Johannes Eriksson

Tool-Supported Invariant-Based Programming

Turku Centre *for* Computer Science

TUCS Dissertations
No 127, August 2010

Tool-Supported Invariant-Based Programming

Johannes Eriksson

To be presented, with the due permission of the Department of Information Technologies at Åbo Akademi University, for public criticism in Auditorium Gamma, the ICT building, Turku, Finland, on August 17, 2010, at 12 noon.

Åbo Akademi University
Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland

2010

Supervisor

Prof. Ralph-Johan Back
Department of Information Technologies
Åbo Akademi University
Joukahaisenkatu 3-5, 20520 Turku
Finland

Reviewers

Prof. Bernhard Aichernig
Institute for Software Technology
Graz University of Technology
Inffeldgasse 16 b, 8010 Graz, Austria

Prof. Joseph Kiniry
Software Development Group (SDG)
Programming, Logic, and Semantics Group (PLS)
IT University of Copenhagen
Rued Langgaards Vej 7, 2300 Copenhagen S., Denmark

Opponent

Prof. Bernhard Aichernig
Institute for Software Technology
Graz University of Technology
Inffeldgasse 16 b, 8010 Graz, Austria

ISBN 978-952-12-2446-1
ISSN 1239-1883

Abstract

The development of correct programs is a core problem in computer science. Although *formal verification methods* for establishing correctness with mathematical rigor are available, programmers often find these difficult to put into practice. One hurdle is deriving the *loop invariants* and proving that the code maintains them. So called *correct-by-construction* methods aim to alleviate this issue by integrating verification into the programming workflow. *Invariant-based programming* is a practical correct-by-construction method in which the programmer first establishes the invariant structure, and then incrementally extends the program in steps of adding code and proving after each addition that the code is consistent with the invariants. In this way, the program is kept internally consistent throughout its development, and the construction of the correctness arguments (proofs) becomes an integral part of the programming workflow. A characteristic of the approach is that programs are described as *invariant diagrams*, a graphical notation similar to the state charts familiar to programmers.

Invariant-based programming is a new method that has not been evaluated in large scale studies yet. The most important prerequisite for feasibility on a larger scale is a high degree of automation. The goal of the *Socos project* has been to build tools to assist the construction and verification of programs using the method. This thesis describes the implementation and evaluation of a prototype tool in the context of the Socos project. The tool supports the drawing of the diagrams, automatic derivation and discharging of verification conditions, and interactive proofs. It is used to develop programs that are correct by construction.

The tool consists of a diagrammatic environment connected to a verification condition generator and an existing state-of-the-art theorem prover. Its core is a semantics for translating diagrams into verification conditions, which are sent to the underlying theorem prover. We describe a concrete method for 1) deriving sufficient conditions for total correctness of an invariant diagram; 2) sending the conditions to the theorem prover for simplification; and 3) reporting the results of the simplification to the programmer in a way that is consistent with the invariant-based programming workflow and that allows errors in the program specification to be efficiently detected. The tool uses an efficient automatic proof strategy to prove as many conditions as possible automatically and lets the remaining conditions be proved interactively. The tool is based on the verification system PVS and

uses the SMT (Satisfiability Modulo Theories) solver Yices as a catch-all decision procedure. Conditions that were not discharged automatically may be proved interactively using the PVS proof assistant.

The programming workflow is very similar to the process by which a mathematical theory is developed inside a computer supported theorem prover environment such as PVS. The programmer reduces a large verification problem with the aid of the tool into a set of smaller problems (lemmas), and he can substantially improve the degree of proof automation by developing specialized background theories and proof strategies to support the specification and verification of a specific class of programs. We demonstrate this workflow by describing in detail the construction of a verified sorting algorithm.

Tool-supported verification often has little to no presence in computer science (CS) curricula. Furthermore, program verification is frequently introduced as an advanced and purely theoretical topic that is not connected to the workflow taught in the early and practically oriented programming courses. Our hypothesis is that verification could be introduced early in the CS education, and that verification tools could be used in the classroom to support the teaching of formal methods. A prototype of Socos has been used in a course at Åbo Akademi University targeted at first and second year undergraduate students. We evaluate the use of Socos in the course as part of a case study carried out in 2007.

Sammanfattning

Utveckling av korrekta program är ett grundläggande problem inom datavetenskapen. Ehuru det existerar *formella verifieringsmetoder* för att fastställa korrekthet med matematisk stränghet, upplever programmerare ofta dessa som svåra att tillämpa i praktiken. En stötesten är att härleda *invarianter* samt bevisa att programkoden bevarar dessa. Så kallade *correct-by-construction*-metoder strävar till att lindra problemet genom att integrera verifiering i programutvecklingsarbetet. *Invariantbaserad programmering* är en praktiskt orienterad correct-by-construction-metod i vilken programmeraren först fastställer programmets invariantstruktur, och därefter utökar programmet stegvis. Varje steg består av en kodningsfas följd av en verifieringsfas som fastställer att den tillagda programkoden är konsistent med invarianterna. Genom detta förfarande hålls programmet internt konsistent under hela utvecklingsarbetet, och konstruktionen av riktighetsargumenten (bevisen) blir en väldefinierad del av arbetsflödet. Ett särdrag hos metoden är att program beskrivs i form av *invariantdiagram*, en grafisk notation som är snarlik de för programmerare välbekanta tillståndsmaskinerna.

Invariantbaserad programmering är en ny metod som ännu inte utvärderats i större skala. Den viktigaste förutsättningen för att metoden skall vara utförbar i större skala är att den kan automatiseras i hög grad. Målet med *Socos-projektet* har varit att bygga verktyg som stöder utveckling och verifiering av program enligt metoden. Föreliggande avhandling beskriver implementationen och utvärderingen av ett prototypverktyg inom Socos-projektet. Verktyget stöder diagramritning, härledning och avskrivning av bevisvillkor, samt interaktiva bevis. Det används för att bygga program som är korrekta som en direkt följd av utvecklingsprocessen.

Verktyget består av en diagrambaserad omgivning kopplad till en teorembevisare. Dess kärna är en semantik för översättning av diagram till bevisvillkor, vilka skickas till en underliggande teorembevisare för förenkling. Vi beskriver en konkret metod för att 1) härleda tillräckliga villkor för total korrekthet hos ett invariantdiagram; 2) skicka bevisvillkoren till en avancerad teorembevisare för förenkling; och 3) rapportera verifieringsresultaten till användaren på ett sätt som stöder arbetsflödet i invariantbaserad programmering och som gör det möjligt att identifiera fel i programspecifikationen. Verktyget använder existerande effektiva bevisstrategier för att avskriva så många villkor som möjligt automatiskt och stöder därutöver interaktiv hantering av de återstående villkoren. Verktyget är baserat

på verifieringssystemet PVS och använder SMT-verktyget (Satisfiability Modulo Theories) Yices som grundläggande beslutsprocedur. Villkor som inte avskrivs automatiskt kan bevisas interaktivt i PVS.

Det verktygsstödda arbetsflödet för invariantbaserad programmering är mycket snarlikt den process genom vilken en matematisk teori utvecklas i en datorstödd teorembevisningsomgivning såsom PVS. Programmeraren reducerar ett stort bevisproblem med hjälp av verktyget till mindre delproblem (lemman), och denne kan väsentligt förbättra automatiseringen av bevisen genom att utveckla specialiserade bakgrundsteorier och bevisstrategier för att stöda specifikationen och verifieringen av en specifik klass program. I avhandlingen demonstrerar vi detta arbetsflöde genom att ge en detaljerad beskrivning av konstruktionen av en verifierad sorteringsalgoritm.

Det är vanligt att verktygsbaserad programverifiering ingår i ingen eller ringa omfattning i universitetens undervisningsprogram i datavetenskap. Därutöver introduceras programverifiering ofta som ett fördjupat och rent teoretiskt ämne utan koppling till det arbetsflöde som lärs ut i de tidiga och praktiskt orienterade programmeringskurserna. Vår hypotes är att verifiering kan introduceras tidigt i datavetenskapsutbildningen, och att verifieringsverktyg kan användas för att stöda undervisningen av formella metoder. En prototyp av Socos-verktyget har använts i en kurs som ingår i grundundervisningen i datavetenskap vid Åbo Akademi och som är riktad till första och andra årets studerande. Vi utvärderar här tillämpningen av Socos i kursen som en del av en fallstudie genomförd år 2007.

