf :: *SNSEQ* : seq of *SpecName*
STATE : *SEStateOp*

A configuration *mk-Conf*(*sn-seq*, S) will be written as $\langle$*sn-seq*, $S\rangle$.

The configuration $\langle$*sn-seq*, $S\rangle$: *Conf* describes the fact that the sequence of specifications given by *sn-seq* is to be applied to S. Given some *sm*: *SpecMap*, the denotation of a configuration is therefore defined as below, using the auxiliary function *evalseq* which, given a sequence *a-seq* and a function f on its elements, applies f to all the elements of *a-seq*:

$evalseq : (A \to B) \times \text{seq of } A \to \text{seq of } B$

$evalseq(f, a\text{-}seq) \triangleq$ if $a\text{-}seq = [\,]$
then $[\,]$
else $\text{cons}(f(\text{hd } a\text{-}seq), evalseq(f, \text{tl } a\text{-}seq))$

$\mathcal{M}_{Conf} : Conf \to SEStateDen$

$\mathcal{M}_{Conf}[\![\langle sn\text{-}seq, S\rangle]\!] \triangleq symbolic\text{-}ex\text{-}s[\![evalseq(sn\text{-}seq, sm)]\!](\mathcal{M}_{SEStateOp}[\![S]\!])$

Transitions are defined as

$Trans = \biguplus_E E \times E$

where $\uplus$ denotes disjoint union, and E ranges over *Conf* and the different syntactic categories of the specification language such as *Expr*. A transition $mk\text{-}Trans(e_1, e_2)$ will be written as $e_1 \hookrightarrow e_2$. $\langle Op_1, S_1\rangle \hookrightarrow \langle Op_2, S_2\rangle$ denotes the fact that one interpretation step transforms $\langle Op_1, S_1\rangle$ into $\langle Op_2, S_2\rangle$, but $\hookrightarrow$ will also be used to denote its transitive-reflexive closure.
Rules take the form

Rule :: *hyps* : set of (*Trans* | *PredS*)
conc : *Trans*

This fits with the definition of rules (or rule statements) in FRIPSE, (cf. Appendix C), since both *Trans* and *PredS* are special forms of *Assertions*.

An important general rule that shows how symbolic execution of a sequence of specifications can be split up into symbolic execution of its elements is the following:

Rule 4.2.5

$$\frac{\langle [sn], S\rangle \hookrightarrow \langle [\,], S'\rangle}{\langle \text{cons}(sn, sn\text{-}seq), S\rangle \hookrightarrow \langle sn\text{-}seq, S'\rangle}$$

Lemma 4.2.6 *Rule 4.2.5 preserves faithfulness: if the hypothesis transition is faithful, then so is the conclusion.*

Proof Let *sm*: *SpecMap* be given. Assume that

$$\langle [\![sn]\!], S\rangle \hookrightarrow \langle [\,], S'\rangle$$

is faithful. This implies that

$$symbolic\text{-}ex[\![sm(sn)]\!](\mathcal{M}_{SEStateOp}[\![S]\!]) = \mathcal{M}_{SEStateOp}[\![S']\!]$$

Then

$$\begin{aligned}
&\mathcal{M}_{Conf}[\![\langle \text{cons}(sn, sn\text{-}seq), S\rangle]\!] \\
&\quad = symbolic\text{-}ex\text{-}s[\![evalseq(\text{cons}(sn, sn\text{-}seq), sm)]\!](\mathcal{M}_{SEStateOp}[\![S]\!]) \\
&\quad = symbolic\text{-}ex\text{-}s[\![evalseq(sn\text{-}seq, sm)]\!](symbolic\text{-}ex[\![sm(sn)]\!](\mathcal{M}_{SEStateOp}[\![S]\!]))
\end{aligned}$$

$$= \textit{symbolic-ex-s}[\![\textit{evalseq}(\textit{sn-seq}, \textit{sm})]\!](\mathcal{M}_{\textit{SEStateOp}}[\![S']\!])$$

$$= \mathcal{M}_{\textit{Conf}}[\![\langle \textit{sn-seq}, S' \rangle]\!]$$

as required. □

4.2.4 Simplification

Now assume we are given a fixed specification language $\mathcal{L}$. To reason about *PredS*, for example to decide whether a *PredS* ps_1 can be simplified to ps_2, one needs a suitable theory of *PredS*. This theory, which will be called $\widetilde{Th(\mathcal{L})}$, needs to be based on the theory used to reason about terms in $\mathcal{L}$, but additionally an indexing mechanism is needed to differentiate between the values of program variables (identifiers or names) at different stages in an execution sequence. To do so, sequences $(\sigma_i)_i$ of states are introduced, where $\sigma_i: \Sigma_\perp$. Since the definition of *SEStateOp* is recursive, simple sequences are not enough — we actually need iterated sequences where σ_i might be a sequence of states itself. This is modelled by introducing a function $\tilde{\sigma}$, which returns the name of the value of the identifier n at a given stage in the execution, with the signature

$\tilde{\sigma}$: *Index* × *Name* → *Val-ref*

For simplicity, we shall in the following identify the element $i: \mathbb{N}_1$ with the index $[i]$.
Now a *PredS* is a predicate that contains names of values of identifiers *at some stage*, instead of the identifiers themselves. See §5.3.2 for a more detailed explanation of *PredS*. The resulting theory of *PredS* is the theory used for simplification: ps_1: *PredS* inside some *SEStateOp* can be simplified to ps_2: *PredS* if they are equivalent in $\widetilde{Th(\mathcal{L})}$. Weak simplification, as used in weak symbolic execution, requires that ps_1 implies ps_2 in $\widetilde{Th(\mathcal{L})}$.

The language of $\widetilde{Th(\mathcal{L})}$ has to include the provided-then construct on *PredS*, which is used for expressing predicates with pre-conditions. The following should hold

(provided true then φ) $\Leftrightarrow$ φ

and

(provided false then φ) $\Leftrightarrow$ true

The reason for not expressing "provided φ then ψ" as "if φ then ψ else true" is that one may want to treat unsatisfied pre-conditions differently and for example provide a warning message. The *denotational* semantics of both expressions are the same.

4.2.5 Block structures and local variables

We start off the description of operational semantics of language constructs with some rules describing block structures. The approach taken, for example, by Plotkin [Plo81] for operational semantics of *actual* execution of blocks and local variable declarations is not adequate here, since it discards information about earlier states, only the current values of variables being stored. In symbolic execution, this is not sufficient since the predicates describing a current value of a variable in general refer to earlier values, therefore the whole history needs to be preserved.

Therefore, as mentioned before, blocks will be modelled by *SEStateOps* that are *elements* of the sequence *SEQ* of the original *SEStateOp*. In order to be able to describe how this is done, the following auxiliary functions will be needed:

$$\textit{current-names} : \textit{SEStateOp} \to \text{set of } \textit{Name}$$

$$\begin{aligned}\textit{current-names}(S) \triangleq\ & \text{if last } \textit{SEQ}(S) \text{: } \textit{SE-map} \\ & \text{then dom last } \textit{SEQ}(S) \\ & \text{else dom hd } \textit{SEQ}(\text{last } \textit{SEQ}(S))\end{aligned}$$

and

$$\textit{add-to-SEStateOp} : \textit{SEStateOp} \times \textit{SE-elem} \to \textit{SEStateOp}$$

$$\textit{add-to-SEStateOp}(S, e) \triangleq \textit{mk-SEStateOp}(\textit{SEQ}(S) \frown e, \textit{INDEX}(S))$$

The function *start-block* starts a new block by creating a new *SEStateOp* which is then added as a new element to the sequence *SEQ* of the current one. *SEQ* of the new *SEStateOp* only consists of one element which describes that "nothing changes" — all identifiers keep the same value that they had before.

$$\textit{start-block} : \textit{SEStateOp} \to \textit{SEStateOp}$$

$$\begin{aligned}&\textit{start-block}(S) \triangleq \\ &\quad \text{let } S' = \textit{mk-SEStateOp}\Big([\{n \mapsto \{\bar{\sigma}([1, \text{len } \textit{SEQ}(S) + 1] \frown \textit{INDEX}(S), n) = \\ &\qquad\qquad \bar{\sigma}([\text{len } \textit{SEQ}(S)] \frown \textit{INDEX}(S), n)\} \\ &\qquad\qquad \mid n \in \textit{current-names}(S)\}], \\ &\qquad \text{cons}(\text{len } \textit{SEQ}(S) + 1, \textit{INDEX}(S))\Big) \text{ in} \\ &\quad \textit{add-to-SEStateOp}(S, S')\end{aligned}$$

$$\textit{finish-block} : \textit{SEStateOp} \to \textit{SEStateOp}$$

$$\begin{aligned}&\textit{finish-block}(S) \triangleq \\ &\quad \text{let } m = \{n \mapsto \{\bar{\sigma}([\text{len } \textit{SEQ}(S)] \frown \textit{INDEX}(S), n) = \bar{\sigma}(\textit{INDEX}(S), n)\} \\ &\qquad \mid n \in \text{dom hd } (\textit{SEQ}(S))\} \text{ in} \\ &\quad \textit{add-to-SEStateOp}(S, m)\end{aligned}$$

The rule for describing the operational semantics of a block is then given by

Rule 4.2.7

$$\frac{\langle \textit{sn-seq}, \text{last } \textit{SEQ}(\textit{start-block}(S))\rangle \hookrightarrow \langle [\,], S'\rangle}{\langle \texttt{begin } \textit{sn-seq} \texttt{ end}, S\rangle \hookrightarrow \langle [\,], \textit{add-to-SEStateOp}(S, \textit{finish-block}(S'))\rangle}$$

where `begin` *sn-seq* `end` is used as the *name* of the appropriate sequence of specifications. A similar convention will be used for other constructs below.

The functions *start-block* and *finish-block* are not just auxiliary functions for expressing these rules, but will also be used in the specification of the UI of SYMBEX, in order to be able to display

the newly-started *SEStateOp*, which represents the block (or, similarly, a loop), as an element of *SEQ* of the old one. As long as discharging the hypotheses in a rule such as Rule 4.2.7 above can be done automatically, one does not need such a special mechanism, but if user interaction is required then one needs to display *some* of the results *before* the hypothesis has been fully discharged. In this case, the functions *start-block* and *finish-block* should be used to "tell the system" that it is dealing with a block which should be displayed accordingly (cf. page 100).

Like provided-then, these functions therefore have to be part of the language of *PredS* (cf. pages 78 and 97). See §7.3 for an example of the UI of a block started by a `while`-loop.

Declarations of local variables are handled by mapping them to the empty set of restrictions and keeping all other variables equal:

Rule 4.2.8

$$\begin{aligned}
&\vdash \langle [\texttt{var}\ x{:}\mathit{Type}], \mathit{mk\text{-}SEStateOp}(\mathit{Seq}, \mathit{ind})\rangle \\
&\quad \hookrightarrow \langle [\,], \mathit{mk\text{-}SEStateOp}(\\
&\qquad \mathit{Seq} \curvearrowright \{n \mapsto \text{if } n = x \\
&\qquad\qquad \text{then } \{\,\} \\
&\qquad\qquad \text{else } \{\bar{\sigma}(\text{cons}(\text{len}\,\mathit{Seq}+1, \mathit{ind}), n) = \bar{\sigma}(\text{cons}(\text{len}\,\mathit{Seq}, \mathit{ind}), n)\}\} \\
&\qquad \mathit{ind})\rangle
\end{aligned}$$

4.2.6 Operational semantics of VDM-operations

Given a VDM-specification of an operation

$$\begin{aligned}
&\mathit{Op}\ \ (a{:}T_1)\ r{:}T_2 \\
&\text{ext rd } \mathit{er}\ \ : T_3 \\
&\quad \text{wr } \mathit{ew}\ : T_4 \\
&\text{pre } \varphi(a, \mathit{er}, \mathit{ew}) \\
&\text{post } \psi(a, \overleftarrow{\mathit{ew}}, r, \mathit{er}, \mathit{ew})
\end{aligned}$$

This specification is actually the concrete representation of an object of type (from BSI-Proto-standard for VDM [And88, BSI88], but without exception handling)

$$\begin{aligned}
\mathit{ImplOpDef} :: \ & \mathit{dom} : \text{seq of } \mathit{NameTypePair} \\
& \mathit{rng} : [\mathit{NameTypePair}] \\
& \mathit{exts} : \text{seq of } \mathit{ExtVarInf} \\
& \mathit{pre} : \mathit{Expr} \\
& \mathit{post} : \mathit{Expr}
\end{aligned}$$

$$\begin{aligned}
\mathit{ExtVarInf} :: \ & \mathit{mode} : (\text{READ}, \text{READWRITE}) \\
& \mathit{rest} : \mathit{NameTypePair}
\end{aligned}$$

namely the element

$$\begin{aligned}&\textit{mk-ImplOpDef}([\textit{mk-NameTypePair}(a, T_1)],\\ &\qquad \textit{mk-NameTypePair}(r, T_2),\\ &\qquad [\textit{mk-ExtVarInf}(\text{READ}, \textit{mk-NameTypePair}(er, T_3)),\\ &\qquad\quad \textit{mk-ExtVarInf}(\text{READWRITE}, \textit{mk-NameTypePair}(ew, T_4))],\\ &\qquad \varphi,\\ &\qquad \psi)\end{aligned}$$

The auxiliary functions used in the following are defined in Appendix B.3. The operational semantics of an operation can be described as in Figure 4.4 (ignoring the modularisation provided in the BSI-Protostandard).

If the language of the theory $\widetilde{Th(\mathcal{L})}$ of *PredS* was not rich enough to express these predicates, one would have to be content with *weak* symbolic execution and use predicates which are *implied* by the those used in the rule in Figure 4.4. However, this language should be derived from LPF in the way described in §4.2.4, in which case it is expressive enough.

Example 4.2.9 Given the specification OP_1 from Example 4.1.2. As before, we assume that the pre-condition holds. Since x and y are the only identifier used, we therefore start with the *SEStateOp*

$$S = \textit{mk-SEStateOp}([\{x \mapsto \{\tilde{\sigma}([1], x) \geq 0\}, y \mapsto \{\tilde{\sigma}([1], x) \geq 0\}\}], [\,])$$

The appropriate instantiation of the rule giving the operational semantics of VDM-operations is then given by (after some simplification)

$$\begin{aligned}&\text{let } m = \left\{ \begin{array}{lcl} x & \mapsto & \{\tilde{\sigma}([2], y)^2 \leq \tilde{\sigma}([1], x) \\ & & \quad \wedge\, \tilde{\sigma}([2], x) = \tilde{\sigma}([1], x) + 1\} \\ y & \mapsto & \{\tilde{\sigma}([2], y) \leq \tilde{\sigma}([1], x)\} \end{array} \right\} \text{ in}\\ &\vdash \langle [OP_1], S\rangle \hookrightarrow \langle [\,], \textit{mk-SEStateOp}(SEQ(S) \curvearrowright m, [\,])\rangle\end{aligned}$$

Theorem 4.2.10 *The transition scheme in Figure 4.4 giving the operational semantics of* VDM-*operations is faithful, provided that S is well-behaved.*

Proof See Appendix A.3 □

4.2.7 Operational semantics of `if-then-else`

Unfortunately, the rule describing the operational semantics of the `if-then-else` combinator as used for symbolic execution turns out to be far more complicated than those used for actual execution as given by Plotkin [Plo81]. This is due to the fact that, as mentioned before, in symbolic execution one has to store the whole *history* of results, not just the current ones, and the recursive definition of states *SEStateOp* needed accordingly. The rule is therefore expressed using a (recursive) auxiliary function that 'merges' two *SEStateOps*, and at the same time turns each *ps*: *PredS* in either of the two *SEStateOps* into the appropriate conditional. The latter is done by *ITE-merge-map*, which is then called by the general function *ITE-merge*. *ITE-merge* has to distinguish nine different cases, since either of the two sequences to be merged may be empty or start with a *SE-map* or start with a *SEStateOp*. Define the auxiliary function

Define the map m: *SE-map* as

$$
m(n) \triangleq \left\{
\begin{array}{l}
\text{let } a = names(dom(Op)) \text{ in} \\
\text{let } r = name(rng(Op)) \text{ in} \\
\text{let } at = types(dom(Op)) \text{ in} \\
\text{let } rt = type(rng(Op)) \text{ in} \\
\text{let } er = readnames(exts(Op)) \text{ in} \\
\text{let } ew = readwritenames(exts(Op)) \text{ in} \\
\text{let } ewt = readwritetypes(exts(Op)) \text{ in} \\
\text{let } rest = current\text{-}names(S) - (\text{rng } a \cup \text{rng } ew \cup \{r\}) \text{ in} \\
\text{let } oldseq = \text{cons}(\text{len } Seq, index) \text{ in} \\
\text{let } old = \lambda x\text{: seq of } Name\cdot \\
\qquad evalseq(\lambda y\text{: } Name \cdot \tilde{\sigma}(oldseq, y), x) \text{ in} \\
\text{let } newseq = \text{cons}(\text{len } Seq + 1, index) \text{ in} \\
\text{let } new = \lambda x\text{: seq of } Name\cdot \\
\qquad evalseq(\lambda y\text{: } Name \cdot \tilde{\sigma}(newseq, y), x) \text{ in} \\
\{inv\_of\_Type(at[i])[nm/\tilde{\sigma}(newseq, nm) \mid nm\text{: } Name]\} \quad \text{if } n = a[i] \quad \} \psi_1 \\
\\
\{\text{provided } pre(new(a), old(er), old(ew)) \quad \text{if } n = ew[i] \\
\text{then } post(new(a), old(er), old(ew), \\
\qquad new(er), new(ew)) \ , \quad \} \psi_2 \\
inv\_of\_Type(ewt[i])[nm/\tilde{\sigma}(newseq, nm) \mid nm\text{: } Name]\} \quad \} \psi_3 \\
\\
\{\text{provided } pre(new(a), old(er), old(ew)) \quad \text{if } n = r \\
\text{then } post(new(a), old(er), old(ew), \\
\qquad new(er), new(ew)) \ , \quad \} \psi_4 \\
inv\_of\_Type[rt][nm/\tilde{\sigma}(newseq, nm) \mid nm\text{: } Name]\} \quad \} \psi_5 \\
\\
\{\text{provided } pre(new(a), old(er), old(ew)) \quad \text{if } n \in rest \\
\text{then } \tilde{\sigma}(newseq, n) = \tilde{\sigma}(oldseq, n)\} \quad \} \psi_6
\end{array}
\right.
$$

(The annotations ψ_i are just used to give a name to the relevant *PredS* and make the proof of Theorem 4.2.10 easier to read.)

Then

$$\langle [Op], S \rangle \hookrightarrow \langle [\,], add\text{-}to\text{-}SEStateOp(S, m) \rangle$$

Figure 4.4: Transition for VDM-operations

$$ITE\text{-}merge\text{-}map : SE\text{-}map \times SE\text{-}map \times PredS \to SE\text{-}map$$

$$ITE\text{-}merge\text{-}map(m_1, m_2, ps) \triangleq$$
$$\{n \mapsto \begin{cases} \{\text{if } ps \text{ then } ps_1 \text{ else true} \mid ps_1 \in m_1(n)\} \\ \cup \{\text{if } ps \text{ then true else } ps_2 \mid ps_2 \in m_2(n)\} & \text{if } n \in \text{dom } m_1 \cap \text{dom } m_2 \\ \{\text{if } ps \text{ then } ps_1 \text{ else true} \mid ps_1 \in m_1(n)\} & \text{if } n \in \text{dom } m_1 - \text{dom } m_2 \\ \{\text{if } ps \text{ then true else } ps_2 \mid ps_2 \in m_2(n)\} & \text{if } n \in \text{dom } m_2 - \text{dom } m_1 \end{cases}\}$$

In the first case, one could alternatively join the two kinds of expressions to get if ps then ps_1 else ps_2, but in general it is not obvious how to select ps_1 and ps_2 to get a useful result.

If the two sequences to be merged have different length, then, by the recursive definition of *ITE-merge* below, one will eventually get into the position where one of the sequences starts with a map, and the other one is empty. This case is handled by

$$ITE\text{-}merge\text{-}empty : SE\text{-}map \times PredS \times Index \to SE\text{-}map$$

$$ITE\text{-}merge\text{-}empty(m, ps, ix) \triangleq$$
$$\{n \mapsto \{\text{if } ps \text{ then } ps_1 \text{ else } '\tilde{\sigma}(ix, n) = \tilde{\sigma}(previous(ix), n)' \mid ps_1 \in m(n)\}\}$$

using the auxiliary function *previous* which, given the index ix of an element of some *SEStateOp*, finds the index of the previous element:

$$previous : Index \to Index$$

$$previous(ix) \triangleq \text{if hd } ix = 1 \text{ then tl } ix \text{ else cons}(\text{hd } ix - 1, \text{tl } ix)$$

$$\text{pre } ix \neq [\,]$$

Now define

$$case : \text{seq of } SE\text{-}elem \to \{\text{EMPTY}, \text{MAP}, \text{SES}\}$$

$$case(e\text{-}seq) \triangleq \text{if } e\text{-}seq = [\,] \text{ then EMPTY else if hd } e\text{-}seq : SE\text{-}map \text{ then MAP else SES}$$

Then

$$ITE\text{-}merge : \text{seq of } SE\text{-}elem \times \text{seq of } SE\text{-}elem \times PredS \times SEStateOp \to SEStateOp$$

$$
\begin{array}{l}
\textit{ITE-merge}(\textit{e-seq}_1, \textit{e-seq}_2, ps, S) \triangleq \\
\quad \text{cases } \textit{case}(\textit{e-seq}_1), \textit{case}(\textit{e-seq}_2) \text{ of} \\
\quad \text{EMPTY, EMPTY} \rightarrow S \\
\quad\quad \text{EMPTY, MAP} \rightarrow \text{let } S_1 = \textit{add-to-SEStateOp}(S, \\
\quad\quad\quad\quad\quad\quad \textit{ITE-merge-empty}(\text{hd } \textit{e-seq}_2, ps, \textit{INDEX}(S))) \text{ in} \\
\quad\quad\quad\quad \textit{ITE-merge}([\,], \text{tl } \textit{e-seq}_2, ps, S_1) \\
\quad\quad \text{EMPTY, SES} \rightarrow \text{let } S_1 = \textit{add-to-SEStateOp}(S, \\
\quad\quad\quad\quad\quad\quad \textit{ITE-merge}([\,], \text{hd } \textit{e-seq}_2, ps, S)) \text{ in} \\
\quad\quad\quad\quad \textit{ITE-merge}([\,], \text{tl } \textit{e-seq}_2, ps, S_1) \\
\quad\quad \text{MAP, EMPTY} \rightarrow \text{let } S_1 = \textit{add-to-SEStateOp}(S, \\
\quad\quad\quad\quad\quad\quad \textit{ITE-merge-empty}(\text{hd } \textit{e-seq}_1, ps, \textit{INDEX}(S))) \text{ in} \\
\quad\quad\quad\quad \textit{ITE-merge}(\text{tl } \textit{e-seq}_1, [\,], ps, S_1) \\
\quad\quad\quad \text{MAP, MAP} \rightarrow \text{let } S_1 = \textit{add-to-SEStateOp}(S, \\
\quad\quad\quad\quad\quad\quad \textit{ITE-merge-map}(\text{hd } \textit{e-seq}_1, \text{hd } \textit{e-seq}_2, ps)) \text{ in} \\
\quad\quad\quad\quad \textit{ITE-merge}(\text{tl } \textit{e-seq}_1, \text{tl } \textit{e-seq}_2, ps, S_1) \\
\quad\quad\quad \text{MAP, SES} \rightarrow \text{let } S_1 = \text{last}(\textit{SEQ}(\textit{start-block}(S))) \text{ in} \\
\quad\quad\quad\quad \text{let } S_2 = \textit{finish-block}(\textit{add-to-SEStateOp}(S_1, \text{hd } \textit{e-seq}_1)) \text{ in} \\
\quad\quad\quad\quad \text{let } S_3 = \textit{ITE-merge}(S_2, \text{hd } \textit{e-seq}_2, ps, S) \text{ in} \\
\quad\quad\quad\quad \textit{ITE-merge}(\text{tl } \textit{e-seq}_1, \text{tl } \textit{e-seq}_2, ps, S_3) \\
\quad\quad\quad \text{SES, EMPTY} \rightarrow \text{let } S_1 = \textit{add-to-SEStateOp}(S, \\
\quad\quad\quad\quad\quad\quad \textit{ITE-merge}(\text{hd } \textit{e-seq}_1, [\,], ps, S)) \text{ in} \\
\quad\quad\quad\quad \textit{ITE-merge}(\text{tl } \textit{e-seq}_1, [\,], ps, S_1) \\
\quad\quad\quad \text{SES, MAP} \rightarrow \text{let } S_1 = \text{last}(\textit{SEQ}(\textit{start-block}(S))) \text{ in} \\
\quad\quad\quad\quad \text{let } S_2 = \textit{finish-block}(\textit{add-to-SEStateOp}(S_1, \text{hd } \textit{e-seq}_2)) \text{ in} \\
\quad\quad\quad\quad \text{let } S_3 = \textit{ITE-merge}(\text{hd } \textit{e-seq}_1, S_2, ps, S) \text{ in} \\
\quad\quad\quad\quad \textit{ITE-merge}(\text{tl } \textit{e-seq}_1, \text{tl } \textit{e-seq}_2, ps, S_3) \\
\quad\quad\quad \text{SES, SES} \rightarrow \text{let } S_1 = \textit{ITE-merge}(\text{hd } \textit{e-seq}_1, \text{hd } \textit{e-seq}_2, ps, S) \text{ in} \\
\quad\quad\quad\quad \textit{ITE-merge}(\text{tl } \textit{e-seq}_1, \text{tl } \textit{e-seq}_2, ps, S_1) \\
\quad \text{end}
\end{array}
$$

Note that this definition implies

$$
\begin{aligned}
&\text{len } \textit{SEQ}(\textit{ITE-merge}(\textit{e-seq}_1, \textit{e-seq}_2, ps, S)) \\
&\quad = \text{len } \textit{SEQ}(S) + \max(\text{len } \textit{e-seq}_1, \text{len } \textit{e-seq}_2)
\end{aligned}
$$

(proof by double induction over len $\textit{e-seq}_1$ and len $\textit{e-seq}_2$).
Now the rule describing the operational semantics of `if-then-else` can be given as

Rule 4.2.11 let $ps(S) = \varphi[n/\tilde{\sigma}(\textit{cons}(\text{len } \textit{SEQ}(S), \textit{INDEX}(S)), n) \mid n\text{: Name}]$ in

$$
\frac{\begin{array}{l} ps(S) \vdash \langle \textit{sn-seq}_1, S\rangle \hookrightarrow \langle [\,], \textit{mk-SEStateOp}(\textit{SEQ}(S) \frown \textit{e-seq}_1, \textit{INDEX}(S))\rangle \\ \neg ps(S) \vdash \langle \textit{sn-seq}_2, S\rangle \hookrightarrow \langle [\,], \textit{mk-SEStateOp}(\textit{SEQ}(S) \frown \textit{e-seq}_2, \textit{INDEX}(S))\rangle \end{array}}{\langle [\texttt{if } \varphi \texttt{ then } sn_1 \texttt{ else } sn_2], S\rangle \hookrightarrow \langle [\,], \textit{ITE-merge}(\textit{e-seq}_1, \textit{e-seq}_2, ps(S), S)\rangle}
$$

assuming that the language of the simplification theory $\widetilde{Th(\mathcal{L})}$ (cf. §4.2.4) has the connective if-then-else. Here "`if` φ `then` sn_1 `else` sn_2" is the *name* of the appropriate specification. Note that the combinator `if-then-else` and the connective if-then-else are different constructs, of different types.

The simplification theory $\widetilde{Th(\mathcal{L})}$ should then contain some rules for handling if-then-else, for example

$$\text{if true then } \psi_1 \text{ else } \psi_2 \Leftrightarrow \psi_1$$

$$\text{if false then } \psi_1 \text{ else } \psi_2 \Leftrightarrow \psi_2$$

and

$$\text{if } ps \text{ then } \psi \text{ else } \psi \Leftrightarrow \psi$$

Of course one could additionally introduce two rules that handle the case when either $ps(S)$ or $\neg ps(S)$ is known to hold. Although these rules are not strictly speaking necessary since they can be derived from the above (assuming that the operational semantics given always allow one to find $e\text{-}seq_1$ and $e\text{-}seq_2$), they would save a lot of simplification work.

An example of the application of these rules for `if-then-else` is given as part of the example of a `while`-loop in §7.3.