Acknowledgements

This thesis is not the product of a single person, but the result of a joint endeavor of which I feel greatly privileged to have partaken. Without the dedicated support of many people, this thesis would not have come about.

I wholeheartedly thank my supervisor, Professor Ralph-Johan Back, for introducing me to the world of computer science research, and for believing in my ability to finish this project more than I did myself. His proficiency in applying sharp theory to versatile areas of computer science and software engineering, as well as methodical goal-oriented approach, have been constant sources of inspiration. During the whole process he sustained me with unfailing encouragement and immensely helpful criticism on the shaping of the thesis.

Professor Bernhard Aichernig and Professor Joseph Kiniry kindly agreed to review the thesis. I sincerely thank them both for taking time to provide useful comments and suggestions, which greatly helped improve my manuscript. I would additionally like to thank Bernhard Aichernig for accepting to be the opponent in the public defense of my thesis.

Thanks go to my co-authors Victor Bos, Linda Mannila, Luka Milovanov and Magnus Myreen. During my undergraduate studies Victor showed me, through impeccable example, how mathematics is implemented in a computer. Linda is an excellent researcher as well as teacher, and our collaboration in the classroom has taught me many things. Without the process-oriented and agile coaching of Luka, the software on which the research of this thesis is based would not have been realized; our collaboration on experimental software engineering was also inspiring. Magnus conceived and implemented a large portion of the first Socos system, and his uncompromising example taught me a great many things about the mechanics of program verification.

I want to thank Ivan Porres for offering me a job in the Gaudí Software Factory during my undergraduate studies, and for ultimately convincing me to pursue doctoral studies. Whenever I was mathematically challenged, Viorel Preoteasa patiently explained the intricacies of higher-order logic and automatic theorem proving; moreover, he has also been a good colleague and friend. While working at the department I have benefited from many interesting software engineering-related discussions with Marcus Alanen and Torbjörn Lundkvist. I also want to thank Herman Norrgrann for his active involvement in the early days of the Socos project.

Brian Plüss provided during his stay with the department in 2007 several inspiring ideas.

In developing the software to carry out the research in this thesis, I have been helped in various capacities by many people. I worked with several great teams in the Gaudí Software Factory, and maintain a fond memory of the unique spirit (and engaging coffee break discussions) that pervaded these teams. In particular, I want to thank the people involved in developing and testing Socos: Federico Dobal, David Eränen, Jon Haikarainen, Fredrik Holmberg, Sören Höglund, Nazrul Islam, Juuso Jokiniemi, Sami Nevalainen, Niklas Nylund, Håkan Olin, Mikael Sand, Daniel Sjöblom, Sebastian Strand, John Tran and Johan Ånäs. I also want to thank the participants in the "Programsemantik" courses of 2007 and 2008 for valuable feedback on the Socos tool.

Thanks go the department secretary Christel Engblom for keeping the department up, Joakim Storrank and Jockum Lillsund for keeping the hardware up, and Tomi Mäntylä at TUCS for taking care of the publications. During the years of working at the department, I have shared many interesting discussions with fellow colleagues Pontus Boström, Fredrik Degerlund, Åke Gustavson, Mats Neovius, Mikolaj Olszewski, Patrick Sibelius, and Kim Solin.

I am honored to have participated in the TUCS graduate school, and grateful for its financial support and travel grants. I am also honored to have been able to work with the Department of Information Technologies at Åbo Akademi University, under the direction of Professor Babro Back and Professor Johan Lilius. I am grateful to the Academy of Finland for funding the Center for Reliable Software Technology (CREST), and to TEKES for financing the development of Socos via the Tores project.

I also want to thank my friends from outside of work for not asking too many questions about the topic of my thesis. Finally, I am grateful for the love and support of my family throughout my education and academic career.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter gives the motivation why we consider the subject of this thesis to be timely and relevant. We define the goals of the research, the methodology used and the tools built. We summarize the contributions of the thesis, and present a structural overview of the remainder of the contents.

1.1 Motivation

Despite more than 50 years of development, the software industry is still fraught with faulty products. Undetected errors introduced in the specification, design and implementation phases of the software process manifest as faults in the deployed products. Test-based validation is the standard quality control method in industry, but is generally unable to discover all faults in a software before deployment. Faulty software leads to low reliability, incurs high maintenance costs, and impugns the credibility of the software industry as a whole [56].

A stronger alternative to testing, *program verification*, also has a research track record dating back over 50 years [102]. Formally verifying a program involves the construction of a rigorous mathematical proof establishing that the program satisfies its specification. While so called *formal methods* have long sought to deliver the elusive goal of software that is provably correct, integration of formal methods into industry practice has been marked by recalcitrance. The late eighties and early nineties saw a heated debate on the role of formal methods in software engineering [75, 90]. Formal methods have been hampered by high up front cost, difficulty to apply in existing processes, lack of practicable tools [52, 98], but also by misconceptions on both academic [137] and industrial [45, 89] sides of the fence. Formal methods still remain very much in the domain of experts and their use is restricted to security features like cryptographic protocols, and to environments where the cost of failure is high such as safety-critical and other dependable systems.

Customers of software have traditionally prioritized feature richness and per-

formance over reliability. However, as more sectors of society become software dependent, one may expect the market to grow less tolerant of faulty software, thereby increasing cost-effectiveness of formal methods and escalating incentive to invest in them [147]. Economics is a main driver for acceptance; for instance, the hardware industry widely adopted formal verification in response to much-publicized and expensive fiascos such as the Intel FDIV bug. As a case in point, the world's largest software manufacturer, Microsoft, has for a number of years now devoted much resources to its *trustworthy computing* initiative [119], in which reliability is one of the fundamental pillars; a concrete example is the focus on static analysis tools for device driver development [31].

Testing and debugging. The *modus operandi* of most programmers is to write an initial version of a program, invent a set of test data, execute the program on the test data, and validate its results against the specification. Discovered errors, "bugs", are corrected and the "debugging" cycle is re-iterated until the tests pass. The debugging cycle involves a feedback loop, since a programmer often does not have a complete understanding of how a program he (or someone else) has written works (or fails). The cycle is therefore a kind of empirical exploration in which the specification serves as a hypothesis, and the test cases become observations to either strengthen or falsify the hypothesis. Programmers often record their tests in a structured way—e.g., using *unit testing* [100]—so that existing tests can catch new bugs that are introduced as the program is changed. It is generally much easier to keep a program consistent if the feedback loop is short and the tests are run frequently, compared to debugging a large set of changes at once. The practice of continuous testing [144] takes this to the extreme.

Program verification. Since even the most meticulous of testing regimens cover only a fraction of all possible inputs, faults inevitably remain undiscovered—a point succinctly enunciated by Dijkstra's oft-quoted remark [64] that *"program testing can be used to show the presence of bugs, but never to show their absence*." Ensuring the absence of errors requires reasoning about all possible execution paths of a program, proving mathematically that the program behaves correctly (according to some primitive axiomatization of its runtime environment) with respect to its specification. This is a much stronger result, but also one that is much harder to achieve, since it requires the construction of a mathematical correctness proof. In the verification process the onus is on the programmer to not only write the code, but also to motivate with mathematical rigor that each part of the code satisfies the specification. For this to be feasible, the verifying programmer cannot assent to the view that the program is an entity into which malicious bugs can "creep" from the outside. Every fault is put into the program by the programmer, who therefore must attain complete control over and full understanding of the artifacts he is producing.

The role of workflow. However, programming as a process usually does not start from a fundamental, complete understanding of the program and program domain at hand. Achieving this level of understanding is typically an iterative process. Just like the debugging cycle gradually increases the programmer's understanding of the program under construction, a verified program may have been preceded by many incomplete verification attempts, each acting as a stepping stone towards the final and correct version. For the programmer the stepping stones are lessons learned, providing additional insight into and understanding of the problem being solved. Thus, also in verification there is a feedback loop, in which the actions of the programmer are influenced by interaction with a verification tool. If the feedback loop is short, and the process is modular in the sense that each unit of the program is verified before the next is added, the process can be considered a *correct-by-construction* development process.

Verification tools. Computer aided software engineering (CASE) tools and integrated development environments (IDE) increase programmer productivity by automating many of the chores of software engineering. The traditional design, implementation, testing and debugging activities are supported by a panoply of modeling tools, source code editors, debuggers, unit testing frameworks, test data and oracle generators, and other specialized tools. Formal verification has its own range of activities that can be automated. Research in mechanized reasoning has made available a host of powerful *theorem provers*, capable of automating many of the deductions required in formal verification. These tools are key to making formal methods tractable, since they significantly reduce the human effort required in the verification process. Research in tool-supported verification has reached a point where several tools have been applied successfully in industrial projects, and use is becoming increasingly widespread [103, 158]. There may be reason to anticipate that verification tools will be as ubiquitous in the future as type checkers are today in software development: Hoare has introduced the concept *verifying compiler* [95]—a compiler that verifies the correctness of the programs it compiles—and envisions that such a compiler may be reality within 15-20 years [96, 97]. If this scenario unfolds, it is not unreasonable to think that the programmers of tomorrow may be producing verified software on a daily basis. Nevertheless, tool-supported program verification—in academia as well as in industry—certainly enjoys a prominent presence in the zeitgeist.

1.2 Invariant-based programming

This thesis is an inquiry into the practical utility of *invariant-based programming*—in the sequel abbreviated IBP—when supported by state-of-the art verification tools. IBP is a correct-by-construction oriented program development method introduced in the 1980's by Back [11]. It was inspired by early work of Reynolds

[140] on reasoning with transitions and invariants, and has since been developed by Back towards a hands-on correct-by-construction method [16].

IBP is based on diagrammatic notations, invariants, and a close relationship between the program and its correctness proof. IBP provides a light formalism that is easy to learn and that supports teaching formal methods. The programs are called *invariant diagrams* and expressed in a notation superficially similar to Harel statecharts [91], or UML state machines [129, pp. 582–586]. However, rather than describing finite sets of states and transitions, invariant diagrams use the concept of *situations* to partition the program into a finite collection of state subsets. A situation is a collection of *constraints* over the program statespace. Situations can be *nested* to inherit the constraints of outer situations. Situations are connected by *transitions*—program statements comprised of guards (tests) and assignments that constitute the actual executable part of the program.

Figure 1.1 shows an example invariant diagram. The program in the diagram computes the sum of the natural numbers up to n using a loop to accumulate the sum in the variable s. The situations are drawn as boxes with rounded corners. Constraints are written in the top left hand corner of the situations. The transitions, drawn as arrows connecting to the edges of the situations, describe the flow of control. Program statements are written next to the transitions. Guards are written inside square brackets, and := denotes assignment of a value to variable.

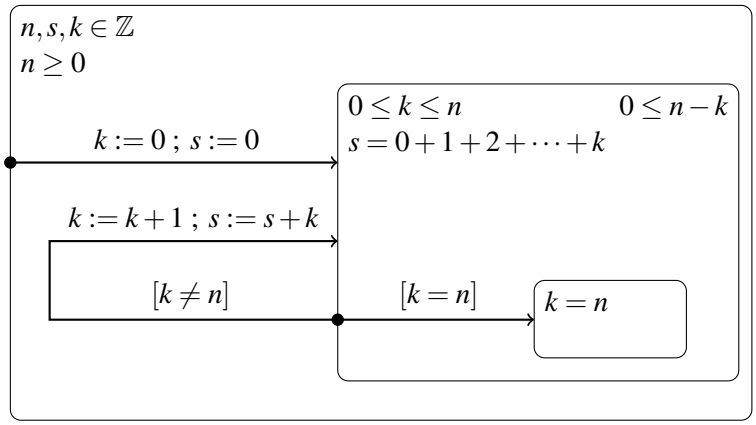


Figure 1.1: An invariant diagram representing a program that computes the sum of the first n natural numbers

The workflow of IBP differs significantly from that of *a posteriori* verification approaches, in which the programmer annotates the already written code with invariants and assertions. In IBP, the programmer instead establishes the situation structure of a program before he adds the transitions; the program is adapted to fit the invariants rather than vice versa. IBP is also an incremental verification methodology: while developing a program, the programmer verifies that each transition is *consistent* as it is added and before moving to the next transition. A

transition is consistent if and only if its effect on the state is consistent with the target situation; a program is consistent if and only if all transitions are consistent. Additionally, the programmer verifies *liveness* by checking that at least one transition is executable from each non-final situation, and *termination* by checking that each cycle decreases a *variant*. A variant is a measure with a lower bound of the remaining number of cycles, and is written in the upper right hand corner of a situation.

1.3 Tool support for IBP

An invariant diagram is not only an executable program, but also a correctness proof outline. *Verification conditions*—logical formulas whose validity imply the correctness of a program—can be derived straightforwardly from the diagram. The number of conditions so produced is large, even for small programs; however, the majority of them are typically rather shallow and could be discharged automatically by a state-of-the-art theorem prover. Thus, if the generation and discharging of conditions is automated, the remaining conditions that the programmer has to consider can be significantly simplified. The *Socos Project*, directed by professor Ralph-Johan Back at the Department of Information Technology of Åbo Akademi University has since 2005 focused on the development of tool support for IBP. The goal of the project has been to build a tool—the *Socos environment*—that supports the construction, verification and compilation of invariant-based programs within a single framework. Below we summarize the design goals behind the Socos environment along three lines: *practicability*, *transparency* and *learnability.*

Practicable. Since IBP was introduced as a hands-on method, we wanted Socos to be usable by, and provide value to, programmers with little or no formal methods experience. Firstly, familiarity with theorem provers should not be required for basic use of the tool; a basic background in predicate logic and the IBP workflow should be enough to get started. Secondly, the tool should provide a high degree of automation. This involves automatic verification condition generation and the use of state-of-the-art theorem provers to discharge as many conditions as possible without user intervention. Interactive theorem proving should also be available to handle the remaining conditions. Thirdly, the tool should have an easy to use interface that supports the IBP workflow, in particular the development of the situations before the transitions. It must support a correct-by-construction workflow in the sense that the user can verify the consistency of incomplete versions of a program as stepping stones towards a complete, consistent program. Incompleteness means that some transitions have not yet been added, or that liveness or termination has not yet been proved. Feedback from the theorem prover should be communicated to the user in a visually intuitive way, to allow errors and inconsistencies to be identified quickly when things go wrong.

Transparent. Program verification is often viewed as being comprised of two sequential activities: constructing the program, and proving the verification conditions. A condition that is not automatically discharged poses a challenge to the programmer, who must determine if the condition is either false, or indeed true and provable with additional help (such as a guided proof). Making the wrong judgment can result in a time-consuming futile search for a proof. Also, when a program error is detected and the program is rectified, existing proofs must be rechecked as they may no longer be valid for the updated verification conditions. Predicting the effect of changes on proof effort requires a good understanding of the relationship between a program and its correctness conditions. In IBP, the correctness conditions are already present directly in the diagram. The advantage of this is that the programmer gets a much better intuition of the relation between the program and its correctness proof, and hence when the program is changed, better understanding of the effects of the changes on the proof effort. This ought to be mirrored in the tool: the programmer should work with a single data representation of the program—the diagram—while any intermediate steps (such as verification condition generation) should be hidden. The tool-supported workflow should integrate the programming and verification activities seamlessly.

Learnable. A commonly quoted barrier to industrial uptake of formal methods is the lack of familiarity with correctness concepts among software practitioners and the subsequent need for extensive and expensive retraining. Unless one resigns to the idea that formal methods are useful only if invisible—i.e., encapsulated into tools such that the user is completely shielded from the correctness concepts—program verification demands new working skills from programmers, in particular logical and mathematical reasoning. If these skills were learned in university, the transition could be smoother. However, mathematics and programming education are often separated within CS curricula. In 1998, Parnas [137, p. 196] wrote: *"Most computer science programs are schizophrenic. In some courses, we teach mathematical theory but spend little time showing how to apply it in software development. In other courses, we teach programming but rarely apply any theory."* We therefore think that a tool that integrates programming and verification should be introduced in the classroom, so that these skills can be learned within a single environment and in support of each other. We also think that there is value in familiarizing tools at an early stage in the CS program, rather than only in later advanced courses; hence, the tool should provide value to and be usable by students with no experience in formal methods or program verification.