4.2.8 Operational semantics of `while`-loops

Similar to block structures, loops are considered as a single step even though their execution may consist of any number of steps. This is achieved by describing the results of this execution sequence in a different *SEStateOp* or block which is then considered as a single step in the original *SEStateOp*. However, there is an additional complication in that with the usual approach to operational semantics, using a rule like

$$\langle \texttt{while } \varphi \texttt{ do } [\![spec]\!] \texttt{ od}, S\rangle \hookrightarrow \langle \texttt{if } \varphi \texttt{ then } ([\![spec]\!]\texttt{;while } \varphi \texttt{ do } [\![spec]\!] \texttt{ od}) \texttt{ else skip}, S\rangle$$

it is not clear when encountering a `while`-statement whether to start a new block (because it is a new `while`-statement) or continue the current one (because it is a new iteration of a previously encountered statement). Therefore we introduce two different versions of the `while`-statement that allow one to distinguish the two. `while-do` is the 'proper' statement that starts a new block, and *WHILE-DO* is an auxiliary version that is used to continue the current block.

This leads to the rules (again identifying the names of specifications with the specifications they name)

Rule 4.2.12

$$\frac{\langle WHILE\ \varphi\ DO\ sn\ OD, \text{last}SEQ(start\text{-}block(S))\rangle \hookrightarrow \langle [\,], S'\rangle}{\langle \texttt{while } \varphi \texttt{ do } sn \texttt{ od}, S\rangle \hookrightarrow \langle [\,], add\text{-}to\text{-}SEStateOp(S, finish\text{-}block(S'))\rangle}$$

Rule 4.2.13

$$\vdash \langle WHILE\ \varphi\ DO\ sn\ OD, S\rangle \hookrightarrow \langle \texttt{if } \varphi \texttt{ then cons}(sn, WHILE\ \varphi\ DO\ sn\ OD) \texttt{ else } [\,], S\rangle$$

An example of symbolic execution of a `while`-loop using these rules is given in §7.3.

4.2.9 Handling non-determinism

As mentioned before (in §4.1.4), in symbolic execution the effects of non-determinism should be captured by the *state* rather than by supplying different transitions that apply to the same configuration. As an example, consider the rule given below describing the operational semantics of the command *IF* (as defined in §4.1.4). Since it is quite similar to the rule for `if-then-else` as given in §4.2.7, only the analogue of *ITE-merge-map* is given here, the other cases are completely analogous to `if-then-else`. In *IF-merge-map*, the case of $n \in \mathrm{dom}\, m_1 \cap \mathrm{dom}\, m_2$ is treated slightly differently from the way it was done in *ITE-merge-map* in order to demonstrate the 'joining' of expressions as mentioned after the definition of *ITE-merge-map*.

$$\textit{IF-merge-map} : \textit{SE-map} \times \textit{SE-map} \times \textit{PredS} \times \textit{PredS} \to \textit{SE-map}$$

$$\textit{IF-merge-map}(m_1, m_2, ps_1, ps_2) \triangleq$$

$$\{n \mapsto \begin{cases} \{\text{if } ps_1 \to ps_1' \,\square\, ps_2 \to ps_2' \text{ fi} \\ \quad \mid ps_1' \in m_1(n) \wedge ps_2' \in m_2(n)\} & \text{if } n \in \mathrm{dom}\, m_1 \cap \mathrm{dom}\, m_2 \\ \{\text{if } ps_1 \to ps_1' \,\square\, ps_2 \to \text{true fi} \\ \quad \mid ps_1' \in m_1(n)\} & \text{if } n \in \mathrm{dom}\, m_1 - \mathrm{dom}\, m_2 \\ \{\text{if } ps_1 \to \text{true} \,\square\, ps_2 \to ps_2' \text{ fi} \\ \quad \mid ps_2' \in m_2(n)\} & \text{if } n \in \mathrm{dom}\, m_2 - \mathrm{dom}\, m_1 \end{cases} \}$$

Rule 4.2.14 let $ps_i(S) = \varphi_i[n/\bar{\sigma}(\mathit{cons}(\mathrm{len}\, \mathit{SEQ}(S), \mathit{INDEX}(S)), n) \mid n{:}\,\mathit{Name}]$ in

$$\frac{\begin{array}{c} m_1, m_2 : \textit{SE-map} \\ ps_1(S) \vdash \langle [sn_1], S\rangle \hookrightarrow \langle [\,], \textit{add-to-SEStateOp}(S, m_1)\rangle \\ ps_2(S) \vdash \langle [sn_2], S\rangle \hookrightarrow \langle [\,], \textit{add-to-SEStateOp}(S, m_2)\rangle \end{array}}{\begin{array}{l} \langle [\texttt{if } \varphi_1 \to sn_1 \,\square\, \varphi_2 \to sn_2 \,\texttt{fi}], S\rangle \\ \quad \hookrightarrow \langle [\,], \textit{add-to-SEStateOp}(S, \textit{IF-merge-map}(m_1, m_2, ps_1(S), ps_2(S)))\rangle \end{array}}$$

Again, as for `if-then-else`, the consequent transition of this rule is used to transform the combinator `if-fi` into the connective if-fi. This connective is then dealt with in $\widetilde{Th(\mathcal{L})}$ by rules such as those below. These simplification rules are slightly more complicated than those for if-then-else since they have to consider the different alternatives in parallel — in symbolic execution it is not enough to know if *one* of the guards is true.

$$\frac{\varphi_1 \wedge \varphi_2}{\text{if } \varphi_1 \to \psi_1 \,\square\, \varphi_2 \to \psi_2 \text{ fi} \Leftrightarrow \psi_1 \vee \psi_2}$$

$$\frac{\varphi_1 \wedge \neg\varphi_2}{\text{if } \varphi_1 \to \psi_1 \,\square\, \varphi_2 \to \psi_2 \text{ fi} \Leftrightarrow \psi_1}$$

$$\frac{\neg\varphi_1 \wedge \varphi_2}{\text{if } \varphi_1 \rightarrow \psi_1 \,\square\, \varphi_2 \rightarrow \psi_2 \text{ fi} \Leftrightarrow \psi_2}$$

4.3 Language genericity

One important and innovative aspect of IPSE 2.5 is its emphasis on *genericity* with respect to the languages and development methods supported. The most important aspect of such genericity in the context of symbolic execution is genericity with respect to the specification language, symbolic execution should be supported for a variety of specification languages rather than a single one. This implies that one has to provide a description of the syntax and semantics of the language (as described in §4.2), and symbolic execution has to be based on these descriptions.

Given the mechanism for describing the (operational) semantics of a language as used for symbolic execution and as provided in §4.2, the question of the mechanism's genericity arises. How generic is this mechanism, and how wide a class of specification languages can therefore be symbolically executed using the approach described in this thesis?

§4.2 showed that the approach is well suited to model-oriented specification languages like VDM or Z, since they are explicitly based on the notion of states and specify a software system in terms of state transformations. The same applies to the GIST specification language.

Furthermore, any programming language fits, on some level, into the description mechanism provided, since its semantics can always be expressed in terms of an interpreter. However, this interpreter might work on a very low level — the hardware level, in the extreme — in which case the results of symbolic execution of a program, based on such an operational semantics, would not help the user to understand the program at all. It is therefore not enough that the semantics of a specification language *can* be expressed in the way described. This must indeed be a *natural* way of expressing them if the resulting instantiation of SYMBEX is to fulfil its intended purpose. Of course, this is a very vague requirement, there is no clear-cut distinction between 'natural' and 'unnatural' descriptions, but this only reflects the fact that the distinction between languages that can usefully be supported and those that cannot is not clear-cut itself. The more natural a language description is, the more likely symbolic execution will be able to help the user in understanding specifications in this language. For example, it is quite natural to express the semantics of VDM or Z in this way, while it would be very unnatural to impose a notion of state on an algebraic specification language (cf. below).

This discussion has so far ignored the fact that the operational semantics needed for symbolic execution are different from those needed for actual execution. However, this does not really matter, since one can mechanically generate the former from the latter: given the 'actual' operational semantics and a 'symbolic' configuration, generate the new symbolic configuration by taking all matching actual transitions and combine their results into a suitable case-statement. These case-statements then form (after some syntactic transformations, in particular introduction of $\tilde{\sigma}$, cf. §4.2.4) the *PredS* description values of identifiers in the new state, and are used to construct the new *SEStateOp* in the resulting configuration. See the rule for `if-then-else` in §4.2.7 for an example. Again, the operational semantics resulting from this process might not be very natural

and therefore not lead to any *useful* results. The process was only described in order to show that for any programming language there is a level of interpretation at which the language may be symbolically executed using the approach described here.

Symbolic execution was originally developed (in systems such as EFFIGY or DISSECT, cf. §2.1.2) for conventional imperative programming languages such as PASCAL or FORTRAN. Any such language can also be symbolically executed in a very natural way under our model of symbolic execution: if the identifier x gets assigned the symbolic value $v(x)$ by 'conventional' symbolic execution, then under our model x gets assigned the description value '$x = v(x)$' (but again first substituting any identifiers y by $\tilde{\sigma}(index, y)$ for the appropriate *index*).

Functional languages such as LISP [All78, §1.2], and logic programming languages such as PROLOG [SS86, §3.5], support some form of symbol processing directly. What influence does this have on the way they can be symbolically executed themselves?

LISP can be said to consist of a functional part ('pure' or 'applicative' LISP) and a non-functional part [All78, §4.1]. The treatment of the non-functional part of LISP is not essentially different from the treatment of imperative languages. The functional part of LISP has to be treated differently, since it does not use the concept of an external state. Instead, variables have values associated with them in association lists or a-lists [All78, §3.3]. One would therefore have to modify the concept of symbolic execution to consider such a-lists as a form of state, which implies that some functions have a side-effect since they change the a-list. With this modification, our model of symbolic execution should be well suited to cope with functional languages such as LISP and provide useful results.

PROLOG does not have an obvious notion of state that the operational semantics could be based on. Under a 'natural' definition of the denotational semantics of PROLOG, given a set A of clauses, an n-ary predicate symbol denotes an n-ary relation over the Herbrand universe of A [vEK76].

There are a number of different concepts of state which could be defined for PROLOG: for example, consider the set of n-tuples known to belong to the relation (i.e. the set of ground clauses known to be provable from A) at any given stage as a state, with a suitable transition induced by the inference system and search strategy (similar to the transformation T used in the fixpoint semantics in [vEK76]).

Slightly closer to the notion of state used in this thesis would be the definition of a state as a substitution that maps variables to terms such that the goal statement is provable from A.

However, none of these concepts of state really fits into the semantic framework provided here, since none uses states in the sense of mappings from program variables to their values. Therefore, it does not seem possible to usefully apply the ideas of this thesis to PROLOG.

Going back to *specification* languages, a different style of specifications is used in algebraic specification languages. These specify a software system as an abstract data type (ADT), described in terms of equations or rewrite rules [EM85]. The natural way to define the semantics of such an ADT is as a class of algebras, using an initial, loose, or final interpretation of the ADT. Again, there is no obvious concept of state involved and algebraic specification languages therefore do not fit into our model of symbolic execution.

So far, none of the languages considered was suitable for the specification of interactive, distributed or concurrent systems, called 'reactive systems' [Pnu86]. As an example of such a

language, consider Hoare's CSP [Hoa85]. In CSP, individual processes can be specified in Z, but CSP extends this to provide a notation for combination and communication of such processes. Since the specifications of individual processes use the notion of state, the semantics of CSP can be expressed in the way described in §4.2: the individual processes denote a relation on states, and CSP can be considered as a (very elaborate) combinator language for such relations. CSP can therefore be executed symbolically in the way described in this thesis.

Another language that fits very well into the framework described here is LARCH — or the family of LARCH-languages, to be exact. Each member of this family has a component common to all members of the family, called the LARCH shared language, and a component particular to this member, called the interface language [GHW85, Piece I]. The interface languages are based on predicate calculus and describe specifications, here called procedures, in terms of a **requires** predicate providing a pre-condition, a **modifies at most** clause giving the variables writable by the procedure, and an **ensures** predicate which is a predicate over two states and provides a post-condition. Such procedures can thus be symbolically executed in just the same way as e.g. VDM operations.

The shared language is essentially an algebraic specification language. It is based on equations which give rise to a collection of theories. These theories in turn define the notion of equality as used in the interface language. In symbolic execution they can be used to do simplification by extending the notion of equality LARCH-terms to *PredS*. Essentially, this means that a large part of the simplification theory $\widetilde{Th(\mathcal{L})}$ for LARCH (as mentioned in §4.2.4 and described in more detail in §5.3.2) is already provided by the shared language.

Chapter 5

Symbolic execution and formal reasoning

"Well," said Owl, "the customary procedure in such cases is as follows."

"What does Crustimoney Proseedcake mean?" said Pooh. "For I am a Bear of Very Little Brain, and long words Bother me."

"It means the Thing to do."

"As long as it means that, I don't mind," said Pooh humbly.

A.A. Milne: Winnie-the-Pooh

5.1 Introduction

This chapter analyses the relationship between (our model of) symbolic execution and formal reasoning. The term "formal reasoning" is used here rather than, say, "theorem proving", in order to emphasize the fact that this analysis is not only concerned with the activity of theorem proving itself, but also with related activities such as structuring and storing of theories.

As shown in §4.2, the operational semantics of a specification language can be expressed in terms of rules and transitions, and transitions can be considered as a particular kind of proposition. The operational semantics then form a theory, and applying rules or transitions from this theory to a configuration is a form of formal reasoning.

Before such theories of operational semantics are investigated in more detail in §5.3, the topic of simplification is taken up again. While §4.2.4 only considered some technical points, §5.2 will discuss the general approach to simplification taken, and consider simplification as a formal reasoning activity.

First, however, a brief introduction to how one can apply the operational semantics of a language to derive the wanted result: assume one is given a configuration $\langle sn\text{-}seq, S \rangle$ and operational

semantics of the relevant language. Such a configuration, as defined in §4.2.3, consists of a sequence *sn-seq*: seq of *SpecName* of specification names and an *SEStateOp S*, and denotes an interpreter configuration in which the sequence of specifications referred to by *sn-seq* is to be applied to *S*. The operational semantics will be expressed as a collection of theories, its details are described in §5.3.

We now want to transform the configuration $\langle sn\text{-}seq, S\rangle$ into an equivalent configuration (under the equivalence relation induced by $\mathcal{M}_{Conf}$) of the form $\langle [\,], S'\rangle$, since this provides the resulting *SEStateOp* S'. This transformation is done by repeatedly applying transitions from the operational semantics to the configuration until it has the right form.

Considered as an object handled by FRIPSE, a transition is an expression of type *Prop* (proposition). The rules of the operational semantics correspond to axioms or rules as defined in the specification of FRIPSE (cf. Appendix C), and thus consist of zero or more assertions (possibly transitions) and sequents as hypotheses and an assertion which is its conclusion, and, in the case of (derived) rules, a justification.

So what exactly happens if one has a configuration $\langle sn\text{-}seq, S\rangle$ and wants to evaluate it? There should be (an instantiation of) a rule that has as its conclusion the transition $\langle sn\text{-}seq, S\rangle \hookrightarrow conf$ for some configuration *conf*. The hypotheses of such a rule consist of a (possibly empty) set of transitions and *PredS* (predicates on sequences of states). In the example of the configuration $\langle [OP_1], S\rangle$ (Example 4.2.9) one would get

$$conf = \langle [\,], add\text{-}to\text{-}SEStateOp(S, m)\rangle$$

with no hypotheses.

Before a rule can be used, its hypotheses would have to be discharged. For each *PredS* this is done by trying to prove it from the assumptions *collect-preds*$(S, [\,])$ (as defined in §4.2.1), working in the simplification theory $\widetilde{Th(\mathcal{L})}$ (cf. §5.3.2) which is a parent theory of the theory of operational semantics itself.

A transition is discharged by recursively running the same algorithm on this transition as on $\langle sn\text{-}seq, S\rangle \hookrightarrow conf$:

transform(*trans*) $\triangleq$

1. Try to find a rule *r* with (instantiated) conclusion *trans*
cases *Number of such rules* of
0 → answer NO and stop
1 → make the appropriate substitution and store it
≥ 2 → let user decide which one to use
end

2. for every *preds* in *hyps*(*r*)
do try to decide — is it the conclusion of a provable rule,
where all hypotheses are known to hold?
cases *result* of
true → nothing needs to be done
false → ignore the rule *r* and go back to 1.
otherwise keep it as a 'provided' condition on future results
end

3. for every *trans* in *hyps*(*r*)
do *transform*(*trans*)
4. If you get this far, the collected substitution applied to *trans* gives
the provable instantiation

This recursive algorithm will be expressed as a proof tactic (cf. §5.2.2), which can then be used to find a proof of the relevant transition. Alternatively, it could be expressed as a program procedure or an oracle (see §5.2.1).

If *conf* has the form $\langle [\,], S' \rangle$ then symbolic execution of *sn-seq* on *SEStateOp S* is finished with the resulting *SEStateOp S'*. If not, then *conf* must itself have the form $\langle sn\text{-}seq', S' \rangle$. In this case there should be (an instantiation of) a rule that has as its conclusion the transition $\langle sn\text{-}seq', S' \rangle \hookrightarrow conf'$ for some configuration *conf'*. Now one needs a rule that says: if the transitions $a \hookrightarrow b$ and $b \hookrightarrow c$ are (instances of) conclusions of rules, then one can derive a rule with conclusion $a \hookrightarrow c$, i.e. transitions are transitive. In the example, this leads to a rule with conclusion $\langle sn\text{-}seq, S \rangle \hookrightarrow conf'$, and the same cycle starts again. If *conf'* has the right form, symbolic execution is finished, otherwise there should be a transition starting with *conf'*. This cycle is repeated until one gets to a configuration of the form $\langle [\,], S'' \rangle$.

Effectively, in symbolic execution one tries to prove a theorem but, in contrast to the usual way of doing so, one does not know the conclusion of the theorem when starting to prove it. Instead, one knows that it should take the form $\langle sn\text{-}seq, s \rangle \hookrightarrow S'$, and that at any stage either there is only one rule that applies, or the user gets the choice which one to apply. This leads to Requirement 1 in §6.4, FRIPSE must allow proofs with unknown goals.

5.2 More on simplification and formal reasoning

It is important in this context to distinguish two different kinds of simplification, which get used for different purposes. First there is simplification by transformation into a suitable normal form, such as CNF, DNF or if-then-else NF. These normal forms are very useful if the simplified terms are to be handled mechanically by other tools, but in most cases they do not look simpler to a human user reading them. In some cases such normal forms can be found as normal forms with respect to an appropriate rewrite system, in other cases one needs a more general algorithm for transforming a given term into its normal form. A special case of such tool-oriented simplification is simplification using decision procedures for deciding the validity of a formula, i.e. deciding whether it can be simplified to true or false.

The second form of simplification is simplification with respect to a user: making an expression easier to understand for the user. Strictly speaking, it might be better to use the more general term 'transformation' in this case instead of 'simplification', since it depends very much on the individual user whether one term is simpler or more complicated than another. 'Simplification' used in this second meaning is a very subjective term, it always denotes simplification with respect to a particular user. 'Simplifying' a term so as to make it easier to understand for one user, might make it completely incomprehensible to another. This implies that this form of simplification has to be highly interactive and give the user a lot of choice about the information presentation, for example what simplification to apply.

Sometimes the two forms of simplification overlap, in particular when a term can be simplified to true or false, but in general the two are concerned with rather separate issues. Since symbolic execution as described in this thesis is concerned with making a specification easier to understand for a user, the following is only concerned with the second, user-oriented kind of simplification, and only considers the first one if the two overlap.

What support is needed for simplification in the context of IPSE 2.5? Simplification means replacing a term with a different but equivalent one (ignoring for now "weak" simplification as used in weak symbolic execution), and therefore needs a means of checking or proving the equivalence of terms. This needs to be an *interactive* process to allow and support user-guided simplification.

To support simplification in the context of symbolic execution, one therefore needs an interactive formal reasoning tool. For SYMBEX, this is provided by FRIPSE.

The simplification transformations used in SYMBEX (like those in FRIPSE) can take any one of the following forms:

- Applying a rewrite rule/inference rule. These are the main form of simplification transformations.
- Applying an 'oracle', such as a decision procedure or an algorithm for doing integer arithmetic. Oracles are used in a similar way to rules, but can be written in a general programming language, cf. §5.2.1.
- Applying a tactic. Tactics combine rules and oracles using suitable combinators, and thus enable the user to do larger simplification steps than would otherwise be possible. They are

described in more detail in §5.2.2.

The justification of a simplification transformation can come from one of several different sources:

- From the logic: a term may be simplified to another if the two terms are equivalent in the underlying logic.
- From a theory used, e.g. integers: a term may be simplified to another if the two terms are equivalent in the theory used.
- From a particular specification: a term may be simplified to another if the two terms are equivalent in the theory described by the specification. This includes for example folding and unfolding of functions defined in the specification.

The distinction between these three is mainly practical, rather than theoretical. A specification can be looked upon as defining a theory, although obviously a very specific one which is of little *general* use. Similarly, a logic can be considered as a theory that is mainly concerned with truth and falsehood of propositions.

Now assume one wants to show that a *PredS* ps_1 inside an *SEStateOp* S can be simplified to ps_2. Then the rule expressing that ps_1 and ps_2 are equivalent may contain as hypotheses any number of *PredS*s that occur in S up to the current stage that contains ps_1, since these *PredS*s are known to hold. The specification of *SIMPLIFY* in §6.1.2 will formalise this notion of 'up to the current stage'.

As an example, assume that at some stage in the symbolic execution of a specification one has *pushed* an element e onto the stack st and this is described by giving the description value

$$\bar{\sigma}([3], st) = push(\bar{\sigma}([2], st), e) \tag{5.1}$$

to the *Name* st. If in the next step one *pops* an element from st

$$\bar{\sigma}([4], st) = pop(\bar{\sigma}([3], st)) \tag{5.2}$$

then the *PredS* (5.1) is known to hold and can be used to simplify (5.2) to the equivalent

$$\bar{\sigma}([4], st) = \bar{\sigma}([2], st) \tag{5.3}$$

The discussion so far always assumed *full* symbolic execution, where simplification is an *equivalence* transformation, since every step has to preserve the denotation of the relevant configuration. Weak symbolic execution allows that (*SEQS* of) the denotation of S may increase. In weak symbolic execution one may therefore simplify ps_1 to ps_2 if $ps_1 \Rightarrow ps_2$. Equivalence is not required.

However, even in full symbolic execution one may sometimes seemingly relax this restriction. An expression, such as a description value, may contain more information than needed and wanted. For example, one of the conjuncts in a conjunction forming a description value of a variable x may seem irrelevant, perhaps because it does not contain x. Just deleting this information would in general lead to weak symbolic execution, but instead one may *hide* it, for example by replacing

it with "...". This way, the information is still stored in the state, but the user does not see it and cannot get distracted by the irrelevant *PredS*. The "..." act as a marker for the hidden information and allow the user to get to it.

If one wants to *delete* a *ps*: *PredS* rather than just hide it, then one has to show that it can be simplified to true, i.e. it is provable from the assumptions. Then *ps* is equivalent to true and can therefore be replaced by it.

Since the *PredS* true does not really put any restrictions on the set of states modelled by an *SEStateOp S*, it does not influence the denotational semantics of *S* (cf. definition of $\mathcal{M}_{SEStateOp}$ in Figure 4.3) and can therefore be deleted from any set of *PredS* *SEQ*(*S*)(*n*). This form of simplification extends the model of simplification described until now, since here a *PredS* is *deleted* rather than replaced by another, equivalent *PredS*.

One case when this can be particularly useful is the deletion of duplicate *PredS*s, introduced for example by the rule for VDM-operations in Figure 4.4, where *m*(*n*) contains the same *PredS* provided... for different identifiers *n*. If the user wants, all but one of these *PredS*s can be 'simplified away', since they are provable from the one that remains.

One has to be careful about simplifying too much, since this might lose valuable information. For example, if two operations cancel each other out, such as push and pop, the user should at least be shown this fact explicitly. A similar case is an `if-then-else`-statement, where from the path condition it may follow that only one branch can be taken, the other one is never taken. This is a 'surprising' property that the user should be told of, since it might point to an error in the specification. The problem in both cases is that the simplification preserves the denotation of the state, but the information about the *computation* itself is not captured by the denotation and therefore in general not preserved by simplification. In many cases, of course, the whole point of simplification is to remove distracting information about the computation that is not (or no longer) needed. To counter this problem, when a displayed expression has been simplified then at first both the un-simplified and the simplified version should be displayed (but allowing the user to turn display of the un-simplified version off). After that, only the simplified version need be considered (i.e. this version is automatically **remembered** — of course this only applies to those simplifications designated as "automatic" by the user).

A very basic simplification tactic should be applied automatically after each symbolic execution step. This tactic should simplify arithmetic expressions, replace variables by their actual value if possible, etc. The user interface of simplification is discussed in §6.2.5.

5.2.1 Oracles

Oracles can be considered as a generalisation of (semi-) decision procedures. In its simplest form, an oracle takes a formula as an argument and answers YES if it can prove it; i.e. if the answer is YES, then the formula is provable. This can be extended to oracles which transform an input term into an equivalent output term which is in some sense 'better', e.g. because it is simpler or in some normal form. An example of this is the oracle that takes (ground) terms of integer arithmetic and evaluates them.

Oracles complement tactics in that they describe transformations as well, but can be written in

a general programming language. This greater expressiveness introduces the danger of introducing oracles that are not correct. For every oracle one therefore has to prove that input and output are equivalent in the relevant theory, similarly as for rules.

The reason for the introduction of oracles is that a number of important transformation and simplification algorithms can not, or at least not reasonably efficiently, be implemented using only proof rules. Examples where this is the case are integer arithmetic, and transformation of formulae into prenex normal form. In particular the first one will probably be needed a lot in theorem proving, but evaluation of arithmetic ground terms by application of proof rules, say the rules of Peano arithmetic, would be extremely slow.

5.2.2 Tactics

Tactics are used to relieve the user from some of the tedium of fully formal theorem proving, and to enable her to do larger proof steps in textbook fashion rather than explicitly apply inference rules one at a time. They are built up from inference rules and oracles, using combinators.

A typical simplification tactic for use in symbolic execution would for example unfold certain function definitions, replace variables by their actual value if they have one, evaluate arithmetic ground terms, and eliminate the constants true and false from logical expressions whenever possible.

Combinators do have read access to the state of the proof, but they do not have write access, they never change the state of the proof directly. Instead, they guide the application of rules and oracles. This guarantees that, assuming that all rules and oracles are correct, the result of applying a tactic also is correct, since it has been achieved by only applying correct primitives. Tactics could therefore be considered as functions on the abstract data type of provable rules and oracles. Any application of a tactic can only change the abstract data type by using the basic functions explicitly provided, i.e. rules and oracles.

Tactics are more expressive than derived inference rules, since

- they allow iteration. Although it is conceivable that the language for expressing derived rules could by extended by some form of $\cdots$-operator to handle iteration, this will not be done in either SYMBEX or IPSE 2.5.
- they can fail. Combinators can make use of this to achieve backtracking (e.g. using `orelse`). For example, the tactic t_1 `orelse` t_2 first runs the tactic t_1 which might try to apply a certain proof rule. If this fails, say because no substitution is found that makes the rule applicable, then t_2 is used instead, which might try to apply another proof rule.
- tactics allow combination of rules and *oracles* (and other tactics).

However, the first two reasons only apply when comparing *individual* rules with tactics. The effect of a tactic that is built up using only rules and no oracles is exactly the same as that of a (possibly infinite) *set* of derived rules.

A number of important tactics will be provided as part of SYMBEX and/or FRIPSE, plus a tactic language that allows the user to write her own tactics. The latter is particularly important to allow the user to tailor simplification to her particular needs.

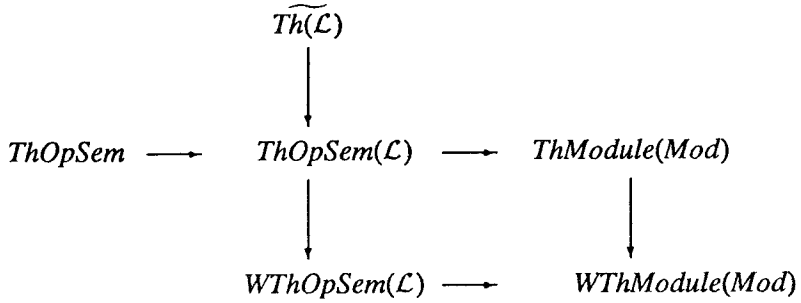
5.3 The theories of operational semantics of symbolic execution

The theory of the operational semantics of a particular language $\mathcal{L}$ consists of a theory *ThOpSem* common to all such theories, and a language-specific part *ThOpSem*($\mathcal{L}$). Simplification is based on a theory *Th*($\mathcal{L}$) which includes the logic used for describing $\mathcal{L}$ (for VDM, this would be LPF), plus the theories of its basic data types. Additionally, every specification module has a theory *ThModule*(*Mod*) of its own containing, among other things, the type and function definitions of the module.

So far, only 'full' theories were mentioned, i.e. theories that describe the operational semantics of full symbolic execution (cf. Figure 4.2). Additionally the 'weak' theories *WThOpSem*($\mathcal{L}$) and *WThModule*(*Mod*) are needed that describe weak symbolic execution where some of the restrictions on the result state are 'lost', leading to non-faithful transitions in the operational semantics.

These theories are all based on the same logic, a common 'logic of operational semantics', rather than having a collection of different language-dependent logics. Here one has to distinguish between the logic of a specification language, which is included in *Th*($\mathcal{L}$) and used for reasoning about terms of the language, and the logic used for reasoning within the various theories about transitions etc., which is independent of the language used. LPF was chosen as this common logic.

Before describing these theories in more detail, the following diagram shows the *parent* → *child* relationships existing between them.



Several of the definitions in this section were already introduced earlier. The new definitions are given because they now define objects in the various theories of operational semantics, used by SYMBEX, while before they defined general data structures.

5.3.1 The common theory *ThOpSem*

Sorts

The theory *ThOpSem* should have the sort symbols *name*, *preds* and *SEStateOp*. Here one has to distinguish between the *primitive* sort symbols *name* and *preds* as introduced in *ThOpSem*,

and the defined sorts *Name* and *PredS* which are language-dependent and therefore introduced in *ThOpSem*($\mathcal{L}$). *ThOpSem*($\mathcal{L}$) will inherit *ThOpSem* via a morphism translating *name* to *Name* and *preds* to *PredS*.

ThOpSem also has the sort constructors seq of A for sequences, with type parameter A, and map A to B for maps and $A \mid B$ for type union, both with type parameters A and B. All these sort constructors have their appropriate theory associated, either as part of *ThOpSem* itself, or, more likely, as a parent theory. Also needed is the sort *Prop* of propositions. Among other things, this includes equations and transitions.

Transitions

ThOpSem has the primitive (polymorphic) constant

$$\hookrightarrow : A \times A \to Prop$$

which denotes transitions, and the constants *from* and *to* denoting the inverses of $\hookrightarrow$. Since $\hookrightarrow$ is polymorphic, *from* and *to* cannot be declared as having a certain type.
Transitions are reflexive and transitive:

$$\vdash E \hookrightarrow E$$

and

$$\frac{E_1 \hookrightarrow E_2 \qquad E_2 \hookrightarrow E_3}{E_1 \hookrightarrow E_3}$$

and *from*, *to* are the inverses of $\hookrightarrow$:

$$\vdash from(E_1 \hookrightarrow E_2) = E_1$$

$$\vdash to(E_1 \hookrightarrow E_2) = E_2$$

$$\frac{from(t) : A}{to(t) : A}$$

SEStateOps

Define

$$Index = \text{seq of } \mathbb{N}_1$$

A state as used for describing the operational semantics of a language for symbolic execution is defined recursively as

$$SE\text{-}map = \text{map } name \text{ to set of } preds$$

$$SE\text{-}elem = SE\text{-}map \mid SEStateOp$$

$$\begin{aligned} SEStateOp :: \quad & SEQ : \text{seq of } SE\text{-}elem \\ & INDEX : Index \end{aligned}$$

where

$$\begin{aligned}&\textit{inv-SEStateOp}(\textit{mk-SEStateOp}(\textit{Seq}, \textit{ix})) \triangleq \\ &\quad \textit{Seq} \neq [\,] \\ &\quad \wedge\ \mathsf{hd}\,\textit{Seq}: \textit{SE-map} \\ &\quad \wedge\ \forall k \leq \mathsf{len}\,\textit{Seq} \cdot \textit{Seq}[k]: \textit{SEStateOp} \Rightarrow \textit{INDEX}(\textit{Seq}[k]) = \mathsf{cons}(k, \textit{ix})\end{aligned}$$

5.3.2 The simplification theories $Th(\mathcal{L})$ and $\widetilde{Th(\mathcal{L})}$

Now assume that a fixed specification language $\mathcal{L}$ is given. Let $Th(\mathcal{L})$ be the theory used to reason about terms in $\mathcal{L}$. For example, $Th(\text{VDM}) = \text{LPF}$. $Th(\mathcal{L})$ should not be considered as denoting one *parameterised* theory, but rather a collection of different theories, one for each language $\mathcal{L}$.

$Th(\mathcal{L})$ is based on the logic of $\mathcal{L}$, with type $Prop_{Simp}$ of propositions, and additionally contains the theories of the basic data types of $\mathcal{L}$ such as sets, sequences, etc. It is thus the theory needed for reasoning about $\mathcal{L}$ in general, independent of symbolic execution.

Let *Name* be the type of identifiers or program variables of $\mathcal{L}$, as before. For a typed language $\mathcal{L}$, this would actually have to be a collection of types, but for simplicity this fact is ignored here. Let *Val* be the type of values that an identifier may take. Again, for a typed language this would have to be a collection of types.

We require that $Prop_{Simp}$ includes the constants true and false, and operators $\wedge$, provided-then, $\Rightarrow$ and $\Leftrightarrow$. Also needed are the functions *start-block* and *finish-block*, as defined in §4.2.5. These are needed in order to be able to describe the operational semantics of blocks (cf. §4.2.5) or loops (cf. §4.2.8). Furthermore, the language should be 'reasonably expressive' in the sense that the transitions of the operational semantics of $\mathcal{L}$, as will be discussed in §5.3.3, can be expressed.

One needs to introduce an indexing mechanism to differentiate between the values of program variables (identifiers or names) at different stages in an execution sequence. To do so, we introduce sequences $(\sigma_i)_i$ of states, where $\sigma_i: \Sigma_\perp$. Since the definition of *SEStateOp* is recursive, simple sequences are not enough — we actually need iterated sequences where σ_i might be a sequence of states itself. This is modelled by introducing a constant symbol $\bar{\sigma}$ with arity (2,0), which is the name of the value of the identifier n at a given stage in the execution, with the only axiom

$$\frac{\textit{i-seq}: \mathsf{seq\ of}\ \mathbb{N}_1 \qquad \textit{i-seq} \neq [\,] \qquad n: \textit{Name}}{\bar{\sigma}(\textit{i-seq}, n): \textit{Val-ref}}$$

For simplicity, the element $i: \mathbb{N}$ will in the following sometimes be identified with the sequence $\textit{i-seq} = [i]$.

Now define a *PredS* as a proposition of $Th(\mathcal{L})$ where each *Name* n has been replaced by $\bar{\sigma}(\textit{i-seq}, n)$ for some *i-seq*.

$$\frac{\Phi: Prop_{Spec} \qquad \textit{i-seq}: \mathsf{seq\ of}\ \mathbb{N}_1}{\Phi[n/\bar{\sigma}(\textit{i-seq}, n) \mid n: \textit{Name}]: \textit{PredS}}$$

These are the predicates the user actually gets to see as description values of variables at any stage in the symbolic execution. The resulting new theory with *PredS* instead of $Prop_{Simp}$ will be called $\widetilde{Th(\mathcal{L})}$. Effectively, this new theory then contains multiple copies of $Prop_{Spec}$, one for each value

of *i-seq*. Note that, in the example of LPF, if n gets replaced by $\bar{\sigma}(i,n)$ then $\overleftarrow{n}$ gets replaced by $\bar{\sigma}(i-1,n)$.

Also needed is the following auxiliary function:

mentions: *PredS* → set of *Name*

which collects the identifiers n mentioned in a given ps: *PredS*, i.e. those n for which ps contains $\bar{\sigma}(i,n)$ for any i. This function has to be defined recursively over the syntax of *PredS* (or $Prop_{Simp}$).

The theory $\widetilde{Th(\mathcal{L})}$ is used for simplification: ps_1: *PredS* inside some *SEStateOp* can be simplified to (i.e. replaced by) ps_2: *PredS* if they are equivalent in $\widetilde{Th(\mathcal{L})}$, given that all *PredS* that occur in the *SEStateOp* at an earlier stage hold. Note that, in symbolic execution, one is not directly interested in the theorems of $\widetilde{Th(\mathcal{L})}$ as such, but indirectly in them providing a justification of those theorems of *ThOpSem*($\mathcal{L}$) which describe simplification steps.

5.3.3 The language-dependent theories *ThOpSem*($\mathcal{L}$)

Again assume one is given some fixed language $\mathcal{L}$ and wants to describe the theory *ThOpSem*($\mathcal{L}$) of its operational semantics. As before, *ThOpSem*($\mathcal{L}$) does not denote one *parameterised* theory, but a collection of different theories. *ThOpSem*($\mathcal{L}$) is the theory that describes the operational semantics of symbolic execution of a particular language $\mathcal{L}$. It inherits as parent theories the theory *ThOpSem* of operational semantics in general (via a morphism translating *name* to *Name* and *preds* to *PredS*), independent of the language $\mathcal{L}$, and $\widetilde{Th(\mathcal{L})}$, the simplification theory of $\mathcal{L}$.

Let *Spec* be the type of all specifications and programs, i.e. all terms in $\mathcal{L}$ denoting a binary relation on states. *SpecName* is the type of specification names, and *SpecMap* associates specification names with specifications:

SpecMap = map *SpecName* to *Spec*

Configurations consist of a sequence of *SpecName*s and an *SEStateOp*:

Conf :: *SNSEQ* : seq of *SpecName*
STATE : *SEStateOp*

One can now introduce the transitions and transition rules describing the language $\mathcal{L}$ as axioms or rules of *ThOpSem*($\mathcal{L}$), as described in §4.2, and derive the wanted transitions from them.

5.3.4 The theories *ThModule*(*Mod*) of specification modules

A specification module *Mod*[1] consists of

- type definitions
- function definitions
- definitions of specifications. Here specifications are terms denoting a relation on states. In VDM, these would be called operations.

[1]Note that a specification module is similar to, but not the same as, a *Module* in the BSI-Protostandard for VDM [And88, BSI88].

The theory *ThModule(Mod)* of a specification module then inherits the theory *ThOpSem*($\mathcal{L}$) of the language used and additionally contains

- symbols for all the defined types of the module, plus their definitions
- symbols for all the functions of the module, plus their definitions (axiomatic or otherwise)
- a constant *specmap*: *SpecMap*.
- transitions found by symbolic execution, expressed as rules.

However, in *ThModule(Mod)* all these definition have to be expressed in the language of FRIPSE rather than the specification language, since they are to be part of a FRIPSE theory. Since they are originally expressed in the specification language (and kept on the LHS, cf. §1.4), they will have to be translated first.

The constant *specmap*

First we have to define the following auxiliary functions. *is-prim-constant-of* checks whether a given constant symbol is declared in the theory referenced by a given *thr*: *Theory-ref*. *specs* takes a reference to a theory (it will be applied to theories of specification modules) and returns the names of specifications in the domain of the constant *specmap*, i.e. the names of specifications in the module. From now on, *specmap* will be a *fixed* constant symbol of type *CESymb*.

is-prim-constant-of (*c*: *CESymb*, *thr*: *Theory-ref*) *r*: $\mathbb{B}$
ext rd *fripse* : *Store*
pre *thr* $\in$ dom *THS*(*fripse*)
post *r* $\Leftrightarrow$ *c* $\in$ dom *PCE*(*EXSIG*(*THS*(*fripse*)(*thr*)))

Alternatively, one could introduce *specmap* as a *defined* constant rather than a primitive constant. I have not specified this alternative here since in a future version of the specification of FRIPSE, the two concepts will be merged anyway.

specs (*thr*: *Theory-ref*) *r*: set of *SpecName*
ext rd *fripse* : *Store*
pre *thr* $\in$ dom *THS*(*fripse*)
$\wedge$ *is-prim-constant-of*(*specmap*, *thr*)
$\wedge \vdash_{THS(fripse)(thr)}$ *specmap*: *SpecMap*
post $\forall sn$: *SpecName* $\cdot$ *sn* $\in$ *r* $\Leftrightarrow \vdash_{THS(fripse)(thr)}$ *sn* $\in$ dom *specmap*

We require for any theory *ThModule(Mod)* that the axioms provided about *specmap* ensure that *specs* is implementable.

5.3.5 The weak theories *WThOpSem*($\mathcal{L}$) and *WThModule*(*Mod*)

The theory *WThOpSem*($\mathcal{L}$) includes those rules of the operational semantics of $\mathcal{L}$ which are not faithful (and therefore do not describe *full* symbolic execution) but which do describe *weak* symbolic execution. It has *ThOpSem*($\mathcal{L}$) as a parent theory. The *PredS*-information contained in an *SEStateOp* after *weak* symbolic execution is also correct under *full* symbolic execution, but it may be incomplete and not fully describe the results of actual execution (cf. §4.1.1).

The theory *WThModule*(*Mod*) merges information about the module *Mod* and information about weak symbolic execution. Therefore it is defined as a theory that does not contain any constants or axioms itself, but only the two parent theories *ThModule*(*Mod*) and *WThOpSem*($\mathcal{L}$).

One can see that from the point of view of the theories involved, *weak* symbolic execution is not essentially different from *full* symbolic execution. The rules used take the same form, and they will be applied in the same way. The essential difference between the two is that the rules for weak symbolic execution convey less information in the sense that the sets of *PredS* one gets as values for the different *Names* may be smaller, and the individual *PredS* may only be consequences of rather than equivalent to those one gets from full symbolic execution. However, this does not affect the structure of these rules, indeed a rule describing weak symbolic execution of one operation may at the same time describe full symbolic execution of some other operation: let *SORT*1 be an operation that sorts a list of *Persons* by their age. If several *Persons* have the same age then they may be put in some arbitrary order. Alternatively, *SORT*2 requires that in this case they should be ordered alphabetically on their names. Then in *weak* symbolic execution of *SORT*2 the additional requirement might be dropped and thus lead to the same result as *full* symbolic execution of *SORT*1.

Chapter 6

The symbolic execution system SYMBEX

'Alright,' said Deep Thought. 'The Answer to the Great Question ...'
'Yes ...!'
'Of Life, the Universe and Everything ...' said Deep Thought.
'Yes ...!'
'Is ...' said Deep Thought, and paused.
'Yes ...!'
'Is ...'
'Yes ...!!! ...?'
'Forty two,' said Deep Thought, with infinite majesty and calm.

Douglas Adams: The Hitchhiker's Guide to the Galaxy

6.1 Specification of SYMBEX

This specification makes heavy use of other work within IPSE 2.5, in particular the specification of FRIPSE [LM88]. A short summary of the data structures and functions from [LM88] used in this thesis is given in Appendix C.

6.1.1 Data structure and some auxiliary functions

SEStateOp

Define

$Index = \text{seq of } \mathbb{N}_1$

A state as used for describing the operational semantics of a language for symbolic execution is defined recursively by

$$SE\text{-}map = \text{map } Name \text{ to set of } PredS$$

$$SE\text{-}elem = SE\text{-}map \mid SEStateOp$$

$$\begin{aligned} SEStateOp :: \quad & SEQ : \text{seq of } SE\text{-}elem \\ & INDEX : Index \end{aligned}$$

where

$$\begin{aligned} & inv\text{-}SEStateOp(mk\text{-}SEStateOp(Seq, ix)) \triangleq \\ & \quad Seq \neq [\,] \\ & \quad \wedge\ \text{hd } Seq : SE\text{-}map \\ & \quad \wedge\ \forall k \leq \text{len } Seq \cdot Seq[k] : SEStateOp \Rightarrow INDEX(Seq[k]) = \text{cons}(k, ix) \end{aligned}$$

This is the same definition as in the definition of the operational semantics of symbolic execution in §4.2.1, repeated in the theory *ThOpSem* (described in §5.3.1). However, it is now considered as a part of the *SYMBEXSTATE*, while before it was a type defined within the operational semantics (§4.2.1) or a FRIPSE-theory (§5.3.1). Similarly, some of the functions defined below have been defined before in §4.2 or §5.3. In the implementation of SYMBEX, a translation mechanism will be needed that translates between these different versions, in particular between an *SEStateOp* in the *SYMBEXSTATE* and the equivalent one in *ThOpSem* (cf. §6.3). This will be necessary so that symbolic execution on the *SEStateOp* in *SYMBEXSTATE* can be performed according to the rules of the theory *ThOpSem*.

Auxiliary functions

The function *get-element* gets a particular element of the sequence in an *SEStateOp* or one of its sub-sequences, as selected by its argument *ix*:

$$get\text{-}element : SEStateOp \times Index \to SE\text{-}elem$$

$$\begin{aligned} & get\text{-}element(S, ix) \triangleq \text{if front } ix = [\,] \\ & \qquad \text{then } SEQ(S)[\text{last } ix] \\ & \qquad \text{else } get\text{-}element(SEQ(S)[\text{last } ix], \text{front } ix) \end{aligned}$$

$$\begin{aligned} & \text{pre } ix \neq [\,] \\ & \quad \wedge\ \text{if front } ix \neq [\,] \\ & \quad\ \text{then } SEQ(S)[\text{last } ix] : SEStateOp \\ & \qquad \wedge\ pre\text{-}get\text{-}element(SEQ(S)[\text{last } ix], \text{front } ix) \\ & \quad\ \text{else true} \end{aligned}$$

The function *current-index* finds the current or last index in an *SEStateOp*:

$$\textit{current-index} : \textit{SEStateOp} \to \textit{Index}$$

$$\begin{aligned}\textit{current-index}(S) \triangleq\ & \text{if last}\,\textit{SEQ}(S)\text{: }\textit{SE-map}\\ & \text{then [len}\,\textit{SEQ}(S)\text{]}\\ & \text{else }\textit{current-index}(\text{last}\,\textit{SEQ}(S)) \frown \text{len}\,\textit{SEQ}(S)\end{aligned}$$

current-index(*S*) is always the index of a *SE-map*:

Lemma 6.1.1

$$\begin{aligned}&\forall S\text{: }\textit{SEStateOp} \cdot\\ &\quad \textit{pre-get-element}(S, \textit{current-index}(S))\\ &\quad \wedge\ \textit{get-element}(S, \textit{current-index}(S))\text{: }\textit{SE-map}\end{aligned}$$

Proof See Appendix A.4. □

The function *previous*, given the index of an element in *SEStateOp*, finds the index of the previous element:

$$\textit{previous} : \textit{Index} \to \textit{Index}$$

$$\begin{aligned}\textit{previous}(ix) \triangleq\ & \text{if hd}\,ix = 1\\ & \text{then tl}\,ix\\ & \text{else cons(hd}\,ix - 1, \text{tl}\,ix)\end{aligned}$$

$$\text{pre } ix \neq [\,]$$

We now introduce the function *collect-preds*, which collects into a set all the *PredS* in a given *SEStateOp S*, up to a certain element (given as argument *ix*) in the execution sequence of *S*. If *ix* is empty, then *all PredS* in *S* are collected:

$$\textit{collect-preds} : \textit{SEStateOp} \times \textit{Index} \to \text{set of } \textit{PredS}$$

$$\begin{aligned}&\textit{collect-preds}(S, ix) \triangleq\\ &\quad \text{let } ix' = \text{if } ix = [\,] \text{ then [len}\,\textit{SEQ}(S)\text{] else } ix \text{ in}\\ &\quad \bigcup_{i=1}^{\text{last } ix'} \Big(\text{if } \textit{SEQ}(S)[i]\text{: }\textit{SE-map}\\ &\qquad\quad \text{then } \bigcup_{n \in \text{dom } \textit{SEQ}(S)[i]} \textit{SEQ}(S)[i](n)\\ &\qquad\quad \text{else if } i = \text{last } ix'\\ &\qquad\qquad \text{then } \textit{collect-preds}(\textit{SEQ}(S)[i], \text{front } ix')\\ &\qquad\qquad \text{else } \textit{collect-preds}(\textit{SEQ}(S)[i], [\,])\Big)\end{aligned}$$

$$\begin{aligned}&\text{pre if } ix \neq [\,] \text{ then}\\ &\quad \text{last}\,ix \leq \text{len}\,\textit{SEQ}(S)\\ &\qquad \wedge\ \text{if } \textit{SEQ}(S)[\text{last}\,ix]\text{: }\textit{SEStateOp}\\ &\qquad\quad \text{then } \textit{pre-collect-preds}(\textit{SEQ}(S)[\text{last}\,ix], \text{front}\,ix)\\ &\qquad\quad \text{else front}\,ix = [\,]\\ &\quad \text{else true}\end{aligned}$$

Assumptions and Beliefs

$$\begin{aligned} Assump :: \; & index : Index \\ & stmt : PredS \end{aligned}$$

Assumptions are used for recording assumed predicates. They consist of an index which records *when* an assumption was made, and the assumed statement itself. The following function extracts the statements from a set of *Assump*.

$$statements : \text{set of } Assump \to \text{set of } PredS$$

$$statements(as) \triangleq \{stmt(a) \mid a \in as\}$$

A *Belief*, which is used to store a believed predicate, is similar to an *Assump*, except that it also stores the description values in the current element of the *SEStateOp*. This is necessary since a *Belief* represents a proof obligation that should later be discharged. To do so, one needs to know the hypotheses that are allowed to be used in the proof, namely all the *PredS*s that are known to hold at the time when the *Belief* is stated (cf. the specifications of the operations *BELIEVE* and *DISCHARGE* in §6.1.2).

$$\begin{aligned} Belief :: \; & index : Index \\ & current : \text{set of } PredS \\ & stmt : PredS \end{aligned}$$

The function name *statements* is now overloaded to extract the statements from *Belief*s as well as *Assump*s:

$$statements : \text{set of } Belief \to \text{set of } PredS$$

$$statements(bs) \triangleq \{stmt(b) \mid b \in bs\}$$

Proven and provable rule statements

The following function checks whether a given *RuleStmt* is established by a given rule under a given instantiation, using various functions given in Appendix C:

$$\begin{aligned} & isProvenRuleStmt : RuleStmt \times Rule\text{-}ref \times Theory\text{-}ref \times Instantiation \\ & \qquad \times Rulemap \times Theorymap \times ThMorphmap \to \mathbb{B} \end{aligned}$$

$$\begin{aligned} & isProvenRuleStmt(rs, rr, thr, i, rm, thm, thmm) \triangleq \\ & \quad \text{let } rule = rm(rr) \text{ in} \\ & \quad \text{let } rs' = mk\text{-}RuleStmt(\\ & \qquad \{Instantiate(s, i) \mid s \in SEQHYPS(STMT(rule))\}, \\ & \qquad \{Instantiate(a, i) \mid a \in ORDHYPS(STMT(rule))\}, \\ & \qquad Instantiate(CONCL(STMT(rule)), i)) \text{ in} \\ & \quad Establishes(rs', rs) \\ & \quad \wedge Is\text{-}Complete\text{-}Proof(PROOF(rule), thr, rm, thm, thmm) \end{aligned}$$

The operation *PROVABLE* checks whether a rule statement is provable in a theory and, if it is, adds it (including its proof) to the theory as a new rule. This operation is to be provided by FRIPSE.

PROVABLE (*rs*: *RuleStmt*, *th*: *Theory-ref*) *r*: {YES, DONTKNOW}
ext wr *fripse* : *Store*
post *r* = YES ⇒ *rs* is provable in *th*
∧ the rule with statement *rs* and a (complete) proof is
added to *th* in *fripse*

Of course there exists a trivial implementation of *PROVABLE* that always returns DONTKNOW. Although this implementation would be correct with respect to the specification, obviously one would hope for something more intelligent, probably implemented by proof tactics and/or using decision procedures for decidable classes of problems. In different contexts, one should presumably use different proof tactics and decision procedures, even though they implement the same operation *PROVABLE*. An example of such a proof tactic is the algorithm *transform* given in §5.1.

The state

The state of a symbolic execution system has the following structure:

SYMBEXSTATE :: *S* : *SEStateOp*
history : seq of *SpecName*
assume : set of *Assump*
beliefs : set of *Belief*
module : *Theory-ref*
wmodule : *Theory-ref*
wflag : $\mathbb{B}$
fripse : *Store*

where

inv-SYMBEXSTATE(*mk-SYMBEXSTATE*(*s*, *h*, *ass*, *b*, *m*, *wm*, *wf*, *f*)) $\triangleq$
$m \in \text{dom } THS(f)$
$\wedge\ wm \in \text{dom } THS(f)$
$\wedge$ *inv-ThModule*(*THS*(*f*)(*m*))
$\wedge$ *inv-WThModule*(*THS*(*f*)(*wm*))
$\wedge\ m \in PARENTS(THS(f)(wm))$
$\wedge\ \text{len } SEQ(s) = \text{len } h + 1$
$\wedge\ \text{rng } h \subseteq specs(m)$
$\wedge\ h = [\,] \Rightarrow wf = \text{false}$
$\wedge\ INDEX(s) = [\,]$

The invariant expresses that the theories *module* and *wmodule* should be the names (in *fripse*) of the theory and weak theory of the same module. The length of *SEQ*(*S*) should be one more than the length of the *history* to allow for the initial starting state. All the *SpecNames* in the *history* should be defined in the *module*. The *wflag* should initially be set to false to show that so far no weak symbolic execution has taken place. The *SEStateOp* should not be an element inside some other *SEStateOp*.

Copying the state

The following function copies an existing *SYMBEXSTATE*, up to a given element in the execution sequence.

$$\begin{aligned}
&\mathit{copy\text{-}SYMBEXSTATE} : \mathit{SYMBEXSTATE} \times \mathbb{N}_1 \to \mathit{SYMBEXSTATE} \\
&\mathit{copy\text{-}SYMBEXSTATE}(\mathit{mk\text{-}SYMBEXSTATE}(s, h, ass, bel, m, wm, wf, f), i) \triangleq \\
&\quad \mathit{mk\text{-}SYMBEXSTATE}(\\
&\qquad \mathit{mk\text{-}SEStateOp}(\mathit{SEQ}(s)(1, \ldots, i), \mathit{INDEX}(s)), \\
&\qquad h(1, \ldots, i-1), \\
&\qquad \{a{:}\,\mathit{Assump} \mid a \in ass \wedge \mathsf{last}\,\mathit{index}(a) \leq i\}, \\
&\qquad \{b{:}\,\mathit{Belief} \mid b \in bel \wedge \mathsf{last}\,\mathit{index}(b) \leq i\}, \\
&\qquad m, \\
&\qquad wm, \\
&\qquad wf, \\
&\qquad f)
\end{aligned}$$

Initial states

A *SYMBEXSTATE* is initial, if it satisfies

$$\begin{aligned}
&\mathit{is\text{-}initial\text{-}SYMBEXSTATE} : \mathit{SYMBEXSTATE} \to \mathbb{B} \\
&\mathit{is\text{-}initial\text{-}SYMBEXSTATE}(\mathit{mk\text{-}SYMBEXSTATE}(s, h, ass, b, m, wm, wf, f)) \triangleq \\
&\quad \mathsf{len}\,\mathit{SEQ}(s) = 1 \\
&\quad \wedge\, \mathsf{dom}\,\mathsf{hd}\,\mathit{SEQ}(s) = \{\,\} \\
&\quad \wedge\, ass = \{\,\} \\
&\quad \wedge\, b = \{\,\}
\end{aligned}$$

The invariant on *SYMBEXSTATE* implies, for an initial *SYMBEXSTATE*, that *history* = [ ] and *wflag* = false. Different initial *SYMBEXSTATE*s at most differ in their *module*, *wmodule* and *fripse*. Given any particular *module*, *wmodule* and *fripse*, the initial *SYMBEXSTATE* will be called ARBITRARY.

6.1.2 Operations

All the operations specified in the following should be accessible to the user. Their user interface is discussed in §6.2.5.

Symbolic execution

Operation *SYMB_EXECUTE* symbolically executes a sequence of specifications. The pre-condition checks that there is a rule or axiom that holds in *module* and has the shape

$$\mathit{hyp\text{-}set} \vdash \langle \mathit{sn\text{-}seq}, S \rangle \hookrightarrow \langle [\,], S' \rangle$$

for some $\mathit{hyp\text{-}set} \subseteq \mathit{collect\text{-}preds}(S, [\,])$ and some S': *SEStateOp*. The post-condition then applies this transition to S.