1.4 The Socos environment

In collaboration with Ralph-Johan Back, Magnus Myreen and Viorel Preoteasa, we have designed, implemented and evaluated two versions of the Socos environment.

We will refer to these as Socos_1 and Socos_2 in the sequel. Both versions support the same verification workflow but differ in the programming language notation and the underlying theorem prover used. We will mention the explicit Socos version in discussions where the distinction is relevant, but otherwise refer to the tool only as Socos.

Socos supports the design, construction, and verification of invariant-based programs. The tool lets the programmer create and edit invariant diagrams in a graphical environment using drawing program like commands. By the click of a button, the tool generates verification conditions from the diagram and sends them to a state-of-the-art automatic prover for simplification. It then shows the simplified conditions to the programmer. These conditions are often help the programmer in identifying errors and omissions in the program. True conditions that were not discharged by the automatic prover may be proved interactively in a proof assistant.

Socos supports incremental correct-by-construction development of programs. The programmer can verify the consistency of a program that is incomplete in the sense that a number of transitions may still remain to be added. In the Socos workflow the programmer verifies continuously during development that each increment preserves the consistency of the program built thus far, and Socos provides intuitive visual feedback when something goes wrong. It also allows fine-grained control over the generation of liveness and termination conditions, making it possible to distribute the proof effort over the development process as desired by the programmer. Figure 1.2 shows a screenshot of a Socos_1 session, in which a version of the summation program in Figure 1.1 is being edited. The graphical interface consists of a menu bar, a tool bar, an editing area in which the diagram elements are created and manipulated, and a pane for inspecting the verification conditions. The leftmost part of the pane indicates that all conditions have been proved to be true by a validity checker (Simplify). The rightmost part shows the verification condition for the selected element (in this case the loop transition).

Socos_2 is based on the theorem prover PVS [143]. The programmer expresses all program specifications, situation constraints, and statements in the PVS language [136]. During verification, the tool generates verification conditions encoded within PVS theories. To simplify the specification and verification of programs, the user can connect custom PVS theories containing domain-specific definitions and proofs—in the sequel referred to as *background theories*—to the program being verified. Any PVS proof strategy [134] may be used to handle the generated verification conditions. We have defined a strategy based on the SMT solver Yices [69] to be used as the default catch-all strategy. The user can extended this strategy to use specific properties derived in background theories in the proofs, thereby raising the level of abstraction in the proofs and improving the degree of automation.

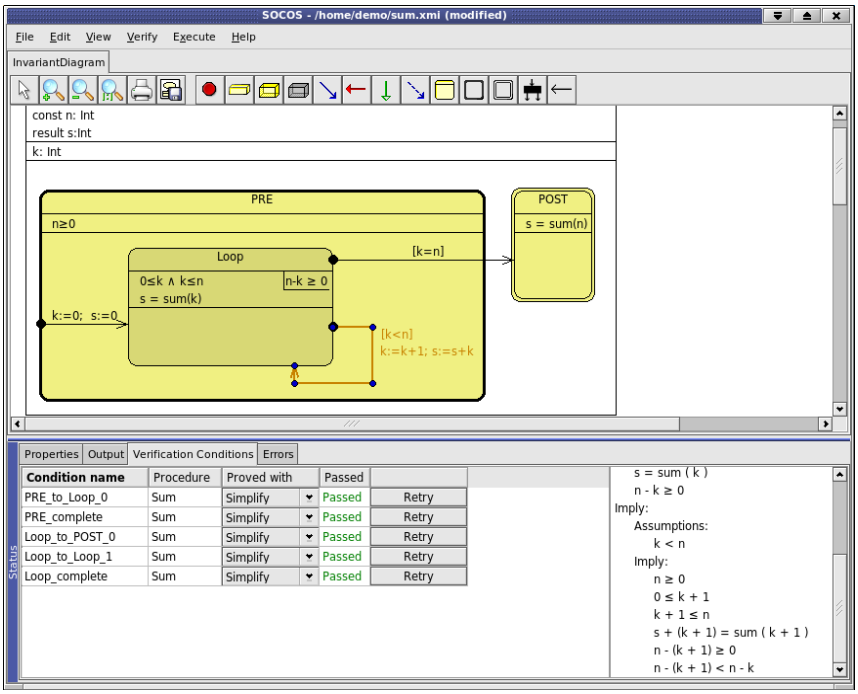


Figure 1.2: Program being verified in Socos$_1$

1.5 Research methodology

The formal methods community studies the theory and application of rigorous mathematical reasoning to software and hardware development. This includes, among others, formal programming and specification languages, semantics, verification techniques, theorem proving, program analysis and static checking, as well as the educational aspects. Rather than placing itself within one of these topics, the work presented in this thesis touches all of them; our research can be considered to be based in the *practical formal methods* field, in which researchers strive to find workflows, processes and tools that concretize the formal methods and make them accessible to programmers. The research methodology used in this study has exploratory, constructive and empirical aspects.

Exploratory. Since IBP is a new method, it was not clear from the outset how an effective programming environment for it should be designed, nor how to implement such a tool. Some of the main research questions are:

- How can we integrate tool support into the workflow of IBP?
- How can we use existing verification tools, such as specification languages and theorem provers, to support IBP?

- How can we achieve a high degree of proof automation?
- What is a suitable software architecture for an IBP tool?
- What is a suitable trade-off between generality and conceptual simplicity?

The research in this thesis investigates these questions and presents some answers based on the experience gained from building and evaluating the Socos environment.

Constructive. While there exists several well documented verification tools, adapting existing tools for a new programming methodology such as IBP is a considerable challenge. The central theme in our study is the prototyping of a proof of concept for demonstrating the feasibility of tool-supported IBP. We describe how the Socos environment is implemented, its interfaces to external components, and how it is used in practice to build correct invariant-based programs.

Empirical. The empirical aspect of our study concerns application of the tool in computer science (CS) education. Our hypothesis is that IBP improves novices' understanding of program verification, and that students who have had no exposure to formal methods can use a tool such as Socos to verify non-trivial programs. We collected data based on questionnaires, classroom observations and hand-in assignments. We analyze this data to shed light on the educational aspects of Socos, the feasibility of introducing IBP to novices, the perceived value of Socos by students as well as teachers, and potential usability issues.

1.6 Contributions of the thesis

We summarize the contributions of the thesis as follows.

Concrete tool-supported IBP. We define a concrete invariant-based programming language with dual textual and diagrammatic syntaxes, and give a set of sufficient *correctness conditions*, based on weakest preconditions, over the language. The correctness conditions are sufficient conditions for an invariant-based program to be consistent, live and terminating. The conditions admit fine-grained verification of invariant diagrams, and are suitable for checking in a theorem prover.

Software architecture. We have developed a software architecture for an IBP tool, comprising three main components: a diagram editor, a verification condition generator, and an interface to a theorem prover. When a diagram drawn in the editor is checked for correctness, the verification condition generator sends the generated conditions to the theorem prover, which tries to discharge as many of them as possible. Conditions that were not discharged automatically are reported

to the programmer. Valid conditions that were not discharged automatically may be proved interactively in the theorem prover.

Implementation. We have implemented the prototype tool Socos_1 according to the above architecture. Based on feedback from the prototype we have subsequently developed Socos_2 to be an improved PVS based verification and programming environment for IBP. We describe the development settings and history of Socos. We also presents the current status of implementation of Socos_2 and the most important ongoing and future work

Translation of invariant diagrams into PVS. We have defined and implemented an embedding of invariant diagrams in PVS. The invariant diagrams are translated into PVS theories such that consistency of the theories implies consistency of the diagrams. We describe the mechanics of the translation process and the issues involved.

Support for specification and proof automation. Socos_2 can use specialized domain-specific *background theories* and *proof strategies* to facilitate the specification of programs and to greatly increase proof automation. We describe a general *end-game strategy* for filtering verification conditions based on the SMT solver Yices, and show how this strategy can be extended to various program domains.

Case study. We have built a set of verified programs using Socos. The case study presented in this thesis is the standard heapsort algorithm. Despite the straightforward implementation of heapsort, verification is non-trivial. We develop in our tool an invariant-based implementation for which almost all verification conditions are proved automatically with the aid of a background theory. The background theory is general, and could be reused for other programs in the same domain.