SYMB_EXECUTE (*sn-seq*: seq of *SpecName*)
ext wr S : *SEStateOp*
wr *history* : seq of *SpecName*
rd *module* : *Theory-ref*
rd *wflag* : $\mathbb{B}$
rd *fripse* : *Store*
pre *wflag* = false
$\wedge$ rng *sn-seq* $\subseteq$ *specs*(*module*)
$\wedge \exists S'$: *SEStateOp* $\cdot \exists$ *hyp-set* $\subseteq$ *collect-preds*$(S, [\,])$ $\cdot$
let *rs* = *mk-RuleStmt*({ },
hyp-set,
$\langle \mathit{sn\text{-}seq}, S \rangle \hookrightarrow \langle [\,], S' \rangle$) in
PROVABLE(*rs*, *module*) = YES
$\wedge$ len *SEQ*(S') = len *SEQ*(S) + len *sn-seq*
post $\exists$ *hyp-set* $\subseteq$ *collect-preds*$(S, [\,])$ $\cdot$
let *rs* = *mk-RuleStmt*({ },
hyp-set,
$\langle \mathit{sn\text{-}seq}, \overleftarrow{S} \rangle \hookrightarrow \langle [\,], S \rangle$) in
PROVABLE(*rs*, *module*) = YES
$\wedge$ *history* = $\overleftarrow{\mathit{history}} \frown$ *sn-seq*

In the special case of *SYMB_EXECUTE*, one should use the "tactic" *transform* introduced in §5.1 to implement the operation *PROVABLE* and find a proof of the *RuleStmt*.

The theory of the operational semantics of a language is expected to be such that at any stage in the symbolic execution, usually (but not necessarily) only one rule will be applicable. In this case the tactic is fully automatic, no user interaction is required. Note however that this remark *only* applies to symbolic execution itself, simplification is a separate step and should certainly be user-guided. Although the operation *SYMB_EXECUTE* allows the user to symbolically execute a whole *sequence* of specifications, she will often only want to execute one at a time and then execute the next specification on the result of the previous symbolic execution.

The reason for the pre-condition *wflag* = false is that once weak symbolic execution has been used on an *SEStateOp*, any further symbolic execution can only lead to a weak result and therefore has to be dealt with using the operation *W_SYMB_EXECUTE* specified below.

Weak symbolic execution

W_SYMB_EXECUTE behaves just like *SYMB_EXECUTE*, except that it uses the weak theory *wmodule* instead of *module*, and sets the *wflag* to show that the result has been derived using *weak* symbolic execution.

W_SYMB_EXECUTE (*sn-seq*: seq of *SpecName*)
ext wr S : *SEStateOp*
wr *history* : seq of *SpecName*
rd *wmodule* : *Theory-ref*
wr *wflag* : $\mathbb{B}$
rd *fripse* : *Store*
pre rng *sn-seq* $\subseteq$ *specs*(*wmodule*)
$\wedge \exists S'$: *SEStateOp* $\cdot \exists$ *hyp-set* $\subseteq$ *collect-preds*(S, [ ]) $\cdot$
let *rs* = *mk-RuleStmt*({ },
hyp-set,
⟨*sn-seq*, S⟩ $\hookrightarrow$ ⟨[ ], S'⟩) in
PROVABLE(*rs*, *wmodule*) = YES
$\wedge$ len *SEQ*(S') = len *SEQ*(S) + len *sn-seq*
post *post-SYMB_EXECUTE*(*sn-seq*, $\overleftarrow{S}$, $\overleftarrow{history}$,
wmodule, *fripse*, $\overleftarrow{wflag}$, S, *history*)
$\wedge$ *wflag* = true

Showing the results

SHOW shows the value of a program variable *name* after execution of a number of operations given by the index *ix*.

SHOW (*name*: *Name*, *ix*: *Index*) *ps-set*: set of *PredS*
ext rd S : *SEStateOp*
pre *pre-get-element*(S, *ix*)
$\wedge$ *get-element*(S, *ix*): *SE-map*
$\wedge$ *name* $\in$ dom *get-element*(S, *ix*)
post *ps-set* = *get-element*(S, *ix*)(*name*)

To see the value of a program variable in terms of the values after n operations (with $n < m$), run *SHOW* and then *SIMPLIFY* the result.

Simplification

SIMPLIFY simplifies an expression by applying a rule to it (but only *displays* the result and does *not* change the state).

To specify *SIMPLIFY*, we need the auxiliary function *simp-hypotheses*. This function collects all the *PredS* in an *SEStateOp S* up to an index *ix*, except for a given *ps* which is a current description value of the given name *n*. This is exactly the set of hypotheses allowed to be used for proving a simplification of *ps*. Note that this definition does not exclude the possibility that *ps* itself is in the resulting set, since it may also be the description value of identifier $nm \neq n$. In this case it is trivial to prove that $ps \Leftrightarrow$ true and *ps* may be deleted from the description value of *n* (cf. page 74).

simp-hypotheses : *SEStateOp* × *PredS* × *Name* × *Index* → set of *PredS*

simp-hypotheses(*S*, *ps*, *n*, *ix*) $\triangleq$
  let *nmset* = dom *get-element*(*S*, *ix*) − {*n*} in
  collect-preds(*S*, *previous*(*ix*))
  ∪ $\bigcup_{nm \in nmset}$ *get-element*(*S*, *ix*)(*nm*)
  ∪ *get-element*(*S*, *ix*)(*n*) − {*ps*}

pre *pre-get-element*(*S*, *ix*)
  ∧ *get-element*(*S*, *ix*): *SE-map*
  ∧ *n* ∈ dom *get-element*(*S*, *ix*)
  ∧ *ps* ∈ *get-element*(*S*, *ix*)(*n*)
  ∧ *pre-collect-preds*(*S*, *previous*(*ix*))

Now *SIMPLIFY* is defined as below. The pre-condition checks that the *SEStateOp* contains *ps* in the right place (as given by *ix*) and that *ps* can be simplified to some *ps'* by rule *rr*. The post-condition then states that this rule should be applied to *ps* to get output *ps'*.

SIMPLIFY (*ps*: *PredS*, *rr*: *Rule-ref*, *inst*: *Instantiation*, *ix*: *Index*,
  n: *Name*) *ps'*: *PredS*

ext rd *S* : *SEStateOp*
  rd *module* : *Theory-ref*
  rd *wflag* : 𝔹
  rd *fripse* : *Store*

pre *wflag* = false
  ∧ *pre-simp-hypotheses*(*S*, *ps*, *n*, *ix*)
  ∧ ∃*ps''*: *PredS* · ∃*hyp-set* ⊆ *simp-hypotheses*(*S*, *ps*, *n*, *ix*) ·
    let *rs* = *mk-RuleStmt*({ }, *hyp-set*, *ps* ⇔ *ps''*) in
    isProvenRuleStmt(*rs*, *rr*, *module*, *inst*, *RULES*(*fripse*), *THS*(*fripse*),
      THMORPHS(*fripse*))

post $\exists hyp\text{-}set \subseteq simp\text{-}hypotheses(\overleftarrow{S}, ps, n, ix) \cdot$
  let $rs = mk\text{-}RuleStmt(\{\}, hyp\text{-}set, ps \Leftrightarrow ps')$ in
  $isProvenRuleStmt(rs, rr, module, inst, RULES(fripse), THS(fripse),$
    $THMORPHS(fripse))$

Weak simplification

W_SIMPLIFY is specified just like *SIMPLIFY*, except that it uses the weak theory *wmodule* instead of *module*, and the conclusion of the rule is an implication rather than an equivalence. Weak simplification is not possible in the initial state when all the *PredS* that could be simplified have been introduced by *ASSUME* or *BELIEVE*.

W_SIMPLIFY (*ps*: *PredS*, *rr*: *Rule-ref*, *inst*: *Instantiation*, *ix*: *Index*,
  n: *Name*) *ps'*: *PredS*
ext rd *S* : *SEStateOp*
  rd *wmodule* : *Theory-ref*
  rd *fripse* : *Store*
pre len $S \geq 2$
  $\wedge$ *pre-simp-hypotheses*(*S*, *ps*, *n*, *ix*)
  $\wedge \exists ps'': PredS \cdot \exists hyp\text{-}set \subseteq simp\text{-}hypotheses(S, ps, n, ix) \cdot$
    let $rs = mk\text{-}RuleStmt(\{\}, hyp\text{-}set, ps \Rightarrow ps'')$ in
    $isProvenRuleStmt(rs, rr, wmodule, inst, RULES(fripse), THS(fripse),$
      $THMORPHS(fripse))$
post $\exists hyp\text{-}set \subseteq simp\text{-}hypotheses(\overleftarrow{S}, ps, n, ix) \cdot$
  let $rs = mk\text{-}RuleStmt(\{\}, hyp\text{-}set, ps \Rightarrow ps')$ in
  $isProvenRuleStmt(rs, rr, wmodule, inst, RULES(fripse), THS(fripse),$
    $THMORPHS(fripse))$

Storing results of simplification

When an expression has been simplified, *REMEMBER* saves the simplified value in the state by replacing the old ps_1: *PredS* with the new ps_2: *PredS*. This is done using the auxiliary function *replace*:

$replace : PredS \times PredS \times SEStateOp \times Index \times Name \rightarrow SEStateOp$

$$
\begin{array}{l}
replace(ps_1, ps_2, S, ix, n) \triangleq \\
\quad mk\text{-}SEStateOp(\\
\qquad \{i \mapsto \text{if } i = \text{hd } ix \\
\qquad\quad \text{then if } SEQ(S)[i]\text{: } SE\text{-}map \\
\qquad\qquad \text{then } \{nm \mapsto \left\{ \begin{array}{ll} SEQ(S)[i](n) - \{ps_1\} \cup \{ps_2\} & \text{if } nm = n \\ SEQ(S)[i](nm) & \text{otherwise} \end{array} \right\}\} \\
\qquad\qquad \text{else } replace(ps_1, ps_2, SEQ(S)[i], \text{tl } ix, n) \\
\qquad\quad \text{else } SEQ(S)[i] \\
\qquad | \; i \in \{1, \ldots, \text{len } SEQ(S)\}\}, \\
\qquad INDEX(S))
\end{array}
$$

pre *pre-get-element*(S, ix)
$\quad \wedge$ *get-element*(S, ix): *SE-map*
$\quad \wedge ps_1 \in$ *get-element*$(S, ix)(n)$

Then *REMEMBER* is specified as

REMEMBER (ps_1, ps_2: *PredS*, *rr*: *Rule-ref*, *inst*: *Instantiation*, *ix*: *Index*, *name*: *Name*)

ext wr S : *SEStateOp*
rd *module* : *Theory-ref*
rd *wflag* : $\mathbb{B}$
rd *fripse* : *Store*

pre *pre-SIMPLIFY*$(ps_1, rr, inst, ix, n, S, module, wflag, fripse)$
$\quad \wedge$ *post-SIMPLIFY*$(ps_1, rr, inst, ix, n, ps_2, S, module, wflag, fripse)$

post $S = replace(ps_1, ps_2, \overleftarrow{S}, ix, n)$

Storing results of weak simplification

W_REMEMBER stores the results of *W_SIMPLIFY*. It is specified as

W_REMEMBER (ps_1, ps_2: *PredS*, *rr*: *Rule-ref*, *inst*: *Instantiation*, *ix*: *Index*, *name*: *Name*)

ext wr S : *SEStateOp*
rd *module* : *Theory-ref*
wr *wflag* : $\mathbb{B}$
rd *fripse* : *Store*

pre *pre-W_SIMPLIFY*$(ps_1, rr, inst, ix, n, S, module, fripse)$
$\quad \wedge$ *post-W_SIMPLIFY*$(ps_1, rr, inst, ix, n, ps_2, S, module, fripse)$

post $S = replace(ps_1, ps_2, \overleftarrow{S}, ix, n)$
$\quad \wedge wflag = \text{true}$

Checking logical expressions

CHECK checks whether a given *PredS ps* is provable in the theory *module*, given all the description values in the current *SEStateOp* up to index *ix*. Of course, this will in general be undecidable, therefore *CHECK* will answer either YES, it has found a proof, or NO, it has found a proof of $\neg ps$, or DONTKNOW, it has not found a proof and therefore does not know whether the expression is provable or not.

CHECK (*ps*: *PredS*, *ix*: *Index*) *r*: {YES, NO, DONTKNOW}

ext rd *S* : *SEStateOp*
rd *module* : *Theory-ref*
wr *fripse* : *Store*

post $(r = \text{YES} \Rightarrow \exists hyp\text{-}set \subseteq collect\text{-}preds(S, ix) \cdot$
$PROVABLE(mk\text{-}RuleStmt(\{\}, hyp\text{-}set, ps), module) = \text{YES})$
$\wedge (r = \text{NO} \Rightarrow \exists hyp\text{-}set \subseteq collect\text{-}preds(S, ix) \cdot$
$PROVABLE(mk\text{-}RuleStmt(\{\}, hyp\text{-}set, \neg ps), module) = \text{YES})$

Assuming a logical expression

Define the auxiliary function

$add\text{-}restriction : SEStateOp \times PredS \to SEStateOp$

$add\text{-}restriction(S, ps) \triangleq$
if last$SEQ(S)$: *SE-map*
then let $new = \{n \mapsto$ if $n \in$ dom last$SEQ(S)$
then last$SEQ(S)(n) \cup \{ps\}$
else $\{ps\}$
$\mid n \in mentions(ps)\}$ in
front$SEQ(S) \frown$ last$SEQ(S) \dagger new$
else front$SEQ(S) \frown add\text{-}restriction($last$SEQ(S), ps)$

pre $mentions(ps) \neq \{\}$

ASSUME adds a given *PredS ps* to *assume*, i.e. assumes that this expression is true. This is mainly useful for simplifying expressions, in particular conditionals. In many cases, the user will first want to make a copy of the starting state, and come back to it later to assume $\neg ps$ in order to cover all cases.

The pre-condition of *ASSUME* only checks that *ps* does actually use a variable, since assuming a ground term would not make much sense.

ASSUME (*ps*: *PredS*)

ext wr *S* : *SEStateOp*
wr *assume* : set of *Assump*

pre $mentions(ps) \neq \{\}$

post $assume = \overleftarrow{assume} \cup \{mk\text{-}Assump(current\text{-}index(\overleftarrow{S}), ps)\}$
$\wedge\ S = add\text{-}restriction(\overleftarrow{S}, ps)$

Believing a logical expression

BELIEVE also assumes that a given logical expression is true. The difference to *ASSUME* is that this leads to a proof obligation that should later be discharged — the belief has to be justified. A *Belief* thus plays the rôle of a lemma that is used before it is proven. One special case when this can be particularly useful is in symbolic execution of *incomplete* specifications (cf. §7.4), where one may use a property of some component that cannot be proven yet because the component itself has not been specified yet.

Since it does make sense to **believe** a ground term, a new auxiliary function is needed to handle this case. If *ps* is a ground term, then the user has to provide the *Name* that *ps* gets associated with, since one can no longer automatically associate *ps* with those *n*: *Name* that are mentioned in *ps*.

$add\text{-}restriction\text{-}g : SEStateOp \times PredS \times Name \to SEStateOp$

$add\text{-}restriction\text{-}g(S, ps, n) \triangleq$
if last $SEQ(S)$: *SE-map*
then let $new = \{n \mapsto$ if $n \in$ dom last $SEQ(S)$
then last $SEQ(S)(n) \cup \{ps\}$
else $\{ps\}\}$ in
front $SEQ(S) \frown$ last $SEQ(S) \dagger new$
else front $SEQ(S) \frown add\text{-}restriction\text{-}g($last $SEQ(S), ps, n)$

BELIEVE (*ps*: *PredS*, *n*: [*Name*])
ext wr S : *SEStateOp*
wr *beliefs* : set of *Belief*
pre $mentions(ps) = \{\} \Rightarrow n \neq$ nil
post let $elem = get\text{-}element(\overleftarrow{S}, current\text{-}index(\overleftarrow{S}))$ in
let $current = \bigcup_{n \in \text{dom } elem} elem(n)$ in
$beliefs = \overleftarrow{beliefs} \cup \{mk\text{-}Belief(current\text{-}index(\overleftarrow{S}), current, ps)\}$
$\wedge$ if $mentions(ps) = \{\}$
then $S = add\text{-}restriction\text{-}g(\overleftarrow{S}, ps, n)$
else $S = add\text{-}restriction(\overleftarrow{S}, ps)$

Discharging *BELIEVE*d proof obligations

DISCHARGE discharges a *BELIEVE*d proof obligation. The pre-condition checks that there exists a rule or axiom that holds in the theory, and whose statement expresses the assumption.

DISCHARGE (*b*: *Belief*)
ext wr *S* : *SEStateOp*
rd *module* : *Theory-ref*
wr *beliefs* : set of *Belief*
rd *fripse* : *Store*
pre let $hyp\text{-}set' = collect\text{-}preds(S, previous(index(b))) \cup current(b)$ in
$\exists hyp\text{-}set \subseteq hyp\text{-}set' \cdot$
let $rs = mk\text{-}RuleStmt(\{\}, hyp\text{-}set, stmt(b))$ in
$PROVABLE(rs, module)$
post $beliefs = \overleftarrow{beliefs} - \{b\}$

6.2 User interface

6.2.1 General philosophy

This section describes some ideas about the user interface (UI) of SYMBEX. The UI is that part of a system that is directly accessible to the user and with which the user interacts. Note that it is not, as the name might suggest, an interface *between* the user and the system, but part of the system itself. So far, this thesis has mainly concentrated on the *functional* aspects of SYMBEX. The following now discusses some of the human factors involved. In particular, this includes the general layout of the UI and some of the commands that should be available. As stressed before, a good UI is essential if SYMBEX is to be useful, since the main purpose of such a system is to help the user convince herself of the correctness of a specification. Since every user is different, this implies that SYMBEX will have to be highly interactive, giving the user a lot of control. Additionally, we have to consider that we are working within the IPSE 2.5 framework, and a system for symbolic execution has to integrate into this framework.

At a more shallow level, the UI of SYMBEX will be based on a windowing system, making full use of the facilities provided such as windows, menus and and pointing facilities such as a "mouse".

6.2.2 The users

Most of the users of SYMBEX will be specifiers of some software system. They will usually be computer experts, although they might not use symbolic execution very often; in general, they should find it easy to understand the concepts behind using a symbolic execution system, but will often find it difficult to remember syntactic details of the facilities provided. This group is called "knowledgeable intermittent users" [Shn87, page 54]. They can be expected to have a good though not necessarily expert knowledge of the specification language.

Originally, it was hoped to support a second group of users as well, namely the users of the system being specified. This second group of users is *very* diverse, since it can contain all kinds of computer users. Usually, they will know little about the specification language, and probably close

to nothing about the symbolic execution system. Most of them will only want to use SYMBEX to validate *one* particular specification, and therefore never be able to gain much experience with it. However, during the development of SYMBEX it gradually became clear that the usefulness of symbolic execution crucially depends on *interactive* simplification. Symbolic execution will often not be able to provide useful results automatically, but provide a means of *interactive exploration* of a specification. While it is still intended to make SYMBEX as easy to use as possible for such inexperienced users, by giving them a lot of support such as suggesting default actions, it is no longer expected that many such users will be able to use SYMBEX successfully for its intended purpose, the validation of specifications. At least, they will usually require support by the specifier to do so.

Since it is hoped that at least some users will become experts in using the symbolic execution system, we usually provide commands both as menu choices and as short sequences of control characters to serve both the novice or intermittent users and the experienced users.

6.2.3 Information presentation

A central aim for the presentation of information gained by symbolic execution is that it should not look too similar to the original specification. Otherwise, if the user overlooked an error in the specification, the same will probably happen again with the results of symbolic execution.

One possible way to do so would be to use a paraphraser as is done in GIST (cf. §2.1.2), so as to make the results more readable. The GIST paraphraser translates formulae into sentences in a language close to natural English. This has been done for both GIST-specifications and for the results of symbolically executing them. The results of this seem quite impressive and genuinely do make it easier to understand a specification or the results of symbolic execution. However, the limited resources within the IPSE 2.5 project probably do not allow for building such a paraphraser, not even a language-specific one.

We also have to make sure that the information provided to the user is split up into fairly small chunks, which can be understood independently. This is partly achieved by associating parts of the information (in form of sets of *PredS*) with individual identifiers, which allows the user to extract (using the **show** command) those parts of the information that relate directly to a certain variable. A simple tactic that would help with the task of splitting up information would take any conjunction $ps_1 \wedge ps_2$ and split it into its components ps_1 and ps_2.

Another important method for making expressions, in particular formulae, easier to understand is to use a suitable unparsing algorithm (pretty printing). It is not intended to provide a separate pretty printing algorithm for symbolic execution, but such an algorithm will be used if it is provided by other parts of IPSE 2.5.

If new information is added, for example by an **assume**-clause introducing a new description value, then

- it is simplified using a tactic as previously determined by the user. (This includes the possibility that the tactic is empty and does not do any simplification at all.) Both unsimplified and simplified version of the description value are displayed (although display of

the unsimplified version can be switched off) and the simplified version is automatically **remembered**.

- if by the previous step a symbolic or actual value is derived for any variable v, i.e. a description value of the form $v = \ldots$ with "..." not containing v, then v is replaced by the new value in other description values. The result is first simplified, if possible, and then shown to the user. She then has the choice to **remember** this new result or to discard it.

  In this case, any other description values *ps* of v that can be simplified to true are deleted from the set of description values of v, since they no longer provide helpful information about v. The condition that *ps* can be simplified to true (or, equivalently, that *ps* can be derived from the other description values in the *SEStateOp*) is necessary to ensure that the deletion does not change the denotation of the *SEStateOp*. In practice, a very simple algorithm or tactic for checking this condition will probably be used, possibly one that only checks whether *ps* does occur elsewhere in the *SEStateOp*.

6.2.4 Windows

With every *SYMBEXSTATE* we associate a symbolic execution window, and vice versa. The information contained in such a symbolic execution window is defined as

$$
\begin{array}{lrl}
\mathit{SymbexWindow} :: & \mathit{NAME} & : \mathit{SYMBEXSTATE\text{-}ref} \\
& \mathit{RESULTS} & : \text{seq of } \mathit{ExecutionStep} \\
& \mathit{WFLAG} & : \mathbb{B} \\
& \mathit{ASSUME} & : \text{set of } \mathit{Assump} \\
& \mathit{BELIEFS} & : \text{set of } \mathit{Assump}
\end{array}
$$

where

$$
\begin{array}{l}
\mathit{inv\text{-}SymbexWindow}(sw) \triangleq \\
\quad \text{let } r = \mathit{RESULTS}(sw) \text{ in} \\
\quad \forall i \leq \text{len } r \cdot \mathit{SNAME}(r[i]) = \text{nil} \Leftrightarrow i = 1
\end{array}
$$

$$
\begin{array}{lrl}
\mathit{ExecutionStep} :: & \mathit{BUTTON} & : \mathit{StepButton} \\
& \mathit{SNAME} & : [\mathit{SpecName}] \\
& \mathit{RESULT} & : \mathit{Result} \mid \mathit{ResultButton} \mid \text{seq of } \mathit{ExecutionStep}
\end{array}
$$

$$
\begin{array}{lrl}
\mathit{Result} :: & \mathit{INPUT} & : \text{set of } \mathit{Name} \times \mathit{Type} \\
& \mathit{OUTPUT} & : \text{set of } \mathit{Name} \times \mathit{Type} \times \text{set of } \mathit{PredS} \\
& \mathit{READ} & : \text{set of } \mathit{Name} \times \mathit{Type} \\
& \mathit{WRITE} & : \text{set of } \mathit{Name} \times \mathit{Type} \times \text{set of } \mathit{PredS}
\end{array}
$$

where *StepButton* and *ResultButton* are disjoint sets of tokens. See page 102 for an example of a *SymbexWindow*. The actions that are possible from a *SymbexWindow* will be described in §6.2.5. The definition of *ExecutionStep* is recursive in order to model the recursion in the definition of *SEStateOp*. A new seq of *ExecutionStep* is started by the function *start-block* as introduced in §4.2.5, and ended by the function *finish-block*.

This definition of *SymbexWindow* is somewhat simplified, in particular it ignores the fact that the *SYMBEXSTATE* referred to by *NAME* already determines most of the other information in the *SymbexWindow*, which should be included as an invariant. Since it is not intended to give a fully formal specification of the UI of SYMBEX, this simplification does not cause any problems.

In addition to *SymbexWindow* windows will be needed that provide access to other parts of IPSE 2.5, in particular to FRIPSE and the LHS (cf. §1.4). Note however that in FRIPSE and the LHS, the term "view" is used rather than window. These windows should allow one to browse specifications, browse theories and prove theorems (e.g. for simplification), etc. No example of such a "simplification window" is given in this thesis, since they are just instances of the general FRIPSE prover windows currently under development by other members of the project.

Another type of window that might be added at a later stage is the "display window". At any stage during symbolic execution, the user would then be able to set up a window which displays some information about the state as decided by the user, such as the current value of a variable x. The user decides what information is displayed, and where within the window this is done. The possible actions associated with display windows are adding information to the display, removing information from the display, or moving information to a different place within the display. Note that these windows are read-only, they provide a view of the underlying information; it would not be possible to change the information itself within a display window, only the display of the information can be changed.

6.2.5 Overview and commands

As described in §1.4, support for formal reasoning in IPSE 2.5 consists of three main parts, the LHS which handles specifications and programs, the RHS which mainly consists of a theorem proving tool, and the symbolic execution system. The LHS will mainly provide a convenient starting point and browsing facilities for specifications and is therefore not absolutely necessary for SYMBEX, while the RHS will be essential for symbolic execution.

When starting up SYMBEX, the user first has to select the specification module *Mod* in which she wants to work. Implicitly, this also selects the relevant theories *ThModule*(*Mod*) etc. One of the options a user has when selecting a particular specification in the module is to symbolically execute it. The user is then asked to select, from a menu or list, a *SYMBEXSTATE* as a starting point. This list of *SYMBEXSTATE*s includes the initial state ARBITRARY. The chosen module *Mod* determines the fields *module*, *wmodule* and *fripse* of ARBITRARY.

Selecting a particular *SYMBEXSTATE symbex* from the list, one gets a menu with options (see below for explanations)

copy asks for a value $i \leq \text{len}\,SEQ(S)(symbex)$ and runs *copy-SYMBEXSTATE* to create a copy of *symbex*, restricted to its first i elements. Then a window is started up on this new *SYMBEXSTATE*.

select starts up a *SymbexWindow* on *symbex*.

Among other things, this new window contains, for every element of the *SEStateOp* sequence

- a *StepButton* that provides a convenient way of accessing this particular element of the sequence.
- the appropriate element of its associated *history*, the sequence of *SpecNames* that have been executed so far. For the first element in the sequence, the *SpecName* is nil since this element denotes the starting state of the symbolic execution.
- the result of the symbolic execution, as described below. These may be collapsed into a *ResultButton*. If the appropriate element of the *SEStateOp* is an *SEStateOp* itself, for example because the specification executed is a block or a loop, then the result is itself a sequence of execution steps.

However, there are some cases when it would not be useful to display the results of symbolic execution quite in the way described. First of all, it may contain a lot of duplication, since several identifiers may have the same *PredS* as description value. Second, some of the description values may be instantiations of a data type invariant (see the rule for VDM-operations in §4.2.6 for an example) which the user may or may not want to see at each step. In both cases, this should be handled by allowing the user to set a flag. If the first flag is set, any *PredS* is only shown once, copies are hidden behind "...". This allows the user to get hold of the duplicates again at any time. If the second flag is set, then similarly any *PredS* that is appropriately marked in the operational semantics is hidden. The marking could for example be done by distinguishing two different kinds of *PredS* in the definition of the operational semantics — those that are displayed by default and those that are hidden by default if the flag is set. This second class would then include the invariants.

It is important to note here that both these flags only concern the default *display*, the underlying state is not affected and the user can always hide or "unhide" a *PredS* as required.

Actions on an *ExecutionStep*

Selecting a particular element of the sequence (by "pushing its *StepButton*") leads to a menu with the options listed below. Except for **copy**, these options are only available if *ExecutionStep* has as *RESULT* a *Result* or *ResultButton* rather than another seq of *ExecutionStep*. This is the case iff the relevant element of *SEQ* of the *SEStateOp* is a map rather than an *SEStateOp* itself.

copy runs *copy-SYMBEXSTATE* to create a new *SYMBEXSTATE* which is a copy of *symbex*, restricted to its first elements up to the selected one. Then a window is started up on this new *SYMBEXSTATE*.

symbolic execution this option runs operation *SYMB_EXECUTE*, thereby extending the sequence *SEStateOp*. The user is first asked to choose a specification as argument if she has not done so yet. The result is simplified according to a simplification tactic given by the user in advance, and then displayed as described below. **symbolic execution** is not available if the *wflag* in the *SYMBEXSTATE* has been set.

weak symbolic execution is like the above, but runs *W_SYMB_EXECUTE* and is available even if the *wflag* has been set.

assume runs operation *ASSUME*, asking for a *PredS* as input.

believe runs operation *BELIEVE*, asking for a *PredS ps* as input. If *ps* is a ground term, then a *Name n* must also be provided and *ps* is stored as a description value of *n*.

show runs operation *SHOW*, asking for the name *n* of a variable and displaying the appropriate value of this variable. With an additional argument *ix*: *Index*, **show** then runs a tactic that tries to express the value of *n* in terms of the values of variables at stage *ix* in the execution sequence.

check asks for a *PredS ps* as input and runs the operation *CHECK* on it. This is done by starting a FRIPSE prover window and running a tactic in it that tries to prove *ps* from the description values that are known at that stage. If the tactic is not successful in proving *ps*, the user may either try to do it "by hand" (possibly using other tactics) or stop, in which case the result of running *CHECK* will be DONTKNOW.

If an option is selected that changes the information contained in the *SEStateOp* (i.e. **symbolic execution, weak symbolic execution, assume** or **believe**), on any element of the sequence other than the last one, a new *SYMBEXSTATE* (with a new *SymbexWindow*) is spawned (using the function *copy-SYMBEXSTATE*) which is identical to the previous one up to the selected element. The appropriate action is then performed on this new *SYMBEXSTATE*. If the selected element is the last one in the *SEStateOp*, this *SEStateOp* is updated or extended accordingly (in the same window).

Displaying the results of symbolic execution

After a specification has been symbolically executed, those parts of the *SEStateOp* that have changed should be displayed. Usually, this change consists of adding one or more elements to the sequence of the *SEStateOp*, although in theory a transition might also change those parts of the *SEStateOp* already there.

First consider the case when the added (or changed) element is a map. In this case, the results of symbolic execution of a specification consist of (cf. definition of *Result* in §6.2.4)

- input variable(s) — name and type
- output variable(s) — name, type and description value
- read variable(s) — name and type
- write variable(s) — name, type and description value

As an example, consider the results of symbolically executing the operation OP_1 from Example 4.1.2:

write x: $\mathbb{Z}$
write y: $\mathbb{Z}$
value of x: provided $x_0 \geq 0$ then $x_1 = x_0 + 1$
value of y: provided $x_0 \geq 0$ then $y_1^2 \leq x_0$

Here x_i was used to denote $\tilde{\sigma}([i+1], x)$ (similar for y). The difference of 1 is introduced to ensure that the sequence of x_i starts with the initial value x_0 instead of x_1.

If the new element is not a map but an *SEStateOp* itself, then this should be expressed in a rule of the form

$$\frac{\langle sn\text{-}seq', \text{last}\,SEQ(start\text{-}block(S))\rangle \hookrightarrow \langle [\,], S'\rangle}{\langle sn\text{-}seq, S\rangle \hookrightarrow \langle [\,], add\text{-}to\text{-}SEStateOp(S, finish\text{-}block(S'))\rangle}$$

For such rules (including, for example, Rules 4.2.7 for blocks and 4.2.12 for `while`-loops), the symbolic execution done to discharge the hypothesis is displayed as an *ExecutionStep* whose result is itself a sequence of *ExecutionSteps*, thus modelling the structure of the *SEStateOp*. The functions *start-block* and *finish-block* were introduced to explicitly allow SYMBEX to do this even if the hypothesis of the rule has not been (fully) discharged yet.

For other rules that have a transition as a hypothesis, SYMBEX first tries to discharge them automatically, without user interaction. If this is possible then the symbolic execution needed to do so is not itself displayed, although of course they will usually be used in the result and displayed there. If it is not possible then a separate *SymbexWindow* will be opened to discharge the hypothesis.

Actions on description values (*PredS*)

From the result of symbolic execution, one can now select a particular description value (or *PredS*). The menu of available actions then contains

simplify runs the operation *SIMPLIFY*. The result can then be stored in the *SEStateOp* using *REMEMBER*. If the *wflag* in the *SYMBEXSTATE* has been set, then this option is not available.

weak simplify runs the operation *W_SIMPLIFY*. The result can then be stored in the *SEStateOp* using *W_REMEMBER*.

hide hides the selected *PredS* behind "...".

unhide A hidden *PredS* can be made visible again at any time by "unhiding" the "...".

When selecting **simplify** or **weak simplify**, the user can decide either to select a rule from those existing already in the simplification theory $Th(\mathcal{L})$, or to build a new rule. The latter starts a new FRIPSE window, ready to prove a rule of the form

$$hyp\text{-}set \vdash ps \Leftrightarrow ps'$$

or, in the case of **weak simplify**,

$$hyp\text{-}set \vdash ps \Rightarrow ps'$$

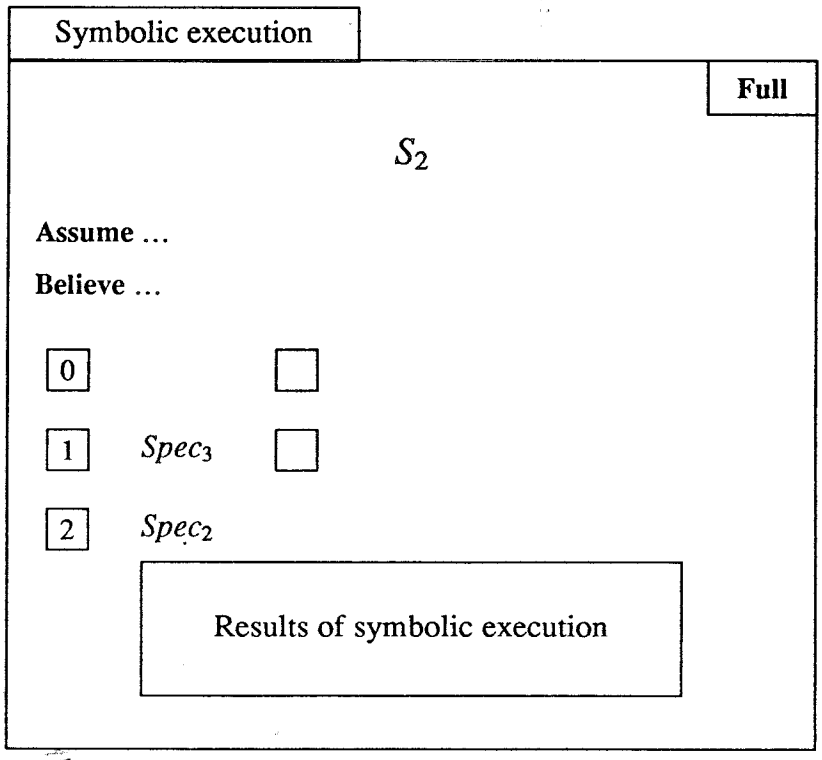
for some ps': *PredS* and *hyp-set* of hypotheses as allowed by the specification of *SIMPLIFY*, in the theory $Th(\mathcal{L})$. The user would have to fill in ps' and *hyp-set* herself, this cannot be done automatically. Hopefully, the pattern matcher supplied by FRIPSE would then able to generate the appropriate instantiation, otherwise it too has to be provided by the user.

When a rule has been selected (possibly a newly created one), then this rule can be "applied" to ps, i.e. ps can be replaced by ps'. If the user decides that the result is actually a simplification, she can **remember** the result, i.e. store it in the *SEStateOp*. The command **simplify** itself only *displays* the new result.

A number of tactics will be needed to do simplification. One of these, used by **show**, tries to express the value of a variable x at stage i in terms of the values of variables at stage j for $j < i$. In general, this would only work if the description values concerned really denote *symbolic* values, i.e. the description values take the form $x_i = f(x_j)$. However, whenever this tactic is applicable, it will provide a useful way of understanding the specification.

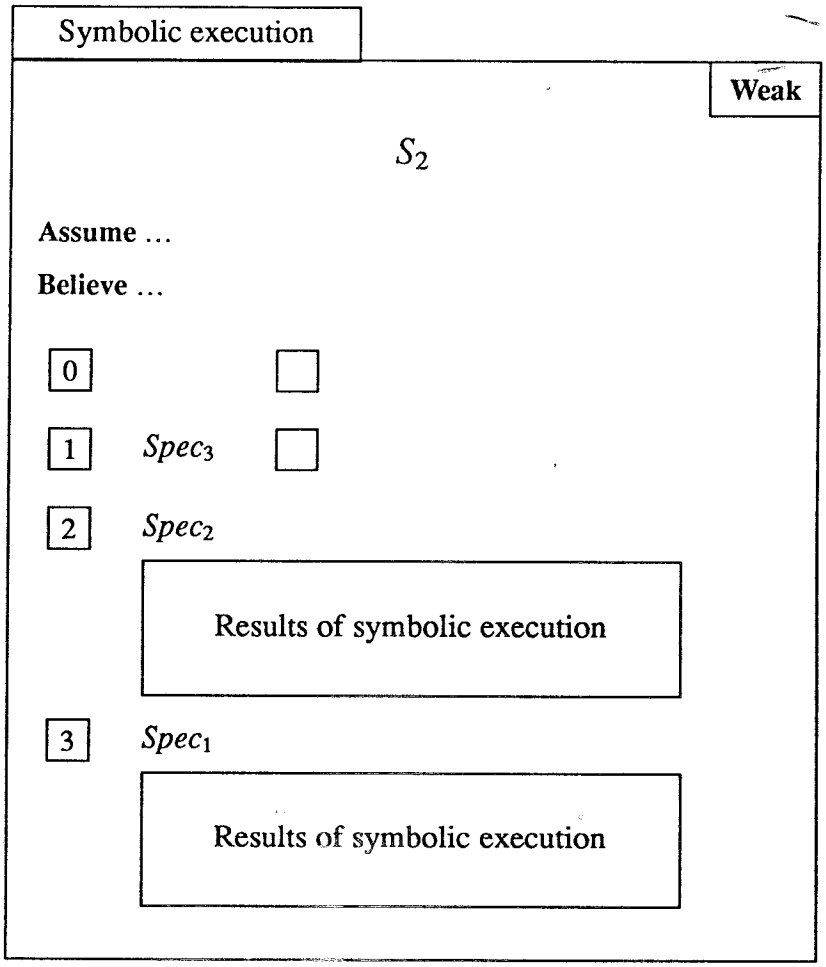
6.2.6 Example

Assume the user selects a specification $Spec_2$, say from an LHS-browser. The user is now offered a menu and selects **symbolic execution**. Prompted by a further menu, she selects to start in the previously created S_2: *SEStateOp*. As a result, the following window appears and displays the results of symbolically executing $Spec_2$ on S_2:



As one can see, S_2 had previously been built by symbolically executing $Spec_3$, starting originally from ARBITRARY (but possibly running some **assume** or **believe** commands first). The results of this first symbolic execution have been collapsed into a button. The results of symbolically executing $Spec_2$ are visible, they take the form shown on page 100 for the example operation OP_1. The Full marker shows that the results have been derived using *full* symbolic execution, the *wflag* is set to false.

Selecting button [2] in this window, the user is presented with a menu of options and selects **weak symbolic execution**. A menu of possible specifications is displayed, from which she selects $Spec_1$. As a result, the symbolic execution window becomes



One of the available options now would be to select a *PredS* within these results displayed and to simplify it, building an appropriate rule in the theory $\widetilde{Th(\mathcal{L})}$ in the process.

6.2.7 Keeping and displaying information about sessions

Information about symbolic execution sessions takes two different forms, which have to be stored and made accessible to the user in different ways:

- The results of symbolic execution itself are stored in the relevant *SEStateOp* (inside some *SYMBEXSTATE*) and accessible there, for example using *SHOW*.
- In addition to that, the user will want to 'commit' some information about the results of a session, without storing all available information. In particular, the user may become convinced that a certain part of a specification is correct and needs no further checking. In this case, information about earlier executions, as stored in *SEStateOp*s, is no longer

needed. Since 'committed' information relates directly to some part of the specification, it should be stored and accessed as annotation to that part.

The two kinds of information described here are quite different in the sense that the first is *dynamic* information about an execution itself, while the second is *static* information that was originally derived from an execution, but associated more closely with the appropriate part of the specification. It therefore seems useful to treat them separately and provide different ways of accessing them.

As described above, the appropriate *SymbexWindow* is used in order to display and access dynamic information. An alternative way, which may be added at a later stage, considers the execution information as a tree and displays it as such. The nodes in this tree will be some form of pointers to the relevant *ExecutionSteps* in a *SymbexWindow*. It should then be possible to access an *ExecutionStep* and the information associated with it by pointing at the appropriate node in the tree. There are two kinds of edges in this tree, representing the two ways of moving from one state to another, namely executing an operation and introducing an assumption.

6.3 Design and implementation issues

Design and implementation of SYMBEX are not considered to be essential parts of this thesis and are therefore only touched upon. Like FRIPSE, SYMBEX will be implemented in SMALLTALK-80. The reason for this decision is that the SMALLTALK-80 programming environment already provides many of the primitives needed. It scored heavily over its main rival ML because of its much better user interface, both of the environment and of programs written in it. This makes it much easier to provide a UI for SYMBEX that satisfies the requirements described above.

Integration with FRIPSE will be achieved by providing a translation mechanism for rules and *PredS* between FRIPSE and SYMBEX. For example, the rule with statement *rs* as constructed by the operation *SYMB_EXECUTE* is then translated into a *Rule* in FRIPSE, and the tactic *transform* is used in the theory referred to by *mod* to prove *rs* and instantiate the variables S' and *hyp-set*. Once this has been done, S' is translated back into the language of SYMBEX and replaces the previous *SEStateOp S* in *SYMBEXSTATE*.

If user intervention is required, as in the case of *SIMPLIFY*, the rule is treated in the same way, except that the rule is proven with user help (in a separate window), but the result is then translated back to SYMBEX in the same way.

6.4 Requirements on IPSE 2.5

The symbolic execution system described here will make use of a number of facilities provided elsewhere in the project. This of course implies certain requirements on the rest of the project. Some of these are 'must'-requirements, i.e. the SYMBEX system will not be able to work without them. Others are 'want'-requirements, which help to provide better support for symbolic execution, but are not absolutely necessary. In this section, we give a summary of these different

requirements.

Since we want to use symbolic execution interactively, this implies that response times have to be short — if the user has to wait too long for a reply, the system will not be used.

It must be possible to do different things in different windows at the same time, e.g. browse a specification, symbolically execute it, and use the theorem proving tool. These windows should be able to communicate easily, for example a result that has been shown using the theorem prover, such as a simplification, can immediately be used in symbolic execution. It must be possible to input an expression that already occurs on the screen by just pointing at it, using a pointing device such as the "mouse". Even though this requirement is not strictly necessary, symbolic execution would become very awkward to use if it is not satisfied.

Tools that will be needed but are not expected to be provided as part of the symbolic execution system, but rather as part of the general IPSE 2.5, include the following:

1. theorem proving tools, as discussed in §5.2. This could be a fairly simple proof editor, an automatic theorem prover, or, most likely, something in between the two. It is essential that this tool allows proofs with unknown or incompletely known goals.

2. support for writing and using tactics, including in particular the tactic *transform*. The tactic language should include the **hide** and **unhide** commands for hiding a *PredS* behind "...", and making it visible again.

3. tools for simplifying and rewriting expressions, as discussed in §5.2.

4. solver for simple (in-)equalities in $\mathbb{N}$. Even though this might not be *necessary* for symbolic execution, it would be very useful and support the simplifier considerably. As an example, in the scenario on page 120, one wants to derive, from $1 < n \wedge -2 < n$ for n:$\mathbb{N}$, that $n = 2$. The solver should be able to do this. It might be realised as a proof tactic, or possibly as a separate tool (oracle) that would then be accessible to the prover and to SYMBEX.

   In addition, such a solver could be used to detect inconsistent sets of restrictions, for example when, as result of a case distinction, only one branch in a conditional statement or expression is possible.

5. pretty printer. This should help us to display expressions in a more readable fashion, and thus make them easier to understand. This again is a "want"-requirement which would make SYMBEX more useful, but is not absolutely necessary.

6. Finally, it would be useful to have suitable help facilities, including a tutorial for new users of the symbolic execution system. If such facilities are provided, then they should be integrated into a general help system for IPSE 2.5. However, at the current stage it is not intended to include any help facilities into IPSE 2.5 or SYMBEX.

At least rudimentary versions of the first two will be absolutely necessary for a symbolic execution system.

The theorem proving tool mentioned in item 1. needs to provide support for proofs in several different theories in parallel, in particular the theories *ThModule*(*Mod*) etc for symbolic execution itself, and $\widetilde{Th(\mathcal{L})}$ for simplification. Although such structuring of theories is not absolutely

essential, it does make development of the necessary theory much easier (similar to modularity in software development).

It seems easiest if the logic and theories used can be expressed in natural deduction style (as in FRIPSE), in which case one needs to be able to express rules of the form $\frac{conf_1 \hookrightarrow conf_2 \varphi}{conf_1' \hookrightarrow conf_2'}$ where the *conf* are expressions denoting configurations. If the theorem proving tool does not support natural deduction style then any other style that allows one to express an equivalent calculus of configurations and transitions will be acceptable.

Chapter 7

Examples of the use of symbolic execution

> 'I'll tell you what the problem is mate,' said Majikthise, 'demarcation, that's the problem!'
>
> 'We demand,' yelled Vroomfondel, 'that demarcation may or may not be the problem!'
>
> 'You just let the machines get on with adding up,', warned Majikthise, 'and we'll take care of the eternal verities thank you very much. You want to check your legal position you do mate. Under law the Quest for Ultimate Truth is quite clearly the inalienable prerogative of your working thinkers. Any bloody machine goes and actually *finds* it and we're straight out of a job aren't we? I mean what's the use of our sitting up half the night arguing that there may or may not be a God if this machine only goes and gives you his bleeding phone number the next morning?'
>
> 'That's right,' shouted Vroomfondel, 'we demand rigidly defined areas of doubt and uncertainty!'
>
> *Douglas Adams: The Hitchhiker's Guide to the Galaxy*

This chapter gives some examples that show the use of symbolic execution and SYMBEX. It starts off with the example of a specification describing a bank, which shows some general points about symbolic execution in SYMBEX. After that, some more specific issues are taken up, such as iteration and incomplete specifications, and examples given that show how these can be handled. The result expressions presented in the following are always those that arise after some default simplification, even though, as described in §6.2.3, both the un-simplified and the simplified expression should be displayed by SYMBEX.

7.1 The dreaded bank example

The following example is based on [Jon86, §6.3].

7.1.1 Data structure

$bank :: acm : \text{map } Acno \text{ to } Acdata$
$\quad\quad odm : \text{map } Cno \text{ to } Overdraft$
where

$inv\text{-}Bank(mk\text{-}Bank(acm, odm)) \triangleq$
$\quad \forall mk\text{-}Acdata(cno, bal) \in \text{rng } acm \cdot$
$\quad\quad cno \in \text{dom } odm \wedge bal \geq -odm(cno)$

$Acdata :: own : Cno$
$\quad\quad bal : Balance$

$Cno = \mathbb{N}$

$Acno = \mathbb{N}$

$Balance = \mathbb{Z}$

$Overdraft = \mathbb{N}$

7.1.2 Operations

Introduce a new customer:

$NEWC\ (od: Overdraft)\ r: Cno$
ext wr odm : map Cno to $Overdraft$
post $r \notin \text{dom } \overleftarrow{odm} \wedge odm = \overleftarrow{odm} \dagger \{r \mapsto od\}$

Remove an existing customer:

$REMC\ (cno: Cno)\ r: \text{map } Acno \text{ to } Balance$
ext wr acm : map $Acno$ to $Acdata$
$\quad$ rd odm : map Cno to $Overdraft$
pre $cno \in \text{dom } odm$
post $r = \{acno \mapsto ba \mid \overleftarrow{acm}(acno) = mk\text{-}Acdata(cno, ba)\}$
$\quad \wedge acm = \text{dom } r \lhd \overleftarrow{acm}$

Open a new account for an existing customer:

NEWAC (*cno*: *Cno*) *r*: *Acno*
ext rd *odm* : map *Cno* to *Overdraft*
wr *acm* : map *Acno* to *Acdata*
pre $cno \in \text{dom}\, odm$
post $r \notin \text{dom}\, \overleftarrow{acm} \wedge acm = \overleftarrow{acm} \dagger \{r \mapsto mk\text{-}Acdata(cno, 0)\}$

Close an existing account:

CLAC (*acno*: *Acno*) *r*: *Balance*
ext wr *acm* : map *Acno* to *Acdata*
pre $acno \in \text{dom}\, acm$
post $\text{dom}\, acm = \text{dom}\, \overleftarrow{acm} - \{acno\} \wedge r = bal(\overleftarrow{acm}(acno))$

Get information about a customer's account(s):

ACINF (*cno*: *Cno*) *r*: map *Acno* to *Balance*
ext rd *acm* : map *Acno* to *Acdata*
post $r = \{acno \mapsto bal(acm(acno)) \mid$
$acno \in \text{dom}\, acm \wedge own(acm(acno)) = cno\}$

7.1.3 Symbolically executing the specification

In our example, the user starts off by introducing a new customer, and the system answers by giving some information about the specification of the operation called. For any *n*: *Name*, *ix*: *Index*, the value $\tilde{\sigma}(ix, n)$ is written as $n_{ix'}$, where ix' is defined by $ix'[i] = ix[i] - 1$ for $i = 1, \ldots, \text{len}\, ix$. ix' is used instead of ix to ensure that the initial value of n is denoted by n_0 rather than n_1. Furthermore, in n_{ix} the brackets around the sequence ix are omitted, so that the overdraft in the example is denoted by od_1 rather than $od_{[1]}$.

Symbolic execution

Full

Bank

Assume { }

Believe { }

0 □

1 *NEWC*

input:	od_1: *Overdraft*
output:	r_1: *Cno*
write:	*odm*: map *Cno* to *Overdraft*
value of *r*:	$r_1 \notin \text{dom}\, odm_0$
value of *odm*:	$odm_1 = odm_0 \dagger \{r_1 \mapsto od_1\}$
	...

Here and in the following it is assumed that duplicate description values and the invariant are hidden behind "...", as described in §6.2.5.

Next, the user might introduce another new customer, and ask (using **show**) for the current value of *odm* in terms of the original value. This is found using the tactic described on page 102.

2 *NEWC*

input:	od_2: *Overdraft*
output:	r_2: *Cno*
write:	*odm*: map *Cno* to *Overdraft*
value of *r*:	$r_2 \notin \text{dom}\, odm_1$
value of *odm*:	$odm_2 = odm_1 \dagger \{r_2 \mapsto od_2\}$
	...

$odm_2 = (odm_0 \dagger \{r_1 \mapsto od_1\}) \dagger \{r_2 \mapsto od_2\}$

If the user had just asked for **show**(*odm*), the system would have displayed the description value of the current value of *odm*.

It would be nice if the two map overwrites in odm_2 could be simplified into one, but in order to do this the system would have to deduce that $r_1 \neq r_2$. The user will probably have to prove this

manually, using the theorem prover. In this case, the system will hopefully be able to generate the proof obligation $r_1 \neq r_2$ automatically. Whether this is possible will however mainly depend on the strength of the simplification tactics used.

Since this new description of odm_2 seems to be clearer than expressing it in terms of odm_1, the user would now simplify the value of odm_2 accordingly. This is done by selecting the value of odm_2, issuing the command **simplify**, and selecting the relevant simplification tactic or rule from a menu. The command **remember** then ensures that in future, when asking for the value of odm_2 (be it explicitly or implicitly), the user would always gets this new description as an answer:

$$odm_2 = odm_0 \dagger \{r_1 \mapsto od_1, r_2 \mapsto od_2\}$$

The user might now decide to create a new account. Since accounts should only be opened for customers who have got a customer number, SYMBEX now has to handle a pre-condition:

3 *NEWAC*

input:	cno_3: *Cno*
output:	r_3: *Acno*
read:	*odm*: map *Cno* to *Overdraft*
write:	*acm*: map *Acno* to *Acdata*
value of *r*:	provided $cno_3 \in \text{dom}\, odm_2$
	then $r_3 \notin \text{dom}\, acm_2$,
	$acm_3 = acm_2 \dagger \{r_3 \mapsto mk\text{-}Acdata(cno_3, 0)\}$
value of *acm*:	...

Let the user now **assume** that $cno_3 \notin \text{dom}\, odm_2$. This assumption violates the pre-condition of the third operation executed, and the system therefore issues a warning. Since the assumption was simply the negation of the pre-condition, SYMBEX should be able to notice and deduce this automatically. In more complicated cases, the user will have to prove such statements manually using the theorem prover.

Since the pre-condition of the operation NEWAC is not satisfied, the specification does not restrict the results of NEWAC at all. Therefore all variables now have the value true.

Symbolic execution

Full

Bank

Assume $\{cno_3 \notin \text{dom}\, odm_2 \quad (index\ 3)\}$

Believe $\{\ \}$

0

1 *NEWC*

2 *NEWC*

3 *NEWAC*

input:	cno_3: *Cno*
output:	r_3: *Acno*
read:	*odm*: map *Cno* to *Overdraft*
write:	*acm*: map *Acno* to *Acdata*
value of *r*:	true
value of *acm*:	true

WARNING: pre-condition of operation 3 (NEWAC) does not hold

The 'index 3' denotes that the assumption has been introduced on the third element of the sequence. Instead of continuing execution on this state, the user might now decide to go back to the previous state, before the assumption, by pointing at the appropriate button 2 and then selecting **copy**, and then execute *NEWAC* to recreate the new account. Now let the user execute *NEWAC* once more and create a second account. This time, the user might give the input 12345, which is interpreted as **assume** $cno_4 = 12345$:

4 *NEWAC*(12345)

input:	12345: *Cno*
output:	r_4: *Acno*
read:	*odm*: map *Cno* to *Overdraft*
write:	*acm*: map *Acno* to *Acdata*
value of *r*:	provided 12345 ∈ dom odm_3
	then $r_4 \notin$ dom acm_3,
	$acm_4 = acm_3 \dagger \{r_4 \mapsto mk\text{-}Acdata(12345, 0)\}$
value of *acm*:	...

The user might again decide to simplify expressions by making some assumptions, and **assume** that the pre-conditions of operations 3 and 4 are true. Actually, the assumption made below says slightly more than that, and the system has to derive the pre-condition itself. It does so (hopefully but not necessarily automatically), and then is able to conclude that all formulas above which depended on the pre-conditions actually do hold. The user would then ask for the current value of *acm* to be shown in terms of the original value. As before, the simplification of two map overwrites into one might have to be done by the user rather than automatically.

In the following, the results will be given as text, without showing the window in which it appears.

4 **assume** ($cno_3 \in$ dom $odm_2 \wedge 12345 \in$ dom odm_0)
12345 ∈ dom odm_3
$r_3 \notin$ dom acm_2
$acm_3 = acm_2 \dagger \{r_3 \mapsto mk\text{-}Acdata(cno_3, 0)\}$
$r_4 \notin$ dom acm_3
$acm_4 = acm_3 \dagger \{r_4 \mapsto mk\text{-}Acdata(12345, 0)\}$
show(*acm*, ...)
$acm_4 = acm_0 \dagger \{r_3 \mapsto mk\text{-}Acdata(cno_3, 0), r_4 \mapsto mk\text{-}Acdata(12345, 0)\}$

Next, the user would execute the operation ACINF, to find out what information is stored about the accounts of a customer. Since the answer given is not immediately useful, the user would ask for r_5 in the answer to be rephrased in terms of acm_0 instead of acm_5, using **simplify**. The result can then be simplified using case distinctions.

5 ACINF
input cno_5: *Cno*
output r_5: map *Acno* to *Balance*
read *acm*: map *Acno* to *Acdata*

$r_5 = \{acno \mapsto bal(acm_5(acno)) \mid$
$acno \in \text{dom}\, acm_5 \wedge own(acm_5(acno)) = cno_5\}$
simplify ...
if $cno_3 = cno_5 \wedge cno_5 = 12345$ then
$r_5 = \{acno \mapsto bal(acm_0(acno)) \mid$
$acno \in \text{dom}\, acm_0 \wedge own(acm_0(acno)) = cno_5\}$
$\dagger\{r_3 \mapsto 0\}$
$\dagger\{r_4 \mapsto 0\}$
if $cno_3 \neq cno_5 \wedge cno_5 = 12345$ then
$r_5 = \{acno \mapsto bal(acm_0(acno)) \mid$
$acno \in \text{dom}\, acm_0 \wedge own(acm_0(acno)) = cno_5\}$
$\dagger\{r_4 \mapsto 0\}$
if $cno_3 = cno_5 \wedge cno_5 \neq 12345$ then
$r_5 = \{acno \mapsto bal(acm_0(acno)) \mid$
$acno \in \text{dom}\, acm_0 \wedge own(acm_0(acno)) = cno_5\}$
$\dagger\{r_3 \mapsto 0\}$
if $cno_3 \neq cno_5 \wedge cno_5 \neq 12345$ then
$r_5 = \{acno \mapsto bal(acm_0(acno)) \mid$
$acno \in \text{dom}\, acm_0 \wedge own(acm_0(acno)) = cno_5\}$
remember ...