Applications in teaching. In a pilot study in 2007, we evaluated Socos in the context of an undergraduate university course as part of a descriptive case study in IBP. The aim of the course was to teach the IBP methodology. The course was given to first and second year CS students who had no prior experience of program verification. Students used Socos to complete their final assignments in the course. The study encompassed analysis of hand-in assignments, classrooms observation and a post-course questionnaire.

1.7 Role of the author

The author of this thesis has been primarily responsible for the implementation of two graphical front-ends for Socos_1—one textual [18] and one diagrammatic

[22]—as well as for the evaluation of both systems. The author has been primarily responsible for the specification, software architecture, concrete design, implementation and evaluation of Socos_2 since the inception of the project. The translation of invariant diagrams into higher-order logic in PVS was developed in joint work with Ralph-Johan Back, Magnus Myreen, and Viorel Preoteasa; it was implemented in Socos_2 by the author. The research on IBP in teaching was carried out in collaboration with Linda Mannila and Ralph-Johan Back; the author has been primarily responsible for the evaluation of the Socos_1 environment as a teaching and learning aid.

The overall design of the Socos_1 and Socos_2 systems is a result of joint work with Ralph-Johan Back, Magnus Myreen and Viorel Preoteasa. A large portion of the concrete implementation and testing work for both Socos_1 and Socos_2 was carried out by student programmers employed at Åbo Akademi University. These developers have worked under direct supervision of the author. The extension of Socos_2 to procedures with multiple exits was carried out in collaboration with Ralph-Johan Back and Brian Plüss.

1.8 List of original publications

The original articles on which this thesis is based are listed in order of publication below.

1. Ralph-Johan Back, Johannes Eriksson, and Luka Milovanov. Using Stepwise Feature Introduction in Practice: An Experience Report. In *Proceedings of the 2nd International Workshop on Rapid Integration of Software Engineering Techniques – RISE 2005*, Heraklion, Crete, Greece, September 2005.[1]

2. Ralph-Johan Back, Johannes Eriksson, and Magnus Myreen. Verifying Invariant Based Programs in the SOCOS Environment. In BCS-FACS Workshop on Teaching Formal Methods: Practice and Learning Experience, London, UK, December 2006.

3. Ralph-Johan Back, Johannes Eriksson, and Magnus Myreen. Testing and Verifying Invariant Based Programs in the SOCOS Environment. In *Proceedings of the First International Conference on Tests and Proofs* – TAP 2007, Zürich, Switzerland, February 2007.[2]

4. Ralph-Johan Back, Johannes Eriksson, and Linda Mannila. Teaching the Construction of Correct Software Using Invariant Based Programming. In *Proceedings of the 3rd South-East European Workshop on Formal Methods*. Thessaloniki, Greece, December 2007.

[1] Also available as TUCS Technical Report number 705.

[2] Also available as TUCS Technical Report number 767.

5. Ralph-Johan Back, Johannes Eriksson, and Victor Bos. MathEdit: Tool Support for Structured Derivations. TUCS Technical Report 854, Turku Centre for Computer Science, December 2007.

6. Johannes Eriksson and Ralph-Johan Back. Applying PVS Background Theories and Proof Strategies in Invariant Based Programming. Accepted for publication in the *12th International Conference on Formal Engineering Methods – ICFEM 2010,* Shanghai, China, November 2010.

1.9 Organization of the thesis

The remainder of the thesis is organized as follows. Chapters 2–4 describe the background of the Socos tool. In Chapter 2 we trace the history of program correctness research leading up to IBP, and give a concrete example of the IBP workflow. Chapter 3 deals with the practical and technical aspects of program verification: workflow and tools. Chapter 4 is an introduction to the PVS language and interactive theorem prover, as well as to the Yices SMT solver.

Chapters 5–7 focus on the design of Socos. Chapter 5 gives a top-down introduction to the Socos programming language. Chapter 6 describes the verification methodology: the correctness rules and how they are used to generate verification conditions. Chapter 7 discusses the embedding of verification conditions into PVS and the use of proof tactics to discharge them.

Chapters 8 and 9 focus on the Socos environment from a user and workflow perspective. Chapter 8 describes an interactive session with Socos, in which we prove a sorting algorithm correct. In Chapter 9 we describe our experience from using Socos in teaching IBP to formal methods novices.

Chapter 10 summarizes the development settings and history of Socos, and presents the architecture of Socos_2 and its current status of implementation.

Chapter 11 concludes the thesis with a recapitulation, a discussion, and a list of future work.

Chapter 2

Programming for Correctness

This chapter gives an overview of the programming methodology research on which the work in this thesis builds. We first consider the goals and history of program verification, leading up to the advent of IBP. We illustrate the IBP workflow with an example.

2.1 Foundations of program verification

While testing can verify correctness provided a program is exhaustively checked for every possible input, even a single 64-bit integer constitutes a statespace too large to be practically covered in this way. Efficient techniques in the field of *model checking* [71] have been developed and successfully used to check finite-state systems of considerable sizes. Model checking typically targets decidable properties and focuses on efficient automation. It is, however, sensitive to the *state explosion problem*, whereby the number of states to be analyzed suddenly grows beyond practical limits. Furthermore, if the statespace is infinite, model checking is not at all applicable.

To assert the correctness of programs over large or infinite statespaces, *deductive methods* must be used. These methods build on logical inferences and rely on theorem proving: the programmer or the machine constructs a mathematical proof that the implementation satisfies the specification. Such a *correctness proof* establishes a formally well-defined relation—such as *refinement*—between the implementation and the specification. Program proofs tend to be tremendously detailed and must be rigorously constructed and checked. In practice machine checking is mandatory, which in turn requires the programming logic and proof theory to be fully mechanized. Programs can also be built to search for correctness proofs. However, deductive methods cannot be fully automated in the general, at least not for specification languages sufficiently expressive to describe non-trivial constraints on a program.

Researchers early recognized that reasoning about program correctness could

be made manageable by attaching *assertions* about the state to intermediate points (cutpoints) in a program. Such assertions are used to form inductive arguments for the correctness the program. The idea of assertion-based reasoning dates back to the forties with work by Goldstine and von Neumann [81], and Turing [148], but the first practical methods were developed independently by Naur [123] and Floyd [80] and presented in 1966 and 1967, respectively. Floyd also discusses the requirement that a program terminates within a finite number of steps, and considers *termination proofs* essential. A seminal paper by Hoare [94] introduced the *Hoare logic*, a set of axioms and rules for deriving correctness. Hoare's method did not initially consider termination: the verified program satisfies its specification under the condition that it terminates. It has become customary to refer to correctness with termination as an obligation as *total correctness*, and correctness with termination as an assumption as *partial correctness*.

The *predicate transformer semantics* introduced by Dijkstra [65, 66] was another step forward. A predicate transformer is a total function on predicates over the statespace. The canonical predicate transformer is the *weakest precondition*. The weakest precondition allows algebraic derivation of the most general constraints under which a program is guaranteed to terminate in a given postcondition. With predicate transformers a unified program calculus could be developed, handling correctness, nondeterminism, and termination within a single mathematical framework. This made it possible to develop correct programs in a "calculational style", analog to the way mathematical proofs are constructed [68].

The work of Dijkstra sparked a number of developments. The idea of *stepwise refinement*, in which a specification is transformed into an implementation in a series of correctness-preserving refinement steps, is due to Dijkstra and Wirth [64, 157]. Stepwise refinement was given a mathematical foundation by Back [8] and Morgan [118] in the *refinement calculus*. Back and von Wright subsequently developed the refinement calculus within a lattice-theoretical framework using higher-order logic [30].

2.2 Correctness proofs

In mathematics, the goal of proof is to demonstrate that some relevant theorem is universally true; in verification, the goal is to show that a program satisfies its specification. It has been argued that programmers should, as part of the programming process, develop correctness proofs in a way similar to how mathematicians write proofs of theorems. Dijkstra and Scholten proposed the *calculational proof format* [68] as a suitable format for constructing and presenting program proofs. In this format, proofs are developed in a linear fashion similar to the way calculations in algebra are performed, but with each step being a logical inference. Proofs in the calculational style are in general easier to read than those in, e.g., natural deduction, and it has become the standard textbook notation to which students of computer

science are subjected [87]. The proof format was later extended by Back and von Wright into a structured hierarchical format, where subproofs are incorporated into the syntactic structure [23]. It has been shown that proofs in this format correspond in generality to Gentzen-like proofs [17].