Such case distinctions might multiply very fast; in that case the user would have to introduce assumptions and only handle a small number of cases at a time.

To continue the example, let the user want to check what happens if a customer is removed. The system again has to distinguish several cases, depending on whether or not this customer was introduced by one of the previous operations, or was in the original data structure, or does not exist at all.

6 REMC
input cno_6: *Cno*
output r_6: map *Acno* to *Balance*
write *acm*: map *Acno* to *Acdata*
read *odm*: map *Cno* to *Overdraft*
provided $cno_6 \in \text{dom}\, odm_5$
then $r_6 = \{acno \mapsto ba \mid acm_5(acno) = mk\text{-}Acdata(cno_6, ba)\}$,
$acm_6 = \text{dom}\, r_6 \lhd acm_5$

By letting $cno_6 = r_2$, the user could now check what happens if the customer introduced in operation 2 is removed again, to make sure that the two operations cancel each other out. Since $cno_6 = r_2$ implies $cno_6 \in \text{dom}\, odm_5$, the precondition of operation 6, REMC, is satisfied. Therefore the system again displays the formulas that describe the postcondition of operation 6, this time without prefixing them with 'provided ...'.

Our example user now **checks** whether $odm_6 = odm_1$, since the customers and their overdraft limits should be the same as they were before customer r_2 was introduced. It is hoped that such

checks can be done automatically when using **check** (this command may therefore take fairly long to execute), but it is possible that the system will rely on help from the user for these deductions. As in many examples before, this depends on the strength of the tactics (and theories) used. It turns out that the domains of the two are *not* equal, the two operations therefore do not cancel each other out:

assume $cno_6 = r_2$
$r_6 = \{acno \mapsto ba \mid acm_5(acno) = mk\text{-}Acdata(r_2, ba)\}$
$acm_6 = \mathrm{dom}\, r_6 \lhd acm_5$
check $odm_6 = odm_1$
NO

In the separate prover window, the proof of $odm_6 \neq odm_1$ is displayed:

$$\begin{aligned} odm_6 &= odm_5 \\ &= \ldots \\ &= odm_2 \\ &= odm_1 \dagger \{r_2 \mapsto od_2\} \\ &\neq odm_1 \end{aligned}$$

Closer investigation shows that REMC does not actually remove a customer, it only removes the customer's accounts. This can be corrected as follows:

REMC (*cno*: *Cno*) *r*: map *Acno* to *Balance*
ext wr *acm* : map *Acno* to *Acdata*
wr *odm* : map *Cno* to *Overdraft*
pre $cno \in \mathrm{dom}\, odm$
post $\mathrm{dom}\, odm = \mathrm{dom}\, \overleftarrow{odm} - \{cno\}$
$\wedge\, r = \{acno \mapsto ba \mid \overleftarrow{acm}(acno) = mk\text{-}Acdata(cno, ba)\}$
$\wedge\, acm = \mathrm{dom}\, r \lhd \overleftarrow{acm}$

Ideally, one would now like to have some kind of version control tool that goes through all the *SEStateOps* in the current *Module* and updates them with the new definition of *REMC*. Those parts of the *SEStateOp* that are not affected by the change would remain unchanged, while those parts that are affected would be changed accordingly, or at least marked as 'unsafe'. Such a version control tool may be added later, but is not considered to be part of the main body of SYMBEX. For the time being, changes to the *SEStateOp* because of changes to the underlying specification have to be made by hand.

The proof of $odm_6 = odm_1$ goes through, i.e. the two operations now cancel each other out (remember $r_2 \notin \mathrm{dom}\, odm_1$):

$$\begin{aligned} \mathrm{dom}\, odm_6 &= \mathrm{dom}\, odm_5 - \{cno_6\} \\ &= \mathrm{dom}\, odm_2 - \{r_2\} \\ &= (\mathrm{dom}\, odm_1 \cup \{r_2\}) - \{r_2\} \\ &= \mathrm{dom}\, odm_1 \end{aligned}$$

Even though in theory, SYMBEX might now have noticed $odm_6 = odm_1$ without help from the user, this seems to be asking too much, since here simple rewriting is not sufficient, one needs proper formal reasoning support. It may even be too difficult to prove this statement automatically once the user has stated it, in this case another window should pop up giving access to the theorem prover.

7.2 Invariants and pre-conditions

Invariants

Before discussing in more detail what happens with invariants when further information is added, one has to distinguish the different rôles that invariants can play in a specification language. Invariants can be characterised as predicates defining *subtypes*, they restrict an abstract data type to those elements that satisfy the invariant. One possibility for treating invariants is to demand that for every specification of an operation acting on a state with an invariant, one has to *prove* that the resulting state again satisfies this invariant and therefore is of the correct type. With this interpretation, invariants play a similar rôle to that of assertions in a program. They help to understand and reason about the specification by introducing some redundancy, by asserting that the underlying data structure or state should have certain properties, but they do not guarantee this in themselves. This view of invariants is taken in INA JO (cf. page 16), and in VDM as described in [Jon80]. However, this interpretation has the disadvantage that, for many operations, one has to explicitly add a number of statements to the post-condition solely to ensure that the invariant is satisfied. In many cases, these additional statements are more or less a repetition of the invariant itself. The newer version of VDM as described in [Jon86] therefore takes the view that an invariant is part of the type definition. Then by providing an operation with the appropriate signature, one automatically implies that only resulting states satisfying the invariant are legal, because only they are of the correct type. Under this interpretation it therefore is not necessary to *explicitly* state this in the specification of the operation. Instead of having to prove that the resulting state of an operation always satisfies the invariant, it is then only necessary to prove *implementability*, i.e. the operation is consistent with the invariant, there exists a legal resulting state of the operation.

How should invariants be treated in symbolic execution? Under the second interpretation, where invariants in themselves *guarantee* certain properties, they should be regarded as part of the description values of all the variables mentioned in the invariant. This was for example done in the transition for VDM-operations in §4.2.6.

Under the first interpretation, there are two alternatives: since invariants introduce redundancy, one might just ignore them, they do not strictly speaking offer any additional information that cannot be deduced from the rest of the specification. However, it would usually still be be useful to use the explicit information provided by them, for example for checking a *PredS*, rather than leave it in an implicit form. Therefore, one might instead decide to treat invariants as description values, just as described above. It is probably best to provide both alternatives, by generally treating invariants as part of the description values, but giving the user the choice to leave them out. In this thesis invariants are treated as part of the description values.

If invariants, especially large ones, always get displayed, this may easily cause a flood of useless information and not really help in understanding the specification. On the other hand, if this is not done then there is a danger that invariants are ignored in symbolic execution. This could undermine the purpose of symbolic execution, since invariants probably cause many of the unexpected interactions between various parts of the specification. SYMBEX therefore compromises between the two by displaying all the invariants, but allowing the user to hide them. Hiding can be done either "by hand" by selecting the **hide** command, or by using it in a tactic.

So far, the discussion has ignored the problem of invariants on the whole state. If a specification only accesses individual components of the state, it is not obvious where in the operational semantics of the specification the invariant on the state should be mentioned. Two possibilities are

- add the invariant on the state to the description values of (at least one of) the variables denoting fields of the state that are accessed by the operation.
- add a new "dummy" variable that has as its description value the invariant on the state.

The choice between these two alternatives is made when defining the operational semantics of the language used, it does not need to be made in general beforehand.

Pre-conditions

How should pre-conditions be handled by SYMBEX? Pre-conditions are predicates over the state. Again, there are different ways of interpreting a pre-condition. In VDM, if an operation is called in a state that does not satisfy the pre-condition, then the specification does not restrict the output of the operation. All the variables in the state will therefore take arbitrary values after execution of the operation, or execution may not terminate at all. A slight variation of this would be to say that only those variables that a specification has got write-access to can take an arbitrary value (but again allowing non-termination). A third possibility is to consider it as an error if the pre-condition is not satisfied, and to demand that in this case, an appropriate error message is given. This thesis will not discuss the advantages and disadvantages of the different interpretations, but just show how they can be handled in symbolic execution.

Under any of these interpretations, a pre-condition can be considered as a special kind of conditional construct. Under the interpretation used in VDM, a pre-condition is handled by introducing constructs of the form 'provided *pre-condition* then *post-condition*'. If at a later stage more information about the variables in a pre-condition is available, for example after an **assume** command, then the relevant description value containing the provided-expression is simplified if possible. Since pre-conditions play a special rôle in specification, a warning should be given to the user if a pre-condition turns out to be false. This was the reason for introducing provided-then rather than using, for example, an if-then-else expression.

As was said before, the simplifier should automatically be invoked when new information is added, and try to simplify all pre-conditions and invariants that share some variables with the new information. Of course, the simplifier should only try such simplifications that actually use the new information, since only such simplifications can provide any new insights.

The effort involved probably will not be as much as one might expect since both pre-conditions and invariants usually relate to only one state, and therefore only to the variables of that state.

7.3 Iteration and recursion

Handling of recursion and iteration is a central part of a symbolic execution system, since in general the various forms of recursion and iteration are, besides implicit definitions, the most difficult constructs in a specification to fully understand. This means that it is particularly important for symbolic execution to provide facilities for handling them.

In general, a recursive function definition, say $fn(n) \triangleq$ if $n = 0$ then $g(n)$ else $h(n, fn(n-1))$, cannot be turned into an explicit definition. Therefore, the value of $fn(n)$, where n does not have an actual value, must be expressed as a description value consisting of the definition of fn, since expressing it as a symbolic or actual value is not possible. Even if n does have an actual value, it is not necessarily useful to evaluate $fn(n)$ in symbolic execution, because the information about the computation would then be lost. In some special cases, however, it is both possible and useful to be more explicit than giving description values. Some examples of such cases are given below. Even if it is not possible, the facilities described below should make it easier to understand the effect of a recursive definition.

Consider the function $fact: \mathbb{N} \to \mathbb{N}$ defined recursively by $fact(0) = 1, fact(n+1) = (n+1) \times fact(n)$, and the function f given by the program fragment

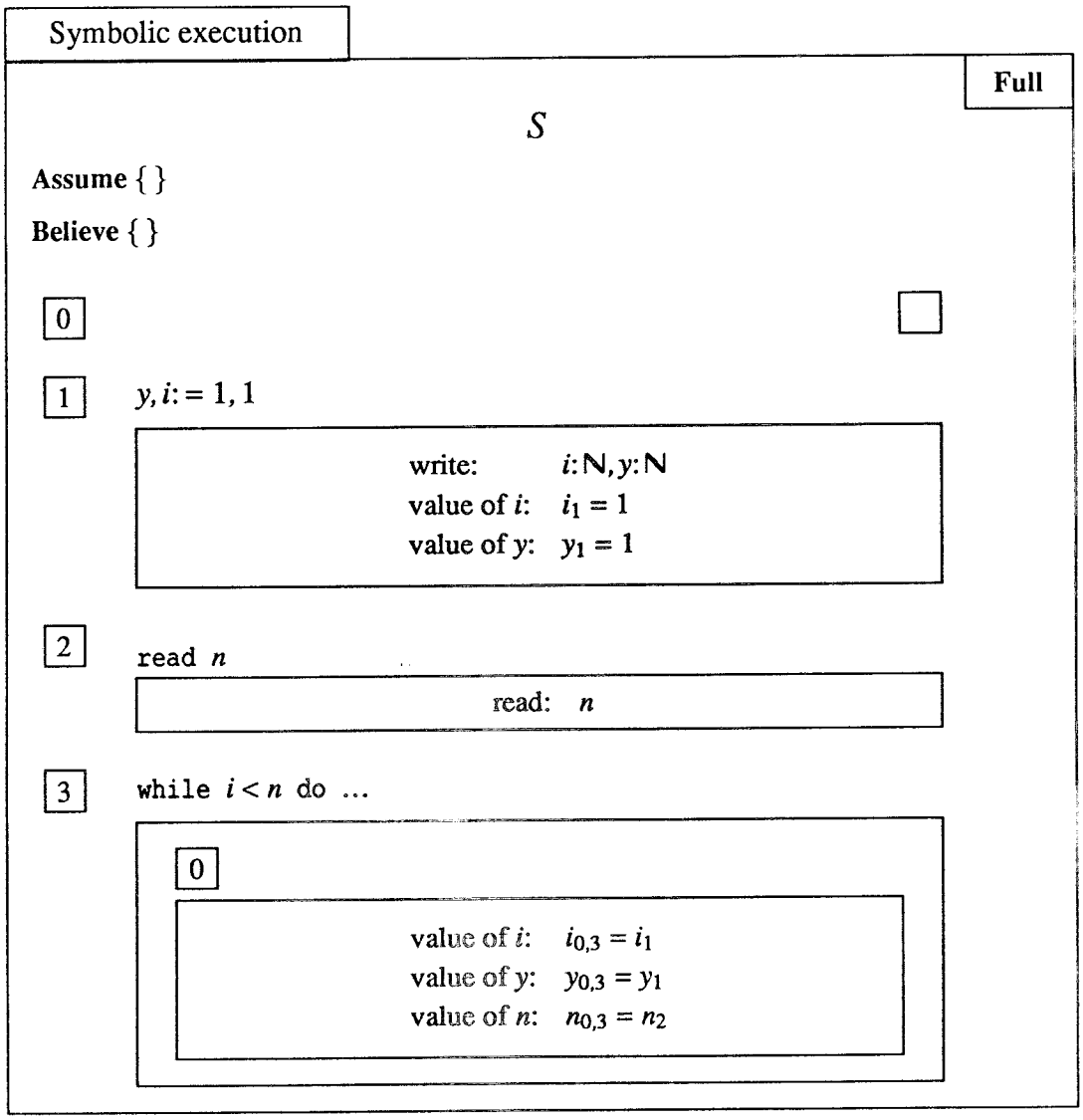
```
y, i := 1, 1
read n
while i < n
do y, i := y × i, i + 1 od
write y
```

Now consider what happens if this program fragment or procedure is symbolically executed, starting in some S: *SEStateOp*. The assignment at the beginning is easy to deal with, it leads to the description values $i_1 = 1$ and $y_1 = 1$; `read` n just introduces the variable n with empty description value. Next, the `while`-statement is handled according to the rules given in §4.2.8. Starting with the configuration $\langle$`while` $i < n$ `do` $\ldots, S\rangle$ for the new S: *SEStateOp*, the appropriate Rule 4.2.12 has as hypothesis the transition

$$\langle WHILE\ \varphi\ DO\ sn\ OD,\ \mathrm{last}\,SEQ(start\text{-}block(S))\rangle \hookrightarrow \langle [\,], S'\rangle \tag{7.1}$$

for some *SEStateOp* S'. Note that the same symbol S was used for the new *SEStateOp* after symbolically executing the assignment and the `read` statement as for the starting state, in order to emphasize the fact that the old one has been *transformed* or extended into the new one. No genuinely new *SEStateOp* has been created (cf. page 100). However, to prevent confusion the following discussion will denote the version of S after i steps as S_i. Similar for S_{ix} for any *Index* ix. S_{ix} is therefore the state that defines the values of nm_{ix} for nm: *Name*.

On the level of the operational semantics, this new *SEStateOp* introduced above then become an *element* in the sequence of the old one. On the UI level, this results in an *ExecutionStep* whose *RESULT* itself consists of a sequence of *ExecutionSteps*. At first, this sequence has length 1, where the first element is given as part of the definition of *start-block*. Assume for this example that the user started in the initial state ARBITRARY. The *SymbexWindow* then takes the form



This is without applying the trivial simplification $i_{0,3} = 1$, $y_{0,3} = 1$. It would be up to the user to decide whether such simplifications should be done immediately and automatically or not.

The "inner" box here represents the *SEStateOp*

$$S_{0,3}' \triangleq \text{last}SEQ(S_{0,3})$$
$$= \text{last}SEQ(start\text{-}block(S_1))$$

To find the right S' to discharge the hypothesis (7.1) of the above transition, one needs to evaluate the configuration

$$\langle WHILE\ i < n\ DO\ y, i := y \times i, i+1\ OD,\ S_{0,3}' \rangle$$

which by Rule 4.2.13 transforms into

$$\langle \texttt{if}\ i < n\ \texttt{then}\ \ cons(y, i := y \times i, i+1,\ WHILE\ \ldots),\ \texttt{else}\ \ [\,],\ S_{0,3}' \rangle \tag{7.2}$$

Now Rule 4.2.11 applies, which leads to the hypotheses

$$i_{0,3} < n_{0,3} \ \vdash\ \langle cons(y, i := y \times i, i+1, WHILE\ \ldots),\ S_{0,3}' \rangle \hookrightarrow \langle [\,], mk\text{-}SEStateOp(SEQ(S_{0,3}') \frown e\text{-}seq_1, INDEX(S_{0,3}')) \rangle \tag{7.3}$$

and

$$\neg i_{0,3} < n_{0,3} \vdash \langle [\,], S_{0,3}' \rangle \hookrightarrow \langle [\,], mk\text{-}SEStateOp(SEQ(S_{0,3}') \frown e\text{-}seq_2, INDEX(S_{0,3}')) \rangle \tag{7.4}$$

Then trivially $e\text{-}seq_2 = [\,]$, while (7.3) implies

$$\mathrm{hd}\ e\text{-}seq_1 = \{y \mapsto \{y_{1,3} = y_{0,3} \times i_{0,3}\}, i \mapsto \{i_{1,3} = i_{0,3} + 1\}, n \mapsto \{n_{1,3} = n_{0,3}\}\}$$

or, after applying some easy simplification

$$\mathrm{hd}\ e\text{-}seq_1 = \{y \mapsto \{y_{1,3} = 1\}, i \mapsto \{i_{1,3} = 2\}, n \mapsto \{n_{1,3} = n_2\}\} \tag{7.5}$$

Then $S_{1,3}' = add\text{-}to\text{-}SEStateOp(S_{0,3}', \mathrm{hd}\ e\text{-}seq_1)$ and hypothesis (7.3) turns into

$$1 < n_{0,3} \ \vdash\ \langle WHILE\ \ldots, S_{0,3}' \rangle \hookrightarrow \langle [\,], mk\text{-}SEStateOp(SEQ(S_{1,3}') \frown \mathrm{tl}\ e\text{-}seq_1, INDEX(S_{1,3}')) \rangle \tag{7.6}$$

with $\mathrm{tl}\ e\text{-}seq_1$ still unknown. Doing another iteration one gets $\mathrm{tl}\ e\text{-}seq_1 = [\,]$ in the case $1 < n_{0,3} \wedge \neg 2 < n_{1,3}$ (i.e. $n_2 = 2$), and

$$n_2 > 2 \vdash \mathrm{hd}\ \mathrm{tl}\ e\text{-}seq_1 = \{y \mapsto \{y_{2,3} = 2\}, i \mapsto \{i_{2,3} = 3\}, n \mapsto \{n_{2,3} = n_2\}\} \tag{7.7}$$

with the new hypothesis

$$n_2 > 2 \vdash \langle WHILE\ \ldots, S_{0,3}' \rangle \hookrightarrow \langle [\,], mk\text{-}SEStateOp(SEQ(S_{2,3}') \frown \mathrm{tl}\ \mathrm{tl}\ e\text{-}seq_1, INDEX(S_{2,3}')) \rangle \tag{7.8}$$

and so on. Restricting the number of iterations to at most 2 is equivalent to issuing the command[1]

assume $\neg i_{0,3} < n_{0,3} \vee \neg i_{1,3} < n_{1,3} \vee \neg i_{2,3} < n_{2,3}$

[1] There are several essentially equivalent possibilities for restricting the number of executions of the loop. Besides giving appropriate values to loop variables, as we have done here, one might explicitly put an upper limit on the number of iterations of the loop, or, as is done in EFFIGY (see §2.1.2), restrict the number of steps in any path to be taken.

or, equivalently,

assume $n_2 \leq 3$

Doing this the user gets

$\langle$`if` $i < n$ `then` *WHILE* ... `else` $[\,], S_{2,3}'\rangle \hookrightarrow \langle [\,], S_{2,3}'\rangle$

(using one of the simplification rules mentioned at the end of §4.2.7). Merging the results now as described in Rules 4.2.11 and 4.2.12 results (after some easy simplification) in the window given in Figure 7.1.

The user might not see yet what is going to happen for other values and decide to try something else. One possibility would be to give a limit on the number of iterations of the loop/number of recursions, say 5. In this case, the result is

if $n \leq 1$ then $f(n) = 1$
else if $n \leq 2$ then $f(n) = 1 \times 1 = 1$
else if $n \leq 3$ then $f(n) = 1 \times 1 \times 2 = 2$
else if $n \leq 4$ then $f(n) = 1 \times 1 \times 2 \times 3 = 6$
else if $n \leq 5$ then $f(n) = 1 \times 1 \times 2 \times 3 \times 4 = 24$
else if $n \leq 6$ then $f(n) = 1 \times 1 \times 2 \times 3 \times 4 \times 5 = 120$
else Nontermination
$y = 1 \times 1 \times 2 \times 3 \times 4 \times 5 = 120; i = 1 + 1 + 1 + 1 + 1 = 5$

When executing a loop such as this, it seems best to display the unsimplified as well as the simplified result, since the unsimplified form might make it easier to understand the general structure of these results.

Now consider the recursive function *fact*. Repeated unfolding of its definition, say 5 times, results in

if $n = 0$ then $fact(n) = 1$
else if $n = 1$ then $fact(n) = 1 \times 1 = 1$
else if $n = 2$ then $fact(n) = 2 \times 1 \times 1 = 2$
else if $n = 3$ then $fact(n) = 3 \times 2 \times 1 \times 1 = 6$
else if $n = 4$ then $fact(n) = 4 \times 3 \times 2 \times 1 \times 1 = 24$
else if $n = 5$ then $fact(n) = 5 \times 4 \times 3 \times 2 \times 1 \times 1 = 120$
else $fact(n) = n \times (n-1) \times (n-2) \times (n-3) \times (n-4) \times fact(n-5)$

Such unfolding can either be done by hand, or using an easy simplification tactic.

The user might now guess that

$f(n) =$ if $n = 0$ then 1 else $fact(n-1)$

This guess can then be checked. If SYMBEX is not able to prove this automatically, the user may instead do so by going into a theorem proving window and proving it there as a theorem. In this case, the user would have to choose the set *hyp-set* of hypotheses out of those *PredS* already known, as given by *collect-preds*$(S, [\,])$ in the specification of *CHECK*. Whenever the user now

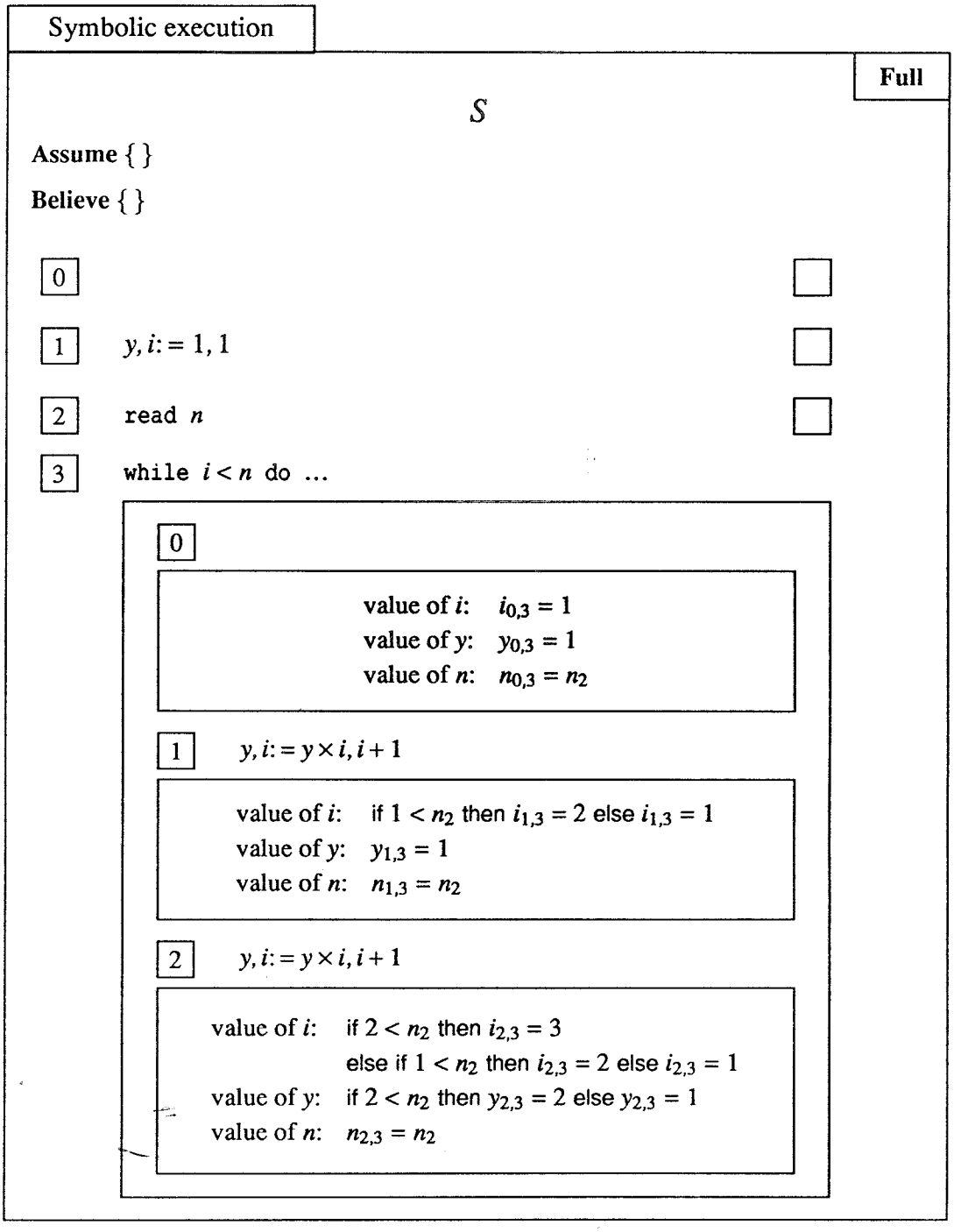


Figure 7.1: Window resulting from symbolic execution

comes across this procedure definition, it is possible to substitute $f(n)$ by if $n = 0$ then 1 else $fact(n-1)$.

Another way of dealing with loops is to include a suitable while-constructor in the language of *PredS*. Using this, one could move the `while`-construct of the specification language into the description language. Although this in itself would not help understanding of the specification, it would allow the user to move on to further exploration without making any assumptions about the number of iterations of the loop, and use a symbol for the result of symbolically executing the loop.

If one is content with *weak* symbolic execution then one can prove theorems about the loop, e.g. prove (presumably by induction) that some ps: *PredS* holds after any number of iterations of the loop. Then ps can be used as a description value describing the result of symbolically executing the loop. Since in general, ps will not be a complete description of the result, this only leads to weak symbolic execution. A similar approach is possible for recursive definitions.

7.4 Incomplete specifications

If a specification is developed in a top-down manner, then it may be possible to execute it symbolically even if it is unfinished. This is the case when the top-level of an operation or function has been specified, but operations or functions called from there have not. In this case, symbolic execution will only return the *name* of these operations or functions. Of course, these names cannot then be unfolded into their definitions, but this might nevertheless be a helpful tool for checking the overall structure of a specification before filling in the details.

A problem that arises is that a number of simplifications cannot be made, since they may rely on lower-level information. For example, introducing a new element to a data structure and then removing it again should leave the structure in its initial state. However, a system for symbolic execution cannot notice that $remove(x, add(x, ds)) = ds$, unless there is more information available about *add* and *remove*. Instead, the user might state this as an assumption, which can then be used. The system will store this assumption as a proof obligation, which the user should later discharge when *add* and *remove* have been specified.

As an example of symbolic execution of an incomplete specification consider the following:

SORT (l: seq of $\mathbb{N}$) r: seq of $\mathbb{N}$
post *is-permutation*$(l, r) \wedge$ *is-sorted*(r)

This might be part of a larger specification, and at an early stage the user might be quite content to check the interactions between this operation and others at this fairly high level, without defining the predicates *is-permutation* and *is-sorted*. At some stage in this process, one might need to know that for the result r of SORT, the sum of the first two elements is less than or equal to the sum of the last two. The user may now believe this until some later stage when *is-sorted* is defined. Then the *Belief* can be proved, which at the same time constitutes a check whether the definition of *is-sorted* matches the intention of the user.

At some later stage, the user then has to define *is-permutation* as well, but by now can be reasonably sure that it is really needed and the high-level specification was correct. Additionally, this process may have generated some requirements on the specification of *is-permutation*, namely the proof obligations mentioned above. This should make it easier to write a correct lower-level specification.

7.5 Dealing with large specifications

Symbolic execution should support understanding of large specifications. Obviously, the effectiveness of this help depends a lot on the specification itself; if the specification is badly structured, even symbolic execution cannot make it easy to understand.

Symbolic execution of a large specification is probably most effective if the specification is built top-down, with high-level operations being specified in terms of lower-level operations. In this case, symbolic execution can be used in two different ways to validate two different aspects of a specification. First, the user can symbolically execute high-level specifications without first having specified the lower-level ones, or just without unfolding these definitions. This helps her to validate the overall structure of a large specification and is done in a similar way to the validation of incomplete specifications discussed above. Indeed, it would probably often be done while the specification is still incomplete.

However, if the specification is complete, then there will usually be no need for introducing believed proof obligations in the same way as there is for incomplete specifications. Nevertheless, a user might decide to use them anyway since they give her more freedom about the order of doing things by allowing her to use a lemma before proving it.

Once the overall structure of a specification has been validated in this way, its individual components and their interactions can then be validated in a similar way to smaller specifications. Of course, this gets much easier if the specification is well modularised, but even if this is not the case then symbolic execution makes it easier for the user to detect the interactions between different parts of the specification.

In the *SORT*-example discussed in §7.4, validation would thus involve first symbolically executing the overall structure of *SORT* as given in the post-condition, and then dealing with its components *is-permutation* and *is-sorted* (which, in this case, do not have any interaction between them).

7.6 Implicit specifications

One of the two main advances of the work described in this thesis over previous work on symbolic execution is the support for symbolic execution of *implicit* specifications. This led to the introduction of *description values*; if the value of a variable is only defined implicitly, then instead of actual or symbolic values a *formula* is associated with this variable. The whole system SYMBEX is thus geared towards supporting implicit specifications. In our model there is no essential difference between the treatments of implicit and explicit specifications. Explicit specifications often

make certain tasks, in particular simplification, easier to do, but in symbolic execution itself the two are not distinguished.

A standard example of a simple implicit specification is the specification of SORT in §7.4. Once *is-permutation* and *is-sorted* have been specified as well, i.e. the specification is complete, one can execute it on a more detailed level. One possibility that seems particularly useful for implicit specifications is to provide both input *and output* for an operation, in this case SORT. The symbolic execution system should then check whether the values provided satisfy the specification.

Assume the user provides as input value the list $[x_1, x_2, x_3, x_4]$ and as output value the list $[x_4, x_3, x_1, x_5, x_2]$. The system then has to check that

$$\textit{is-permutation}([x_1, x_2, x_3, x_4], [x_4, x_3, x_1, x_5, x_2])$$

and

$$\textit{is-sorted}([x_4, x_3, x_1, x_5, x_2])$$

hold. While the latter might be true, false or undetermined depending on the ordering given, the former is definitely false and should be identified as such, either automatically using the simplifier or with help from the user.

Next, the user starts a new execution and provides input value $[x_1, x_2, x_3, x_4]$, and output value $[x_3, x_1, x_4, x_2]$. Now *is-permutation*(...) should simplify to true, and the user makes some assumptions about the ordering. Assuming $x_4 > x_2$ should make *is-sorted*(...) simplify to false, while assuming $x_4 \leq x_2$ should return the information that in order to simplify *is-sorted*(...) to true and thus show that the specification is satisfied, it is necessary (and sufficient) that $x_3 \leq x_1 \leq x_4$.

Chapter 8

Summary and conclusions

Habe nun, ach! Philosophie,
Juristerei und Medizin
Und leider auch Theologie
Durchaus studiert, mit heißem Bemühn.
Da steh' ich nun, ich armer Tor!
Und bin so klug als wie zuvor;

J. W. v. Goethe: Faust I

Summary and contributions of this thesis

The main contributions of this thesis are

- a formal definition of the denotational semantics of symbolic execution for a wide class of specification and programming languages, expressed in terms of the denotational semantics of the language being executed. This provides a language-generic notion of *correctness* for symbolic execution.
- the development of a *language-generic* tool for symbolic execution which can handle specifications as well as programs. This is based on (a particular version of) the operational semantics of the language being executed.

The thesis describes the symbolic execution system SYMBEX that is to help users to validate specifications by symbolically executing them. SYMBEX is a language-generic tool in the sense that it can be used with any specification language in which software systems are specified in terms of states and state transitions. However, it cannot properly deal with languages that are based on a fundamentally different paradigm, for example algebraic specification languages.

A central part of this thesis and an important step in the development of SYMBEX is the definition of the denotational and operational semantics of symbolic execution, including a discussion

of their relationship. Correctness of symbolic execution under the notion introduced by the denotational semantics is equivalent to the faithfulness of the operational semantics with respect to the denotational semantics.

SYMBEX as described in this thesis forms the nucleus of a symbolic execution system. However, to actually use it for any particular language, it needs to be instantiated for that language. This is done using a description of the operational semantics of the language to be symbolically executed, including the simplification theory. §4.2 explains how to develop such a description and provides the appropriate rules for several language constructs, but does not give the whole operational semantics of any particular language.

Also needed is a theorem proving tool that stores the theories of operational semantics, supports application of these rules, and also supports simplification, both with and without user support. Such a tool is provided by the work on FRIPSE within the IPSE 2.5 project, but other systems with similar functionality would be equally suitable.

The question obviously arises how useful SYMBEX is for its intended purpose, the validation of specifications. Since SYMBEX has not been fully implemented yet, a final answer to this question cannot currently be given. Some answers do, however, emerge from the work done to date and are given in the following.

As mentioned before, we require that, among other things, an operational semantics description and suitable simplification tactics for the specification language used are given. To make symbolic execution genuinely useful, these have to be such that the results of symbolic execution of a specification that the user gets to see are both sufficiently simple and sufficiently different from the specification itself to actually help the user in understanding the specification. For any language that is more than just a toy example, developing such an operational semantics description and such simplification tactics is a very difficult task. Remember for example the very complex rule needed to describe `if-then-else` (see §4.2.7).

This is partly due to the emphasis on language-genericity, which prevents the use of language-specific tricks and shortcuts that could be used in order to get results that are easy to understand. Instead, all language-specific knowledge has to be expressed in terms of the operational semantics and the simplification theories. Of course, an important advantage of this approach is that it enforces a clear structure on the language description, which is therefore less error-prone.

However, it remains questionable whether restricting oneself to any particular specification language would really gain a lot, since most of the problems in understanding the results of symbolic execution arise from the inherent complexity of these results and not from the difficulty of providing language-generic simplification mechanisms. For example, the output from the GIST symbolic executor itself is quite difficult to understand, even though this system only has to support the GIST specification language. However, the work done on GIST also points to a possible solution to the problem of results that are too difficult to understand, in form of the *paraphraser* and the *behavior explainer* (as discussed in §2.1.2). The development of such support tools for SYMBEX is therefore one of the tasks suggested for future work. While in GIST these tools were provided for one particular language, in IPSE 2.5 or SYMBEX they can hopefully be provided as language-generic tools working on a description of syntax and semantics of a wide class of languages. Note in this context that the paraphraser and the behavior explainer are different tools, acting

on different languages — the paraphraser handles specifications, while the behavior explainer handles the output from symbolic execution.

Overall one can say that SYMBEX is a useful tool for *interactive exploration* of specifications, even though it does not quite provide as much automatic feedback on a specification as had been hoped for at the outset of this work. This implies that it depends a lot on the user whether or not symbolic execution provides her with useful feedback. It will therefore probably not be quite as useful as had been hoped for for demonstrating what has been specified to a customer with little or no knowledge of the specification language used. This will probably change if a tool like the behavior explainer does become available for SYMBEX. For the specifier, however, it provides an opportunity to explore the specification and its implications to an extent that has, until now, not been possible. This opportunity is available early on in the specification process, long before the specification has been completed, since the description values resulting from symbolic execution may be expressed in terms of the *names* of functions used rather than their *definitions*. Obviously, one will not be able to unfold a function name until that function has been defined — one can only validate those parts of the specification that have been written at any point in time. Instead, symbolic execution allows one to explore and validate the overall structure of the specification as soon as this overall structure has been written and before further work is based on it to fill in all the details.

Comparison with other work

Of all the symbolic execution system discussed in §2.1.2, GIST is the one whose approach is by far the closest to that of SYMBEX. This is mainly due to the fact that both systems support symbolic execution of *specifications*, as opposed to programs, and that the main aim of both systems is the validation of specifications.

There are two main reasons for the differences between the two systems: the first is the fact that GIST supports a fixed specification language, while SYMBEX is generic over a whole class of languages. The second is the fact that SYMBEX aims for a complete description of the results of actual execution (even though this might not always be achieved — weak symbolic execution), while in GIST the results of symbolic execution are *consequences* from such a complete description, GIST from the outset only aims for *weak* symbolic execution (in the sense defined in Figure 4.2).

In comparison with other symbolic execution systems such as EFFIGY or DISSECT, SYMBEX obviously differs a lot because of the extension of symbolic execution to specifications and the genericity of SYMBEX. As a result, the symbolic execution technique used by SYMBEX is *very* different from that used in other systems, indeed at the syntactic level there is little resemblance. The reason for using the same name "symbolic execution" for the new technique is that at the semantic level, SYMBEX provides a straightforward extension of what was done by these other systems. All the symbolic execution systems described have the same denotational semantics, they differ mainly in the languages supported and in the additional facilities provided, such as test case generation or some form of verification support. Another difference between the various systems is the result language permitted, the language for expressing the results of symbolic execution.

Most systems only allow the language of polynomials over input variables, plus a few minor extensions. This leads to obvious problems in expressing the results of symbolic execution of various statements, in particular `while`-loops. None of the systems, with the exception of Harvard PDS, is able to support the full `while`-statement. Instead, they all make various assumptions about the loop, be it by insisting that the body of the loop gives rise to a solvable recurrence relation (e.g. ATTEST), or indirectly by restricting the number of iterations of the loop (e.g. EFFIGY). The approach taken by both Harvard PDS and SYMBEX is to use a considerably more expressive result language by introducing a particular kind of λ-expressions (Harvard PDS, cf. §2.1.2) or description values (SYMBEX).

The extended result language of description values as used in SYMBEX means that a much wider class of language constructs can be symbolically executed (as obvious from the fact that SYMBEX supports *specification* languages). The drawback of this extension is that the results are potentially much "weaker" in the sense that they may not provide as much of a new perspective on the specification (or program) as might be the case with a more conventional result language. Whether this is the case in symbolic execution of any particular specification again depends very much on the simplification of the result. As an example, consider symbolic execution of some small imperative programming language as supported by any of the more conventional systems (cf. §4.3). Using SYMBEX to symbolically execute any program in this language, one would at first get a collection of description values of the form $x_{i+1} = f(x_i)$. To get the same results as one would get from any of the other systems, one needs to apply a simplification tactic that replaces x_{i+1} by $f(x_i)$ in all future description values. The end result would then be, for each variable x, the description value $x_n = g(x_1, y_1, \ldots)$, while using any of the other systems x would finally have the symbolic value $g(x_1, y_1, \ldots)$.

Suggestions for future work

Finally, a short summary of the issues that remain to be solved and are suggested as topics for future work: first of all, this includes the description of the full operational semantics and its simplification theory for some specific formal specification language. This work will be able to build on §4.2 of this thesis, but obviously a lot remains to be done.

A second suggestion for future work that has been mentioned before is the development of a "paraphraser" and a "behavior explainer" (GIST terminology). Like SYMBEX, these should be generic with respect to the language supported, but an instantiation for some particular specification language should of course also be developed. Obviously, the language selected should preferably be one for which a description of its operational semantics and simplification theory already exists.

The work in this thesis has been restricted to specification languages that are based on the notions of states and state transformations. The question remains whether it can be extended to a wider class of languages, and, if so, how this could be done. This thesis only briefly touched on this question in §4.3.

A useful extension to SYMBEX would be to combine the approach to symbolic execution taken here with that taken in GIST and add a tool similar to the FIE (Forward Inference Engine, cf.

§2.1.2) which would automatically derive certain "interesting" inferences of the description values given. These could then either be added to the description values of the relevant identifiers, or just displayed to the user. Of course, there is the dual problem of *generating* interesting consequences as well as trivial ones, and then of selecting these interesting ones. Overcoming this problem would probably need a major project in its own right, which is why the current version of SYMBEX does not include such an extension.

Another useful extension would be to include some form of version control system to allow for changes in the specifications in a *Module* without losing the information gained from symbolic execution in that *Module*. As described on page 115, such a tool would update the *SEStateOps* to take account of changes to a specification. Those parts of the *SEStateOp* that are not affected by the change would remain unchanged, while those parts that are affected would be changed accordingly, or at least marked as 'unsafe'. Since SYMBEX is intended to help a user to check a specification, the results of symbolic execution will often lead to changes of the specification. Obviously, the user will then not want to lose any more than necessary of the validation work already done.

Appendix A

Proofs of theorems and lemmas

> Wir fühlen, dass selbst, wenn alle *möglichen* wissenschaftlichen Fragen beantwortet sind, unsere Lebensprobleme noch gar nicht berührt sind. Freilich bleibt dann eben keine Frage mehr; und eben dies ist die Antwort.
>
> *Ludwig Wittgenstein:*
> *Tractatus Logico-Philosophicus*

A.1 Proof of Lemma 4.1.7

For all τ: *SEStateDen*

$$
\begin{aligned}
&\mathit{yield}(\mathit{symbolic\text{-}ex}[\![\mathit{spec}_1; \mathit{spec}_2]\!]\tau) \\
&= \lambda\sigma \cdot \{\sigma_1 \mid \exists \sigma\text{-}\mathit{seq} \in \mathit{SEQS}(\mathit{symbolic\text{-}ex}[\![\mathit{spec}_1; \mathit{spec}_2]\!]\tau) \cdot \\
&\qquad \mathrm{hd}\ \sigma\text{-}\mathit{seq} = \sigma \wedge \mathrm{last}\ \sigma\text{-}\mathit{seq} = \sigma_1 \\
&\qquad \wedge \mathrm{len}\ \sigma\text{-}\mathit{seq} = \mathit{LEN}(\mathit{symbolic\text{-}ex}[\![\mathit{spec}_1; \mathit{spec}_2]\!]\tau)\} \\
&= \lambda\sigma \cdot \{\sigma_1 \mid \exists \sigma\text{-}\mathit{seq} \cdot \mathrm{front}\ \sigma\text{-}\mathit{seq} \in \mathit{SEQS}(\tau) \\
&\qquad \wedge \mathrm{len}\ \sigma\text{-}\mathit{seq} = \mathit{LEN}(\tau) + 1 \wedge \mathrm{last}\ \sigma\text{-}\mathit{seq} = \sigma_1 \wedge \mathrm{hd}\ \sigma\text{-}\mathit{seq} = \sigma \\
&\qquad \wedge \mathcal{M}_{\mathit{Spec}}[\![\mathit{spec}_1; \mathit{spec}_2]\!](\mathrm{last\,front}\ \sigma\text{-}\mathit{seq}, \mathrm{last}\ \sigma\text{-}\mathit{seq})\} \\
&= \lambda\sigma \cdot \{\sigma_1 \mid \exists \sigma\text{-}\mathit{seq} \cdot \mathrm{front}\ \sigma\text{-}\mathit{seq} \in \mathit{SEQS}(\tau) \\
&\qquad \wedge \mathrm{len}\ \sigma\text{-}\mathit{seq} = \mathit{LEN}(\tau) + 1 \wedge \mathrm{last}\ \sigma\text{-}\mathit{seq} = \sigma_1 \wedge \mathrm{hd}\ \sigma\text{-}\mathit{seq} = \sigma \\
&\qquad \wedge \exists \sigma_2 \cdot \mathcal{M}_{\mathit{Spec}}[\![\mathit{spec}_1]\!](\mathrm{last\,front}\ \sigma\text{-}\mathit{seq}, \sigma_2) \wedge \mathcal{M}_{\mathit{Spec}}[\![\mathit{spec}_2]\!](\sigma_2, \mathrm{last}\ \sigma\text{-}\mathit{seq})\} \\
&= \lambda\sigma \cdot \{\sigma_1 \mid \exists \sigma\text{-}\mathit{seq}' \cdot \mathrm{front\,front}\ \sigma\text{-}\mathit{seq}' \in \mathit{SEQS}(\tau) \\
&\qquad \wedge \mathrm{len}\ \sigma\text{-}\mathit{seq}' = \mathit{LEN}(\tau) + 2 \wedge \mathrm{last}\ \sigma\text{-}\mathit{seq}' = \sigma_1 \wedge \mathrm{hd}\ \sigma\text{-}\mathit{seq}' = \sigma \\
&\qquad \wedge \mathcal{M}_{\mathit{Spec}}[\![\mathit{spec}_1]\!](\mathrm{last\,front\,front}\ \sigma\text{-}\mathit{seq}', \mathrm{last\,front}\ \sigma\text{-}\mathit{seq}') \\
&\qquad \wedge \mathcal{M}_{\mathit{Spec}}[\![\mathit{spec}_2]\!](\mathrm{last\,front}\ \sigma\text{-}\mathit{seq}', \mathrm{last}\ \sigma\text{-}\mathit{seq}')\}
\end{aligned}
$$

$$
\begin{aligned}
= \ & \lambda\sigma \cdot \{\sigma_1 \mid \exists \sigma\text{-}seq' \cdot \text{front } \sigma\text{-}seq' \in SEQS(symbolic\text{-}ex[\![spec_1]\!]\tau) \\
& \quad \wedge \text{len } \sigma\text{-}seq' = LEN(symbolic\text{-}ex[\![spec_1]\!]\tau) + 1 \\
& \quad \wedge \text{last } \sigma\text{-}seq' = \sigma_1 \wedge \text{hd } \sigma\text{-}seq' = \sigma \\
& \quad \wedge \mathcal{M}_{Spec}[\![spec_2]\!](\text{last front } \sigma\text{-}seq', \text{last } \sigma\text{-}seq')\} \\
= \ & \lambda\sigma \cdot \{\sigma_1 \mid \exists \sigma\text{-}seq' \cdot \sigma\text{-}seq' \in SEQS(symbolic\text{-}ex\text{-}s[\![[spec_1, spec_2]]\!]\tau) \\
& \quad \wedge \text{len } \sigma\text{-}seq' = LEN(symbolic\text{-}ex\text{-}s[\![[spec_1, spec_2]]\!]\tau) \\
& \quad \wedge \text{last } \sigma\text{-}seq' = \sigma_1 \wedge \text{hd } \sigma\text{-}seq' = \sigma\} \\
= \ & yield(symbolic\text{-}ex\text{-}s[\![[spec_1, spec_2]]\!]\tau)
\end{aligned}
$$

A.2 Proof of Theorem 4.2.2

We show, by induction on len $SEQ(S)$, that

$$\forall S: SEStateOp \cdot inv\text{-}SEStateDen(\mathcal{M}_{SEStateOp}[\![S]\!])$$

Base case: len $SEQ(S) = 1$ We have to show that

$$
\begin{aligned}
& \text{let } set = \{[\sigma] \mid satisfies\text{-}all\text{-}restrictions([\sigma], S, 1)\} \text{ in} \\
& \forall \sigma\text{-}seq \in set \cdot \text{len } \sigma\text{-}seq \leq 1 \\
& \wedge \forall \sigma\text{-}seq_1, \sigma\text{-}seq_2 \in set \cdot \forall \sigma\text{-}seq: \text{seq of } \Sigma_\perp \cdot \\
& \qquad \sigma\text{-}seq_1 = \sigma\text{-}seq_2 \frown \sigma\text{-}seq \ \Rightarrow \ \sigma\text{-}seq = [\,]
\end{aligned}
$$

which is trivially true.

Induction step: Now assume that, for some S, $inv\text{-}SEStateDen(\mathcal{M}_{SEStateOp}[\![S]\!])$ and consider $S' = mk\text{-}SEStateOp(SEQ(S) \frown e, INDEX(S))$, for some e. We first have to show that

$$\forall \sigma\text{-}seq \in SEQS(\mathcal{M}_{SEStateOp}[\![S']\!]) \cdot \text{len } \sigma\text{-}seq \leq LEN(\mathcal{M}_{SEStateOp}[\![S']\!])$$

This follows immediately from the definition of $\mathcal{M}_{SEStateOp}[\![S']\!]$.

For the second part of the proof assume that

$$\sigma\text{-}seq_1, \sigma\text{-}seq_2 \in SEQS(\mathcal{M}_{SEStateOp}[\![S']\!])$$

and that for some $\sigma\text{-}seq$: seq of $\Sigma_\perp$

$$\sigma\text{-}seq_1 = \sigma\text{-}seq_2 \frown \sigma\text{-}seq$$

We distinguish three cases:

Case 1: len $\sigma\text{-}seq_2 = LEN(\mathcal{M}_{SEStateOp}[\![S']\!])$

In this case $\sigma\text{-}seq = [\,]$ follows immediately, since there are no sequences in $SEQS(\mathcal{M}_{SEStateOp}[\![S']\!])$ that are longer than $LEN(\mathcal{M}_{SEStateOp}[\![S']\!])$.

Case 2: len $\sigma\text{-}seq_2 = LEN(\mathcal{M}_{SEStateOp}[\![S']\!]) - 1$

Then, by definition of $\mathcal{M}_{SEStateOp}$,

$$\neg\exists \sigma: \Sigma_\perp \cdot satisfies\text{-}all\text{-}restrictions(\sigma\text{-}seq_2 \frown \sigma, S', \text{len } SEQ(S'))$$

therefore σ-seq cannot have length 1. It cannot be longer either, since then σ-seq_1 would be too long to be in $SEQS(\mathcal{M}_{SEStateOp}[\![S']\!])$. This only leaves σ-seq = [ ], as required.

Case 3: len σ-$seq_2 < LEN(\mathcal{M}_{SEStateOp}[\![S']\!]) - 1$

In this case σ-$seq_2 \in SEQS(\mathcal{M}_{SEStateOp}[\![S]\!])$, and we have to distinguish two further cases:

Case 3.1: σ-$seq_1 \in SEQS(\mathcal{M}_{SEStateOp}[\![S]\!])$

Then σ-seq = [ ] follows by induction hypothesis.

Case 3.2: σ-$seq_1 \notin SEQS(\mathcal{M}_{SEStateOp}[\![S]\!])$

In this case, since σ-$seq_1 \in SEQS(\mathcal{M}_{SEStateOp}[\![S']\!])$

front σ-$seq_1 \in SEQS(\mathcal{M}_{SEStateOp}[\![S]\!]) \wedge$ len σ-$seq_1 = LEN(\mathcal{M}_{SEStateOp}[\![S]\!]) + 1$

Now assume σ-$seq \neq$ [ ]. Then front σ-$seq_1 = \sigma$-$seq_2 \frown$ front σ-seq, and one can apply the induction hypothesis to get front σ-seq = [ ], or len σ-seq = 1. Then

$$\begin{aligned} \text{len } \sigma\text{-}seq_1 &= \text{len } \sigma\text{-}seq_2 + \text{len } \sigma\text{-}seq \\ &< (LEN(\mathcal{M}_{SEStateOp}[\![S']\!]) - 1) + 1 \\ &= \text{len } \sigma\text{-}seq_1 \end{aligned}$$

which shows that our assumption σ-$seq \neq$ [ ] must have been false.

A.3 Proof of Theorem 4.2.10

Note that len $SEQ(S) \frown m \geq 2$. Let sm: $SpecMap$ be given, let $Seq = SEQ(S)$ and let

$S' = add\text{-}to\text{-}SEStateOp(S, m)$

We have to show that $\mathcal{M}_{Conf}[\![\langle [Op], S\rangle]\!] = \mathcal{M}_{Conf}[\![\langle [\,], S'\rangle]\!]$. Obviously

$LEN(\mathcal{M}_{Conf}[\![\langle [Op], S\rangle]\!]) = LEN(\mathcal{M}_{Conf}[\![\langle [\,], S'\rangle]\!])$

since

$\mathcal{M}_{Conf}[\![\langle [Op], S\rangle]\!] = symbolic\text{-}ex[\![sm(Op)]\!](\mathcal{M}_{SEStateOp}[\![S]\!])$

and

$\mathcal{M}_{Conf}[\![\langle [\,], S'\rangle]\!] = \mathcal{M}_{SEStateOp}[\![S']\!]$

We now show

$$SEQS(\mathcal{M}_{Conf}[\![\langle [\,], S'\rangle]\!]) \subseteq SEQS(\mathcal{M}_{Conf}[\![\langle [Op], S\rangle]\!]) \tag{A.1}$$

Assume σ-$seq \in SEQS(\mathcal{M}_{Conf}[\![\langle [\,], S'\rangle]\!])$. Then by definition of $\mathcal{M}_{Conf}$

$$
\begin{aligned}
&\textit{satisfies-all-restrictions}(\sigma\text{-}seq, S', \text{len}\,Seq + 1)\\
&\wedge\ [\text{front}\ \sigma\text{-}seq \in SEQS(\mathcal{M}_{SEStateOp}[\![S]\!])\\
&\quad\wedge\ \text{len}\ \sigma\text{-}seq = \text{len}\,Seq + 1\\
&\quad\vee\ \sigma\text{-}seq \in SEQS(\mathcal{M}_{SEStateOp}[\![S]\!])\\
&\quad\wedge\ \text{len}\ \sigma\text{-}seq = \text{len}\,Seq\\
&\quad\wedge\ \neg\exists\sigma \cdot \textit{satisfies-all-restrictions}(\sigma\text{-}seq \frown \sigma, S', \text{len}\,Seq + 1)\\
&\quad\vee\ \sigma\text{-}seq \in SEQS(\mathcal{M}_{SEStateOp}[\![S]\!])\\
&\quad\wedge\ \text{len}\ \sigma\text{-}seq < \text{len}\,Seq]
\end{aligned}
$$

and we distinguish three cases:

Case 1: $\text{len}\ \sigma\text{-}seq = \text{len}\,Seq + 1$

Because of $\textit{satisfies-all-restrictions}(\sigma\text{-}seq, S', \text{len}\,Seq + 1)$ and in particular $\textit{satisfies-restrictions}(\sigma\text{-}seq, S', \text{len}\,Seq + 1)$ we know

$$
\begin{aligned}
&\psi_1(\sigma\text{-}seq[\text{len}\,Seq + 1])\\
&\wedge\ \psi_2(\sigma\text{-}seq[\text{len}\,Seq], \sigma\text{-}seq[\text{len}\,Seq + 1])\\
&\wedge\ \psi_3(\sigma\text{-}seq[\text{len}\,Seq + 1])\\
&\wedge\ \psi_4(\sigma\text{-}seq[\text{len}\,Seq], \sigma\text{-}seq[\text{len}\,Seq + 1])\\
&\wedge\ \psi_5(\sigma\text{-}seq[\text{len}\,Seq + 1])\\
&\wedge\ \psi_6(\sigma\text{-}seq[\text{len}\,Seq], \sigma\text{-}seq[\text{len}\,Seq + 1])
\end{aligned}
$$

where the ψ_i are the predicates used in defining m. Now for all σ_1, σ_2

$$\psi_2(\sigma_1, \sigma_2) \wedge \psi_4(\sigma_1, \sigma_2) \wedge \psi_6(\sigma_1, \sigma_2) \iff \mathcal{M}_{Spec}[\![sm(Op)]\!](\sigma_1, \sigma_2) \qquad \text{(A.2)}$$

which implies that $\sigma\text{-}seq \in SEQS(\textit{symbolic-ex}[\![sm(Op)]\!](\mathcal{M}_{SEStateOp}[\![S]\!]))$. Note that $\Rightarrow$ of (A.2) would not hold if $\text{len}\ \sigma\text{-}seq < \text{len}\,Seq + 1$.

Case 2: $\text{len}\ \sigma\text{-}seq = \text{len}\,Seq$

In this case

$$
\begin{aligned}
&\sigma\text{-}seq \in SEQS(\mathcal{M}_{SEStateOp}[\![S]\!])\\
&\wedge\ \neg\exists\sigma \cdot \textit{satisfies-all-restrictions}(\sigma\text{-}seq \frown \sigma, S', \text{len}\,Seq + 1)\\
&\wedge\ \textit{satisfies-all-restrictions}(\sigma\text{-}seq, S', \text{len}\,Seq + 1)
\end{aligned}
$$

Because of the well-behaviour of S (which implies well-behaviour of S'), the last two conjuncts imply that

$$\neg\exists\sigma \cdot \textit{satisfies-restrictions}(\sigma\text{-}seq \frown \sigma, S', \text{len}\,Seq + 1)$$

Again because of well-behaviour, this implies

$$\neg\exists\sigma \cdot \bigwedge_n \bigwedge_{m(n)} \textit{satisfies-restriction}(\sigma\text{-}seq \frown \sigma, ps, \textit{INDEX}(S))$$

This is a contradiction, since (according to Definition 3.1.1) for every input state σ' (in particular last $\sigma\text{-}seq$) there exists an output state σ s.t. $\mathcal{M}_{Spec}[\![Op]\!](\sigma', \sigma)$.