Despite ostensible similarities between proofs in mathematics and program correctness proofs, the actual process of developing program proofs is much different in practice, and is fraught with difficulty. While proofs in mathematics tend to be deep, elegant and short, program verifications are usually tedious, verbose and long. Mathematicians frequently use informal argumentation, while program proofs by way of their high level of detail and subsequent error-proneness require the use of a formal—and mechanized—proof calculus to be convincing. A program proof is also highly dependent on the program and specification—if either one is altered, the proof must be rechecked—a condition that arises often in practice in software development, whereas a mathematics problem tends to be much less of a moving target. Maintaining meticulous proofs under these conditions is a considerable challenge, and it has been argued that the processes involved in program verification are incompatible with how proofs are developed in mathematics [58]. On one hand, it is true that conditions derived from programs not built with verification in mind can be mentally impenetrable, and that proving them is not a tempting prospect. On the other hand, proofs developed together with the program and presented at an appropriate level of detail are not only clearer, but seem necessary to really understand a program; inspiring examples are given by Dijkstra [66] and Gries [84].

2.3 The constructive approach

Early approaches to program correctness often considered verification a separate activity from programming, to be performed after the implementation phase. Given a specification and a putative implementation, one would derive (by hand or with a tool) all the verification conditions and prove them. While sufficient for small programs, such *a posteriori* verification does not scale well. In particular, it is difficult to infer from already written code the intermediate assertions needed in a correctness proof. This observation indicated that perhaps the correctness arguments should influence the design of a program, rather than be used solely to assert the relation between a given implementation and a specification.

Dijkstra devised the *correct-by-construction* approach, in which the correctness proofs are developed hand-in-hand with the program [63]. In this method the programmer writes the intermediate assertions and loop invariants together with the code itself, and proves each subcomponent of the program before proceeding to the next. Adding one component and verifying it before moving on to the next enables errors to be detected earlier, and the verification effort is distributed over development time. Program proofs also tend to be easier to understand and

maintain, since they are connected to the structure of the code in a much more intuitive way. Finally, the approach contributes to the overall readability and understandability of the program code, as it forces the programmer to document the design decisions (pre- and postconditions and loop invariants).

2.4 Invariant-based programming

Invariant-based programming by Back [11, 14, 15, 16] is an elaboration of Dijkstra's original correct-by-construction approach. It goes one step further by letting the correctness arguments—the intermediate assertions and loop invariants—decide the structure of the program, rather than vice versa. Actual program statements are not added until the structure of the invariants has been defined. The rationale is that by lifting the correctness arguments to the design phase of development and giving them first choice in the structure of the final program, the program becomes easier to verify. IBP has the following design characteristics:

Invariants determine structure. Structure the program around invariants rather than control flow. Invariants become first class components, not optional code annotations, and formulating them becomes an integral part of program development.

Invariants are written before the code. The traditional way of writing code and annotating it with invariants may lead to complex invariants, since they have to be adapted to the existing code. In IBP the invariants are written first, and the code structured around them. This may conversely lead to the code being more complex; this is considered a trade-off for prioritizing verification.

Programs represent proofs. An invariant-based program is a correctness proof outline. Its structure allows reduction of the proof into rather fine-grained lemmas. These individual lemmas are usually quite shallow and easy to verify, which makes IBP a good target for automated theorem proving.

Unrestricted control flow. IBP does not restrict statements to single-entry and single-exit. Invariant-based programs are actually similar to flowcharts, in the sense that each sequence of statements is followed by an unrestricted goto-statement. Structured programming is aimed at keeping code disciplined in the absence of invariants, making it easier to understand the control flow; in IBP the discipline comes from the requirement to state and maintain invariants.

Locality of reasoning. There is no inherent difference between entry/exit points and intermediate points in a program. Each state transition can be verified in isolation, i.e., independently of the flow of control leading up to the transition.

Diagrammatic reasoning. Figures play two key roles in IBP. Firstly, informal figures of the data structures involved are a useful aid when formulating the invariants. Secondly, a formal diagrammatic syntax is used for the program itself.

Simple rules. Verification does not require complicated proof rules; everything that should be checked can be inferred from the diagram. No knowledge of a program calculus is required to use IBP; ordinary high school mathematics and the basics of predicate logic is sufficient.

IBP is a method for "programming in the small" in the sense that it covers the design of imperative programs but not of entire systems. For instance, information hiding, data abstraction and encapsulation are not addressed. IBP is not a substitution for object-orientation (O-O) or modularization but rather orthogonal to these techniques. We can, e.g., directly use IBP to implement the internals of a class method in an O-O program. Extension of IBP to object orientation is an open research topic that is outside the scope of this thesis.

2.4.1 Invariant diagrams

Invariant-based programs are called *invariant diagrams*. An invariant diagram is a directed graph with nodes called *situations* and edges called *transitions*. A situation is an indexed predicate, mapping a unique name to a predicate over a program statespace. A transition connects two situations and is labeled with a program *statement*. A situation represents a set of program states, while a transition represents a state change.

Situations are drawn as boxes with rounded corners and transitions as arrows connecting to the edges of situations. Situations can be nested to any depth, but are never drawn partially overlapping. Nesting is used to strengthen the enclosing situation; the nested situation inherits all the constraints from its enclosing situations. Transitions can go between any two situations regardless of the nesting structure. The statement of a transition is drawn adjacent to the arrow. A situation without incoming transitions is an *initial situation*; a situation without outgoing transitions is a *final situation*.

Invariant diagrams are syntactically close to Harel statecharts [91]. The operational interpretation is also similar to that of state machines. Control flow passes from one situation to another through the transitions; any enabled transition (i.e., a transition whose guard is true in the current state) can be triggered. Triggering of transitions is *demonically nondeterministic*: if several transitions are enabled in a state, the execution environment selects arbitrarily one to be taken. However, the semantics of nesting differs from that of statecharts. In statecharts, a transition may be triggered if its guard becomes true when the system is in any one of the nested substates of the source state. In IBP a nested situation inherits the constraints of the outer situations, but the transitions are not inherited; only transitions from the

edge of the current situation can be triggered. This means that while statecharts use nesting to economize the number of transitions, invariant diagrams use nesting to avoid repetition of constraints. It also means that an invariant diagram can be transformed into an equivalent flat diagram without changing the transitions, but by repeating the inherited constraints.

Example. Figure 2.1 shows an invariant diagram for a program determining if an integer array A of size N is a palindrome. If the array is not a palindrome, the program returns the index k of the first element such that $A[N-1-k] \neq A[k]$. The program compares the first element to the last, if they are not equal concludes that the array is not a palindrome, otherwise compares the second to the next to last, and so on. The loop terminates either at the first mismatch, or when the counter k reaches the index $\lfloor N/2 \rfloor$, in which case the array is concluded to be a palindrome. The floor function $\lfloor x \rfloor$ gives the largest integer not greater than x. The empty array is considered a palindrome.