Case 3: len σ-*seq* < len *Seq*

In this case σ-*seq* $\in$ *SEQS*($\mathcal{M}_{SEStateOp}[\![S]\!]$) and

σ-*seq* $\in$ *SEQS*(*symbolic-ex*[$\![sm(Op)]\!$]($\mathcal{M}_{SEStateOp}[\![S]\!]$)

follows immediately from the definition of *symbolic-ex*.

We have thus shown that (A.1) holds. For the converse, let σ-*seq* be in *SEQS*(*symbolic-ex*[$\![sm(Op)]\!$]($\mathcal{M}_{SEStateOp}[\![S]\!]$)).

Case 1: len σ-*seq* = len *Seq* + 1

Then

front σ-*seq* $\in$ *SEQS*($\mathcal{M}_{SEStateOp}[\![S]\!]$)
$\wedge\ \mathcal{M}_{Spec}[\![sm(Op)]\!](\sigma\text{-}seq[\text{len}\,Seq], \sigma\text{-}seq[\text{len}\,Seq + 1])$

The first part together with Lemma 4.2.4 (well-behaviour) implies that

satisfies-all-restrictions(σ-*seq*, *S*, len *Seq*)

while the second together with (A.2) implies that

satisfies-restrictions(σ-*seq*, *S'*, len *Seq* + 1)

Together these imply

satisfies-all-restrictions(σ-*seq*, *S'*, len *Seq* + 1)

and therefore σ-*seq* $\in$ *SEQS*($\mathcal{M}_{Conf}[\![\langle [\,], S'\rangle]\!]$), as required.

Case 2: len σ-*seq* < len *Seq*

In this case we have

σ-*seq* $\in$ *SEQS*($\mathcal{M}_{SEStateOp}[\![S]\!]$)

and therefore

satisfies-all-restrictions(σ-*seq*, *S*, len *Seq*)

The definition of *m* together with the definition of *satifies-all-restrictions* now implies

satisfies-all-restrictions(σ-*seq*, *S'*, len *Seq* + 1)

and again we get σ-*seq* $\in$ *SEQS*($\mathcal{M}_{Conf}[\![\langle [\,], S'\rangle]\!]$), as required.

A.4 Proof of Lemma 6.1.1

Let S be any *SEStateOp*. We first show that

$$\textit{pre-get-element}(S, \textit{current-index}(S))$$

$\textit{current-index}(S) \neq [\,]$ is trivially true by definition of *current-index*. The second part of the pre-condition can be shown by induction over len front $\textit{current-index}(S)$.

Now we have to show that

$$\textit{get-element}(S, \textit{current-index}(S)) : \textit{SE-map}$$

Again, this is done by induction over len front $\textit{current-index}(S)$:
Base case: If len front $\textit{current-index}(S) = 0$ then

$$\text{last}\,\textit{SEQ}(S) : \textit{SE-map}$$

by definition of *current-index* and

$$\begin{aligned} &\textit{get-element}(S, \textit{current-index}(S)) \\ &= \textit{SEQ}(S)[\text{last}\,\textit{current-index}(S)] \\ &= \textit{SEQ}(S)[\text{len}\,\textit{SEQ}(S)] \\ &= \text{last}\,\textit{SEQ}(S) \end{aligned}$$

Induction step: len front $\textit{current-index}(S) > 0$

$$\begin{aligned} &\textit{get-element}(S, \textit{current-index}(S)) \\ &= \textit{get-element}(\textit{SEQ}(S)[\text{last}\,\textit{current-index}(S)], \text{front}\,\textit{current-index}(S)) \\ &= \textit{get-element}(\text{last}\,\textit{SEQ}(S), \text{front}\,\textit{current-index}(S)) \end{aligned}$$

Since front $\textit{current-index}(S) = \textit{current-index}(\text{last}\,\textit{SEQ}(S))$ we can now apply the induction hypothesis to get that

$$\textit{get-element}(S, \textit{current-index}(S)) : \textit{SE-map}$$

as required.

Appendix B

Short summary of VDM

> ...it is now a well-established phenomenon that what is highly abstract for a generation of mathematicians is just commonplace for the next one.
> *J. Dieudonné*

B.1 VDM-notation

The following is a (very) short summary of some VDM-notation that is used in this thesis and which is not standard mathematical notation. Compare also the *Glossary of symbols*, which lists some more notation used. The version of VDM used here is that given in [Jon86]. Note that here I only consider VDM as a *specification language* and ignore the fact that VDM really is a *methodology* for software development.

VDM is based on the notions of states and state transformations called operations. It supports a number of primitive data types such as finite sets, maps and sequences, and also allows product types and defined types. A type definition takes the form

type-name = *type-expression*

where *type-name* is defined to be *type-expression*. Additionally one can provide a type invariant in order to define sub-types:

type-name = *type-expression*

where

inv-type-name(*t*) $\triangleq$ ...

Definitions of record types take the form

type-name :: *field-name*1 : *field-type*1
*field-name*2 : *field-type*2
... : ...

Operations that access the state of a system are specified in the form

OP $(a{:}\,T_1)\ r{:}\,T_2$
ext rd er : T_3
wr ew : T_4
pre $\varphi(a, er, ew)$
post $\psi(a, \overleftarrow{ew}, r, er, ew)$

Here $a{:}\,T_1$ denotes the arguments of the operation, $r{:}\,T_2$ denotes the result, $er{:}\,T_3$ the external or state variables to which the operation has got read access, and $ew{:}\,T_3$ the external or state variables to which the operation has got write access. φ denotes a pre-condition, ψ a post-condition. The semantics of OP is defined as: if the pre-condition holds before the operation, then the post-condition will hold afterwards. Since the operation may have changed the state, the post-condition refers both to the values before ($\overleftarrow{ew}$) and after (ew) the operation.

All the parameters in the definition of an operation are optional. In particular, this notation can be used for implicit function definitions, which do not have external read and write variables.

Explicit function definitions take the form

$f : T_1 \to T_2$
$f(x) \triangleq body(x)$
pre $\varphi(x)$

A sequence seq is identified with the map $\{i \mapsto seq[i] \mid i \leq \mathsf{len}\, seq\}$.

B.2 LPF — The Logic of Partial Functions

The logic used for VDM is LPF, the Logic of Partial Functions [Jon86, BCJ84]. LPF is a three-valued logic that allows handling of undefinedness. It is an extension of classical logic, and in some sense a 'generous' extension in that it always tries to assign a value to an expression, even if components of the expression are themselves undefined. For example, false $\wedge \perp$ is defined to be false rather than $\perp$. Similarly, true $\vee \perp$ is defined to be true.

B.3 Some auxiliary functions

The following functions are based on the BSI-Protostandard for VDM [BSI88, And88]. Given a sequence of *NameTypePairs*, the following functions extract the appropriate sequences of *Names* and *Types*:

names : seq of *NameTypePair* → seq of *Name*

names(*nt-seq*) △ if *nt-seq* = [ ]
then [ ]
else cons(*name*(hd *nt-seq*), *names*(tl *nt-seq*))

types : seq of *NameTypePair* → seq of *Type*

types(*nt-seq*) △ if *nt-seq* = [ ]
then [ ]
else cons(*type*(hd *nt-seq*), *types*(tl *nt-seq*))

Given a sequence of *ExtVarInf*s, the following functions extract the appropriate sequences of *Name*s and *Type*s from it, separately for variables with read- and write-access:

readnames : seq of *ExtVarInf* → seq of *Name*

readnames(*evi-seq*) △
if *evi-seq* = [ ]
then [ ]
else if *mode*(hd *evi-seq*) = READ
then cons(*name*(*rest*(hd *evi-seq*)), *readnames*(tl *evi-seq*))
else *readnames*(tl *evi-seq*)

readwritenames : seq of *ExtVarInf* → seq of *Name*

readwritenames(*evi-seq*) △
if *evi-seq* = [ ]
then [ ]
else if *mode*(hd *evi-seq*) = READWRITE
then cons(*name*(*rest*(hd *evi-seq*)), *readwritenames*(tl *evi-seq*))
else *readwritenames*(tl *evi-seq*)

readtypes : seq of *ExtVarInf* → seq of *Type*

readtypes(*evi-seq*) △
if *evi-seq* = [ ]
then [ ]
else if *mode*(hd *evi-seq*) = READ
then cons(*type*(*rest*(hd *evi-seq*)), *readtypes*(tl *evi-seq*))
else *readtypes*(tl *evi-seq*)

readwritetypes : seq of *ExtVarInf* → seq of *Type*

readwritetypes(*evi-seq*) ≜
  if *evi-seq* = [ ]
  then [ ]
  else if *mode*(hd *evi-seq*) = READWRITE
    then cons(*type*(*rest*(hd *evi-seq*)), *readwritetypes*(tl *evi-seq*))
    else *readwritetypes*(tl *evi-seq*)

The following functions extract the invariant on a *TypeDef*, *Name* (of a *TypeDef*) or *Type* in a module, if any; if it does not have an invariant, the functions return true.

inv_of_TypeDef : *TypeDef* × *Module* → *ExplFnDef*

inv_of_TypeDef(*t*, *m*) ≜ if *inv*(*TypeDef*) ≠ nil
  then *inv*(*TypeDef*)
  else *mk-ExplFnDef*(nil, nil, [ ], true)

inv_of_Name : *Name* × *Module* → *ExplFnDef*

inv_of_Name(*n*, *m*) ≜ *inv_of_TypeDef*(*typem*(*body*(*m*))(*n*), *m*)

pre *n* ∈ dom *typem*(*body*(*m*))
  ∧ *typem*(*body*(*m*))(*n*) ≠ nil

inv_of_Type : *Type* × *Module* → *ExplFnDef*

inv_of_Type(*t*, *m*) ≜ if ∃*n*: *Name* · *t* = *mk-TypeName*(*n*)
  ∧ *n* ∈ dom *typem*(*body*(*m*))
  ∧ *typem*(*body*(*m*))(*n*) ≠ nil
  then let *t* = *mk-TypeName*(*n*) in
    inv_of_Name(*n*, *m*)
  else *mk-ExplFnDef*(nil, nil, [ ], true)

Appendix C

Extracts from the FRIPSE specification

Mein teure Freund, ich rat Euch drum
Zuerst Collegium Logicum. Da wird der Geist Euch wohl dressiert,
In Spanische Stiefel eingeschnürt,
Daß er bedächtiger so fortan
Hinschleiche die Gedankenbahn
Und nicht etwa, die Kreuz und Quer,
Irrlichteliere hin und her.

J. W. v. Goethe: Faust I

This section does not contain any work done by the author of this thesis, and is included only for ease of reference. The following are excerpts from [LM88], and section numbers refer to that document.

Primitives (§1.1)

Apart from the standard VDM primitive types, the primitive types used in this spec include the following:

- Object-level "atomic" symbols: *CESymb* (for primitive constants and functions), *CTSymb* (primitive types and type-constructors), ...
- *VSymb* (for variables), *PESymb* (pattern expression symbols or metavariables), *PTSymb* (pattern type symbols or metavariables)
- Other names: *Rule-ref*, *Theory-ref*, *ThMorph-ref*

These primitive types are assumed to be mutually disjoint, infinite sets of tokens.

Assertions (§1.2)

There are three basic kinds of *assertion* (or *judgement*), although we shall add a fourth to represent "undecided":

Assertion = *LogicalAssertion* | *TypeAssertion*
| *TypingAssertion* | *NullAssertion*

The most common kind of assertion is that a formula *holds*:

LogicalAssertion :: *EXP* : *Exp*

The next most common kind of assertion is that an object is of a certain type:

TypingAssertion :: *EXP* : *Exp*
TYPE : *Type*

where

inv-TypingAssertion(*mk-TypingAssertion*(*e*, *t*)) $\triangleq$ ...

The third kind of assertion is that a type is *well-formed*:

TypeAssertion :: *TYPE* : *Type*

Instantiations (§3)

Instantiation consists of replacing pattern expression symbols with expressions and pattern type symbols with types.

Instantiation :: *PEMAP* : map *PESymb* to *Exp*
PTMAP : map *PTSymb* to *Type*

A family *Instantiate* of functions is defined, which applies an instantiation to some object of type *T*. Among the types *T* for which *Instantiate* has been defined are *Assertion* and *Sequent* (see below).

Signatures and definitions (§4)

Atoms are declared or defined in a *signature*:

Signature :: *PCE* : map *CESymb* to $\mathbb{N} \times \mathbb{N}$
PB : set of *BSymb*
PCT : map *CTSymb* to $\mathbb{N} \times \mathbb{N}$
DEFS : *Definitions*

Definitions include definitions of constants, types and binders. Note that recursive definitions will be allowed.

Sequents and Rules (§5)

Sequent :: *PREMISES* : set of *Assertion*
UPSHOT : *Assertion*

Rulemap = map *Rule-ref* to *Rule*

Rule :: *STMT* : *RuleStmt*
THEORY : *Theory-ref*
PROOF : [*Proof*]

I've assumed that each rule has a single proof so that the circularity check is straightforward. Of course, different rules can have the same statement, so we can still support multiple proofs. Rules with null *PROOF*-field are called *axioms*; the rest are called *derived rules*. It's important not to confuse an axiom with a derived rule having an "empty" proof, ...

RuleStmt :: *SEQHYPS* : set of *Sequent*
ORDHYPS : set of *Assertion*
CONCL : *Assertion*

The function *Establishes* checks whether one *RuleStmt* is established by another (allowing for renaming of variables etc).

Theories (§6)

Theory :: *PARENTS* : set of *Theory-ref*
EXSIG : *Signature*

Theorymap = map *Theory-ref* to *Theory*

where

inv-Theorymap(m) $\triangleq$ ...

Proofs (§8)

Is-Complete-Proof checks whether a proof conducted in a given theory is complete, i.e. whether it is finished and all the lines used to establish the conclusion have complete and reasonable assertions and complete justifications (if appropriate).

The Store (§9)

Store :: *RULES* : *Rulemap*
THS : *Theorymap*
THMORPHS : *ThMorphmap*

Glossary of symbols

Note that symbols which are defined in this thesis are usually only included in the index and not in this glossary.

Basic types

$\mathbb{B}$	Booleans
$\mathbb{N}$	Natural numbers (including 0)
$\mathbb{N}_1$	Natural numbers (starting from 1)
$\mathbb{Z}$	Integers

More on types

inv-T	invariant on T, creates subtype
mk-T	constructor function for T
$T::\ldots$	record construction
$T_1 \mid T_2$	type union
$T_1 \times T_2$	type product
$T_\perp$	$T \cup \{\perp\}$

Sequences

seq of T	type of finite sequences with elements from T
[]	empty sequence
$[e_1, \ldots, e_n]$	sequence enumeration
hd s	head of sequence s
tl s	tail of sequence s
rev s	reverse of sequence s
front s	rev $\circ$ tl $\circ$ rev s
last s	hd $\circ$ rev s
len s	length of a sequence s

$s_1 \frown s_2$	concatenation of two sequences
cons(e, s)	adding an element e at beginning of sequence s
$s \curvearrowright e$	adding an element e at end of sequence s
$s[i]$	selecting the i-th element of sequence s
$s(i, \ldots, j)$	subsequence of s

Note that this differs slightly from VDM as described in [Jon86], where $\curvearrowright$ is used instead of $\frown$, and $s(i)$ instead of $s[i]$. The problem with the latter is that VDM concrete syntax (as described in [Jon86]) does not distinguish between an element of a sequence and a subsequence of length 1.

Sets

set of T	type of finite sets with elements from T
$\{\}$	empty set
$\{e_1, \ldots, e_n\}$	set enumeration
$\{e \mid P(e)\}$	set comprehension

Maps

map T_1 to T_2	maps (functions with finite domain) from T_1 to T_2
$m_1 \dagger m_2$	map overwrite
$\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$	map enumeration
$\{a \mapsto b \mid P(a, b)\}$	map comprehension
$s \lhd m$	domain deletion: $\{d \mapsto m(d) \mid d \in (\mathrm{dom}\, m - \{s\})\}$

Others

let ... in	local variable declaration
$\perp$	bottom element
$\bot$	bottom state, denotes abortion or non-termination
pre-fn	pre-condition of function *fn*
pre-op	pre-condition of operation *op*
post-op	post-condition of operation *op*
$T[t_1/t_2]$	in T substitute t_1 by t_2

Index

In this index, definitions from FRIPSE in Appendix C are marked with a star *.

Index of function and operation definitions

Index of type definitions

Bibliography

[AGPT85] V. Ambriola, F. Gianotti, D. Pedreschi, and F. Turini. Symbolic semantics and program reduction. *IEEE Transactions on Software Engineering*, SE-11(8):784–794, August 1985.

[AH87] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood Ltd., 1987.

[Ale87] Heather Alexander. Executable specifications as an aid to dialogue design. STC Technology Ltd., Newcastle-under-Lyme, 1987.

[All78] John Allen. *Anatomy of LISP*. McGraw-Hill Book Company, 1978.

[And88] Derek Andrews. Report from the BSI panel for the standardisation of VDM (IST/5/50). In [BMJ88], pages 744–78, 1988.

[Apt81] Krysztof R. Apt. Ten years of Hoare's logic: A survey — Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[B+87] F.L. Bauer et al. *The Munich Project CIP, Vol. II.* Lecture Notes in Computer Science 292. Springer-Verlag, 1987.

[BCJ84] H. Barringer, J.H. Cheng, and C.B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[BEJ87] D. Bjørner, A.P. Ershov, and N.D. Jones, editors. *Workshop Compendium: Partial Evaluation and Mixed Computation*, Gl. Avernæs, Ebberup, Denmark, October 1987.

[BEL75] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT — a formal system for testing and debugging programs by symbolic execution. *ACM Sigplan Notices*, 10(6):234–245, June 1975.

[BGW82] R.M. Balzer, N.M. Goldman, and D.S. Wile. Operational specification as the basis for rapid prototyping. *ACM Sigsoft Software Engineering Notes*, 7(5):3–16, December 1982.

[BMJ88] R. Bloomfield, L. Marshall, and R. Jones, editors. *VDM '88 — The Way Ahead. Proc. 2nd VDM-Europe Symposium, Dublin, Ireland.* Springer-Verlag, September 1988. Lecture Notes in Computer Science 328.

[Bro85] M. Broy. Extensional behaviour of concurrent, nondeterministic, communicating systems. In M. Broy, editor, *Control Flow and Data Flow: Concepts of Distributed Programming.* Springer Verlag, 1985.

[BSI88] BSI. VDM specification language protostandard. Technical Report BSI IST/5/50, April 1988.

[Bur74] R.M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing '74.* North-Holland Publ. Co., 1974.

[CC77a] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. ACM Symposium on Principles of Programming Languages, Los Angeles,* pages 238–252, January 1977.

[CC77b] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In *Proc. ACM Conference on Language Design for Reliable Software, Raleigh, South Carolina,* pages 77–94, March 1977.

[CH79] D. Coleman and J.W. Hughes. The clean termination of PASCAL programs. *Acta Informatica,* 11:195–210, 1979.

[CHT79] Thomas E. Cheatham, Jr., Glenn H. Holloway, and Judy A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Transactions on Software Engineering,* SE-5(4):402–417, July 1979.

[Cla76] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering,* SE-2(3):215–222, September 1976.

[Coh83] Donald Cohen. Symbolic execution of the GIST specification language. In *Proc. 8th Int. Joint Conference on Artificial Intelligence '83 (IJCAI-83),* pages 17–20, 1983.

[Coh84] Donald Cohen. A forward inference engine to aid in unterstanding specifications. In *Proc. of the National Conference on Artificial Intelligence (AAAI '84),* August 1984.

[CR84] Lori A. Clarke and Debra J. Richardson. Symbolic evaluation — an aid to testing and verification. In [Hau84], pages 141–166, 1984.

[CSB82] D. Cohen, W. Swartout, and R. Balzer. Using symbolic execution to characterize behavior. *ACM Sigsoft Software Engineering Notes,* 7(5):25–32, December 1982.

[Cut80] Nigel J. Cutland. *Computability.* Cambridge University Press, 1980.

[Dav73] Martin Davis. Hilbert's tenth problem is unsolvable. *American Mathematical Monthly*, 80:233–269, 1973.

[dBZ82] J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54(1/2):70–120, 1982.

[DDJ+85] B.T. Denvir, V.A. Downes, C.B. Jones, R.A. Snowdon, and M.K. Tordoff. IPSE 2.5 project proposal. Technical report, ICL/STC-IDEC/STL/ University of Manchester, 7th February 1985.

[DE82] Roger B. Dannenberg and George W. Ernst. Formal program verification using symbolic execution. *IEEE Transactions on Software Engineering*, SE-8(1):43–52, January 1982.

[Der85] Nachum Dershowitz. Computing with rewrite systems. *Information and Control*, 65(2/3):122–157, May 1985.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Series in Automatic Computation. Prentice-Hall, Inc., 1976.

[EJ87] Andrei P. Ershov and Neil D. Jones. Two characterizations of partial and mixed computation. Typescript, 1987.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[Ers82] Andrei P. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.

[Fet88] James H. Fetzer. Program verification: the very idea. *Communications of the ACM*, 31(9):1048–1063, September 1988.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proc. of Symposia in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[Flo84] Christiane Floyd. A systematic look at prototyping. In Budde, editor, *Approaches to Prototyping; Proc. Namur*, pages 1–18. Springer-Verlag, October 1984.

[Fut71] Yoshihiko Futamura. Partial evaluation of computation processes — an approach to a compiler-compiler. *Systems, Computers, Control*, 2(5):45–50, 1971.

[GH85] A. Geser and H. Hussmann. Rapid prototyping for algebraic specifications — examples for the use of the RAP system. Techn. Report MIP-8517, Univ. Passau, Fakultät für Mathematik und Informatik, December 1985.

[GHW85] J.V. Guttag, J.J. Horning, and J.M. Wing. Larch in five easy pieces. Technical report No. 5, DEC Systems Research Center, Palo Alto, CA, July 1985.

[GJ85] H. Ganzinger and N.D. Jones, editors. *Programs as Data Objects*. Springer-Verlag, 1985. Lecture Notes in Computer Science 217.

[GM82] J. Goguen and J. Meseguer. Rapid prototyping in the OBJ executable specification language. *ACM Sigsoft Software Engineering Notes*, 7(5):75–84, December 1982.

[Hau84] Hans-Ludwig Hausen, editor. *Software Validation — Proc. Symposium on Software Validation, Darmstadt, W. Germany, September 1983*. North-Holland Publ. Co., 1984.

[Heh84] Eric C.R. Hehner. Predicative programming. *Communications of the ACM*, 27(2):134–151, February 1984.

[Hen84] Peter Henderson. me too — a language for software specification and model building — preliminary report. Technical Report FPN-9, Univ. of Stirling, Dept. of Computer Science, December 1984. Second draft.

[HI88] Sharam Hekmatpour and Darrel Ince. *Software Prototyping, Formal Methods and VDM*. Intern. Computer Science Series. Addison-Wesley, 1988.

[HJ88] Ian Hayes and Cliff B. Jones. Specifications are not (necessarily) executable. December 1988. Submitted for publication.

[HK76] Sidney L. Hantler and James C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys*, 8(3):331–353, September 1976.

[HM85] Peter Henderson and Cynthia Minkowitz. The me too method of software design. Technical Report FPN-10, Univ. of Stirling, Dept. of Computer Science, October 1985. Revised draft.

[HO80] Gérard Huet and Derek Oppen. Equations and rewrite rules — a survey. In R. Book, editor, *Formal Language Theory: Perspectives and Open Problems*. Academic Press, 1980.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall Int., Series in Computer Science, 1985.

[How78a] William E. Howden. DISSECT — a symbolic evaluation and program testing system. *IEEE Transactions on Software Engineering*, SE-4(1):70–73, January 1978.

[How78b] William E. Howden. An evaluation of the effectiveness of symbolic testing. *Software — Practice and Experience*, 8:381–397, 1978.

[Hug85] John Hughes. Strictness detection in non-flat domains. In [GJ85], pages 112–135, 1985.

[Hus85] H. Hussmann. Rapid prototyping for algebraic specifications — RAP-system user's manual. Techn. Report MIP-8504, Univ. Passau, Fakultät für Mathematik und Informatik, March 1985.

[Inc87] D.C. Ince. The automatic generation of test data. *Computer Journal*, 30(1):63–69, January 1987.

[JL88] C.B. Jones and P.A. Lindsay. A support system for formal reasoning: Requirements and status. In [BMJ88], pages 139–152, 1988.

[Jon80] Cliff B. Jones. *Software Development — A Rigorous Approach.* Prentice-Hall Int., 1980.

[Jon86] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall Int., 1986.

[JW75] Kathleen Jensen and Niklaus Wirth. PASCAL *User's Manual and Report.* Springer-Verlag, 1975.

[KE85] Richard A. Kemmerer and Steven T. Eckmann. Unisex: a Unix-based symbolic executor for PASCAL. *Software — Practice and Experience*, 15(5):439–458, 1985.

[Kem85] Richard A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, SE-11(1):32–43, January 1985.

[Kin76] J.C. King. Symbolic execution and program testing. *Communications of the ACM*, 19:385–394, July 1976.

[Kin80] J.C. King. Program reduction using symbolic execution, December 1980. IBM Research, San Jose, California.

[Kne87a] Ralf Kneuper. Animation of specifications: a survey. IPSE 2.5 project document 060/00069, December 1987. Univ. of Manchester, Dept. of Computer Science.

[Kne87b] Ralf Kneuper. Symbolic execution of specifications: user interface and scenarios. Technical report UMCS-87-12-6, Univ. of Manchester, Dept. of Computer Science, December 1987.

[LM88] Peter Lindsay and Richard Moore. The all-new FRIPSE specification. IPSE 2.5 project document 060/00146/2.2, November 1988. Univ. of Manchester, Dept. of Computer Science.

[MLP79] R.A. De Millo, R.J Lipton, and A.J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5), 1979.

[Myc81] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, Univ. of Edinburgh, 1981.

[Nau82] Peter Naur. Formalization in program development. *BIT*, 22:437–453, 1982.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall Int., 1987.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Dept., September 1981.

[Plo84] Erhard Ploedereder. Symbolic evaluation as a basis for integrated validation. In [Hau84], pages 167–188, 1984.

[Pnu86] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency.* Springer-Verlag, 1986. Lecture Notes in Computer Science 224.

[QB+87] W.J. Quirk, J.P. Booth, et al. The role of validation in the FOREST method. Technical report, GEC Research, UKAEA Harwell, GEC Avionics, Imperial College, 1987.

[Rud87] Piotr Rudnicki. What should be proved and tested symbolically in formal specifications? In *4th Intern. Workshop on Software Specification and Design. California '87*, pages 190–195. IEEE Computer Society, 1987.

[Sch86] David A. Schmidt. *Denotational Semantics — A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[SDDS86] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to* SETL. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[Ses85] Peter Sestoft. The structure of a self-applicable partial evaluator. In [GJ85], pages 236–256, 1985.

[Shn87] Ben Shneiderman. *Designing the User Interface.* Addison-Wesley, 1987.

[Sin72] M. Sintzoff. Calculating properties of programs by valuations on specific models. *ACM Sigplan Notices*, 7(1):203–207, January 1972. Proc. ACM Conf. on Proving Assertions about Programs.

[Sol82] Daniel M. Solis. The Unisex system — a symbolic executor for the PASCAL language. Master's thesis, Dept. of Computer Science, Univ. of California, Santa Barbara, California, 1982.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* Logic Programming Series. MIT Press, 1986.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Swa82] William R. Swartout. GIST English generator. In *Proc. of the National Conference on Artificial Intelligence (AAAI '82)*, 1982.

[Swa83] William R. Swartout. The GIST behavior explainer. Research report ISI/RS-83-3, USC/Information Sciences Institute, July 1983.

[Tur79] Valentin F. Turchin. A supercompiler based on the language REFAL. *ACM Sigplan Notices*, 14(2):46–54, February 1979.

[TW83] D. Talbot and R.W. Witty. Alvey programme for software engineering. Published by the Alvey Directorate, November 1983.

[vEK76] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, October 1976.

[Wie88] Morten Wieth. Loose specification and its semantics. Submitted for publication, November 1988. Dept. of Computer Science, Technical University of Denmark, Lyngby, Denmark.

[Zav84] Pamela Zave. The operational versus the conventional approach to software development. *Communications of the ACM*, 27(2):104–118, February 1984.