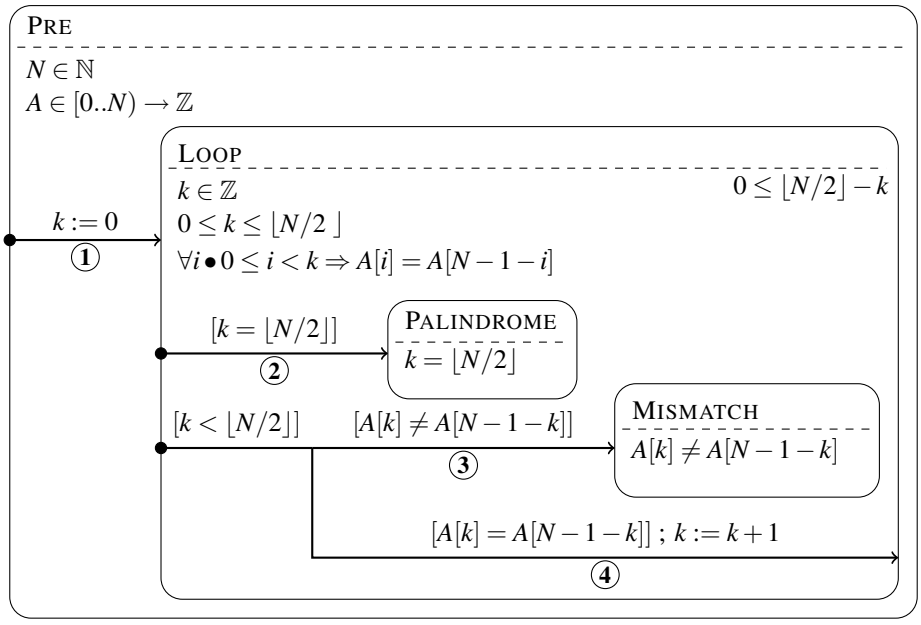


Figure 2.1: Palindrome program

The diagram contains four situations: PRE, LOOP, PALINDROME and MISMATCH. The outermost and initial situation, PRE states that N is a natural number and A is an integer valued function from 0 up to and including $N-1$; we denote this integer range with $[0..N)$. These constraints should hold throughout (the program never modifies A or N), so they are inherited by the remaining two situations.

LOOP is an intermediate situation, having both incoming and outgoing transitions. It states that the program variable k ranges over the integers $0, 1, \ldots, \lfloor N/2 \rfloor$, and additionally that $A[N-1-i] \neq A[i]$ for each i, $0 \leq i < k$. The predicate in the upper right corner is the variant for the loop. The variant identifies a function that is decreased for every iteration of the loop, $\lfloor N/2 \rfloor - k$, and a lower bound, 0.

The program has two final situations, PALINDROME and MISMATCH. In PALINDROME the loop counter k has reached the index $\lfloor N/2 \rfloor$, which together with the outer situations implies that A is a palindrome. MISMATCH states that elements at indexes k and $N-1-k$ differ, which together with the enclosing situations implies that k is the index of the first conflicting element.

There are four transitions in the program. Transition 1 assigns the initial value 0 to the counter k and enters the main loop of the program. The first exit transition, Transition 2, becomes enabled when k has reached the index $\lfloor N/2 \rfloor$. Guards are written inside square brackets; omitting the guard (as in the case of Transition 1) means that the transition is always enabled. We note that there is no default (else-) transition; this is deliberate, as writing each guard makes the assumptions for each transition explicit and serves to make the program closer to its correctness proof. To avoid duplication when several transitions share guards we use *transition trees*: Transition 3 and Transition 4 share a common guard $[k < \lfloor N/2 \rfloor]$, but diverge to either the postcondition MISMATCH if the elements at indexes k and $N-1-k$ are different (Transition 3), or back to LOOP if they are equal (Transition 4), in which case k is incremented.

In the next section, we describe how the program in Figure 2.1 is built and verified step by step.

2.4.2 IBP workflow

The process of refining an informal specification into a verified invariant-based program involves three stages: 1) formal specification, 2) sketching an algorithm, and 3) construction and verification of the final program. In this section we illustrate, in the context of the program in Figure 2.1, the main activities involved in each of the stages 1–3.

Specification. The goal of this stage is to formulate the pre- and postconditions of the program using mathematical logic. The starting point is usually an informal understanding of the concepts involved, e.g., that *"an array is a palindrome if it reads the same in both directions."* Informal figures are often useful at this point to visualize the data structures involved. Figure 2.2 is one way of expressing graphically that A is a palindrome. We draw A as a sequence of linked shapes and add equal signs between shapes to indicate that the two values are equal; for clarity, we draw separate figures for the cases of odd or even N.

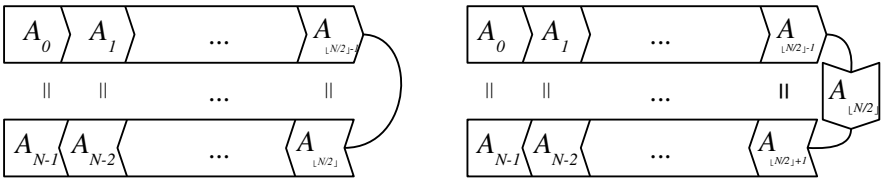


Figure 2.2: PALINDROME illustration

We also need to express that some element k of A violates the palindrome property. This is illustrated in Figure 2.3. Shading is used here for the portion of the array about which the MISMATCH situation states nothing.

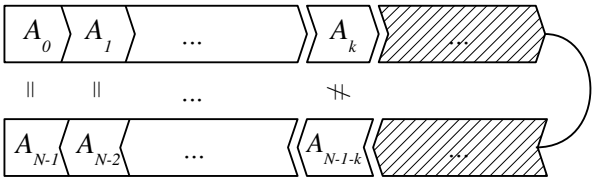


Figure 2.3: MISMATCH illustration

Such figures provide a useful bridge between the informal and the formal problem descriptions, and their construction is considered an essential step in the invariant-based programming approach. Note the use of symbolic (N, k) instead of concrete values in the figures; in this way a figure represents a situation, i.e., a set of states, rather than one specific state. It can be surprisingly challenging to find a figure that is expressive enough and does not overspecify the invariant. In this case, drawing the separate cases of odd and even N clarifies the fact that in both cases, we need only compare the elements in the range 0 to $\lfloor k/2 \rfloor - 1$ to their counterparts.

Formalizing Figures 2.2 and 2.3 involves giving a precise meaning to the relation between the two halves of the array, which in turn requires a notion of arrays and array access. We use the standard interpretation of an array as a total function from the (possibly empty) integer range $[0..N)$ to the value type. Array update is not needed here, since A will not be changed. In general, identifying the basic concepts needed to express the situations is referred to as developing the *background theory* for the program. While the background theory for a larger program would be substantial, the palindrome program is sufficiently simple that given the array concepts above, the rest of the desired properties can be stated directly in the invariants. Formulating the background theory is a challenging stage of invariant-based programming, since it directly influences how the program and

its proof will be developed in the subsequent stages. However, the effort is usually amortized over a large number of programs in the same domain; e.g., an array update theory would be used by most programs.

With the aid of the figures and the array definition, we can now give a first version of the precondition and the two postconditions of the program:

PRE	$N \in \mathbb{N}$ $\land\ A \in [0..N) \to \mathbb{Z}$
PALINDROME	$N \in \mathbb{N}$ $\land\ A \in [0..N) \to \mathbb{Z}$ $\land\ (\forall i \bullet 0 \le i \le \lfloor N/2 \rfloor \Rightarrow A[i] = A[N-1-i])$
MISMATCH	$N \in \mathbb{N}$ $\land\ A \in [0..N) \to \mathbb{Z}$ $\land\ k \in \mathbb{Z}$ $\land\ 0 \le k \le \lfloor N/2 \rfloor$ $\land\ (\forall i \bullet 0 \le i < k \Rightarrow A[i] = A[N-1-i])$ $\land\ A[k] \neq A[N-1-k]$

We note that we can use the common substructure to express the specification more compactly. Instead of repeating the types of N and A, we can nest the two postconditions within PRE to construct a first version of the invariant diagram without the loop invariant and the transitions (Figure 2.4).

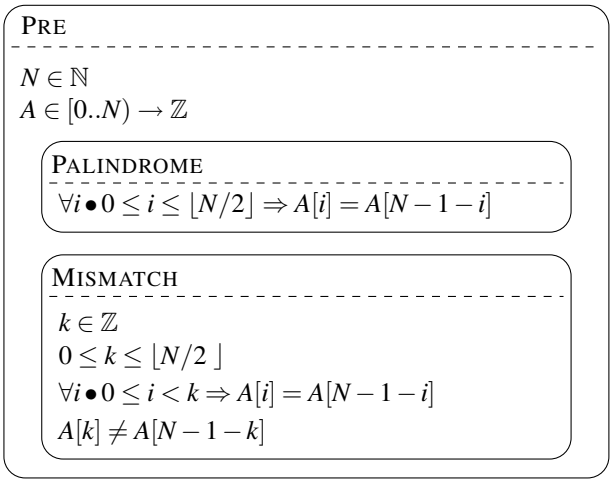
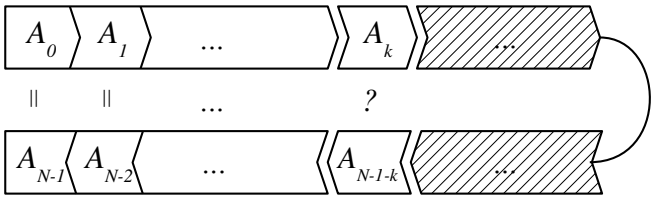


Figure 2.4: PALINDROME specification diagram

Sketching an algorithm. The next stage starts with a rough idea of the algorithm. This idea is refined into a concrete program by identifying intermediate situations

such that it becomes possible to implement a sequence of transitions from the precondition via the intermediate situations to the postcondition. The focus in this stage is on defining the structure of the situations; we do not actually write the transitions into the diagram until the next stage. Since we are solving the palindrome problem using a loop, we need to add a single intermediate situation—which we call LOOP—to define the *loop invariant* of the program. In general, a loop invariant should be formulated in such a way that it 1) can be established from the precondition by an initial assignment statement, 2) for the final value of the loop counter establishes the postcondition(s), and 3) is maintained by the loop transition. The invariant should express that the loop counter k ranges over the first half of the array (i.e., $0 \leq k \leq \lfloor N/2 \rfloor$) and that all elements below k are equal to their counterparts at the end of the array. Figure 2.5 illustrates the LOOP situation graphically and textually.



LOOP	$N \in \mathbb{N}$
	$\wedge\ A \in [0..N) \rightarrow \mathbb{Z}$
	$\wedge\ k \in \mathbb{Z}$
	$\wedge\ 0 \leq k \leq \lfloor N/2 \rfloor$
	$\wedge\ (\forall i \bullet 0 \leq i < k \Rightarrow A[i] = A[N-1-i])$

Figure 2.5: LOOP situation illustration and invariant

We note that the loop invariant is only a slightly weaker variation of the postcondition MISMATCH, the difference being that LOOP does not include the inequality $A[k] \neq A[N-1-k]$ (question mark in Figure 2.5). Furthermore, the postcondition PALINDROME is a special case of LOOP where $k = \lfloor N/2 \rfloor$. Thus, with these simple guards we are able to establish the postconditions. We can also reformulate the postconditions slightly to nest them within LOOP and arrive at Figure 2.6, which is exactly the diagram in Figure 2.1 except for the transitions and the variant.[1]

[1] We have actually altered the specification from Figure 2.4 slightly by inheriting the constraint on k into the postcondition PALINDROME. For the sake of presentation, we disregard this detail here.

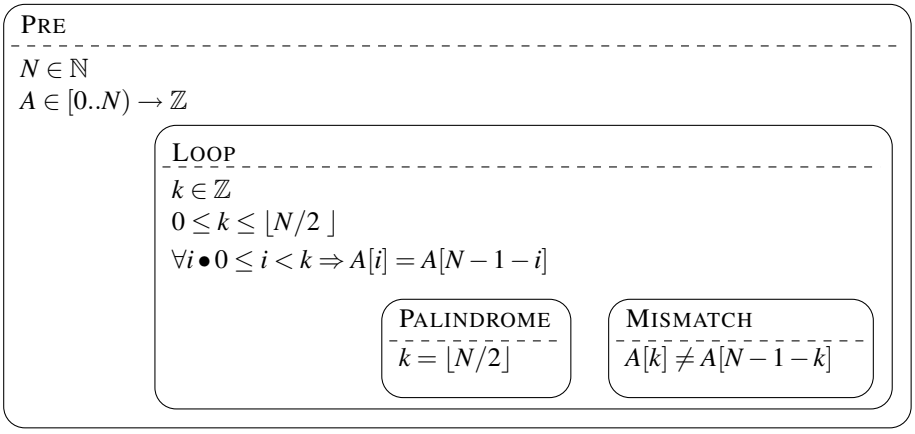


Figure 2.6: Final invariant structure

In summary, at the end of this stage we should feel confident to have formulated a loop invariant that is strong enough, satisfying criteria 1–3 above.

Implementation and verification. This stage consists of implementing and verifying the transitions. At this point, writing the actual statements is straightforward, as we already ought to have a good idea of how the program should work. The main effort in this stage lies in checking that the transitions preserve the invariants. After adding a transition, we verify that it is consistent before moving on to the next. In general, a transition S from situation P to situation Q is consistent if and only if every execution of S starting from a state satisfying P terminates in a state satisfying Q. In the sequel, we denote this proposition with the Hoare-like triple $\{P\}\ S\ \{Q\}$.

We derive the following *consistency condition* for Transition 1:

$$\frac{\begin{array}{rcl} & & N \in \mathbb{N} \\ N \in \mathbb{N} & & \wedge\ A \in [0..N) \to \mathbb{Z} \\ \wedge\ A \in [0..N) \to \mathbb{Z} & \Longrightarrow & \wedge\ \mathbf{0} \in \mathbb{Z} \\ & & \wedge\ 0 \leq \mathbf{0} \leq \lfloor N/2 \rfloor \\ & & \wedge\ (\forall i \bullet 0 \leq i < \mathbf{0} \Rightarrow A[i] = A[N-1-i]) \end{array}}{\{\text{PRE}\}\ k := 0\ \{\text{LOOP}\}}$$

To show that the formula below the line is true, we must prove that the implication above the line is true. Since this transition goes from PRE to LOOP and assigns 0 to the variable k, the antecedent is the predicate of PRE and the consequent is the combined predicates of LOOP and PRE where 0 has been substituted for all occurrences of k. Substituted subterms in the right hand side of the implication are shown in bold. Noting that the quantified expression is vacuously true, it is easy to see that this condition is true.

All verification conditions for the palindrome program are listed in Figure 2.7.

The conditions for Transition 2–4 also include as antecedents the transitions guards. Conditions for Transition 2 and 3 are trivial, as in both cases each conjunct in the consequent is also present in the antecedent.

For the cyclic Transition 4 we must discharge an additional *termination condition* to show that the program cannot loop indefinitely. Derivation of termination conditions requires a variant to be given by the programmer. The variant is a function that maps program states into a well founded set. In the diagram, we draw the variant in the upper right hand corner of the situations that are part of the cycle. We should prove that every cyclic transition decreases the variant:

$$\{\textsc{Loop} \wedge \lfloor N/2 \rfloor - k = V_0\} \quad \begin{array}{l} [k < \lfloor N/2 \rfloor]; \\ [A[k] = A[N-1-k]]; \\ k := k+1 \end{array} \quad \{0 \leq \lfloor N/2 \rfloor - k < V_0\}$$

Given an initial value V_0 of the variant, after executing the transition the new value should be above the lower bound 0 and strictly smaller than V_0. For brevity, we have combined the consistency and termination conditions into a single implication in Figure 2.7. The proof for Transition 4 requires basic quantifier reasoning.

In addition to consistency and termination we usually also want to check that the program is *live*. Liveness means that the program does not get stuck in an intermediate situation because all guards are disabled. Since Transition 1 is unconditional, situation PRE is trivially live. For LOOP we should check the following *liveness condition*:

$$\textsc{Loop} \quad \Longrightarrow \quad \begin{array}{l} k = \lfloor N/2 \rfloor \\ \vee (k < \lfloor N/2 \rfloor \wedge A[k] \neq A[N-1-k]) \\ \vee (k < \lfloor N/2 \rfloor \wedge A[k] = A[N-1-k]) \end{array}$$

The antecedent is the situation LOOP and the consequent is the disjunction of the collected guards of all transitions originating at LOOP. The disjunction is a consequence of $k \leq \lfloor N/2 \rfloor$.

2.5 Related approaches

The basic concepts of invariant-based programming originated in the late seventies. In 1978, Reynolds [140] reported on a method for rigorous development of goto-programs, emphasizing the importance of invariants and a disciplined workflow in reasoning about such programs. At the same time van Emden [150] developed a verification method for flowcharts based on similar ideas. The original *situation analysis* method by Back [9] is a development of Reynold's work and focuses on the use of invariants to guide the structure of the program as it is being developed. Back recently added the nested diagram notation [14, 15] to IBP, and has subsequently been developing it into a programming methodology suitable for teaching introductory program verification [16]. The mathematical foundations are also being refined. Recently, Back and Preoteasa [29] introduced a small-step

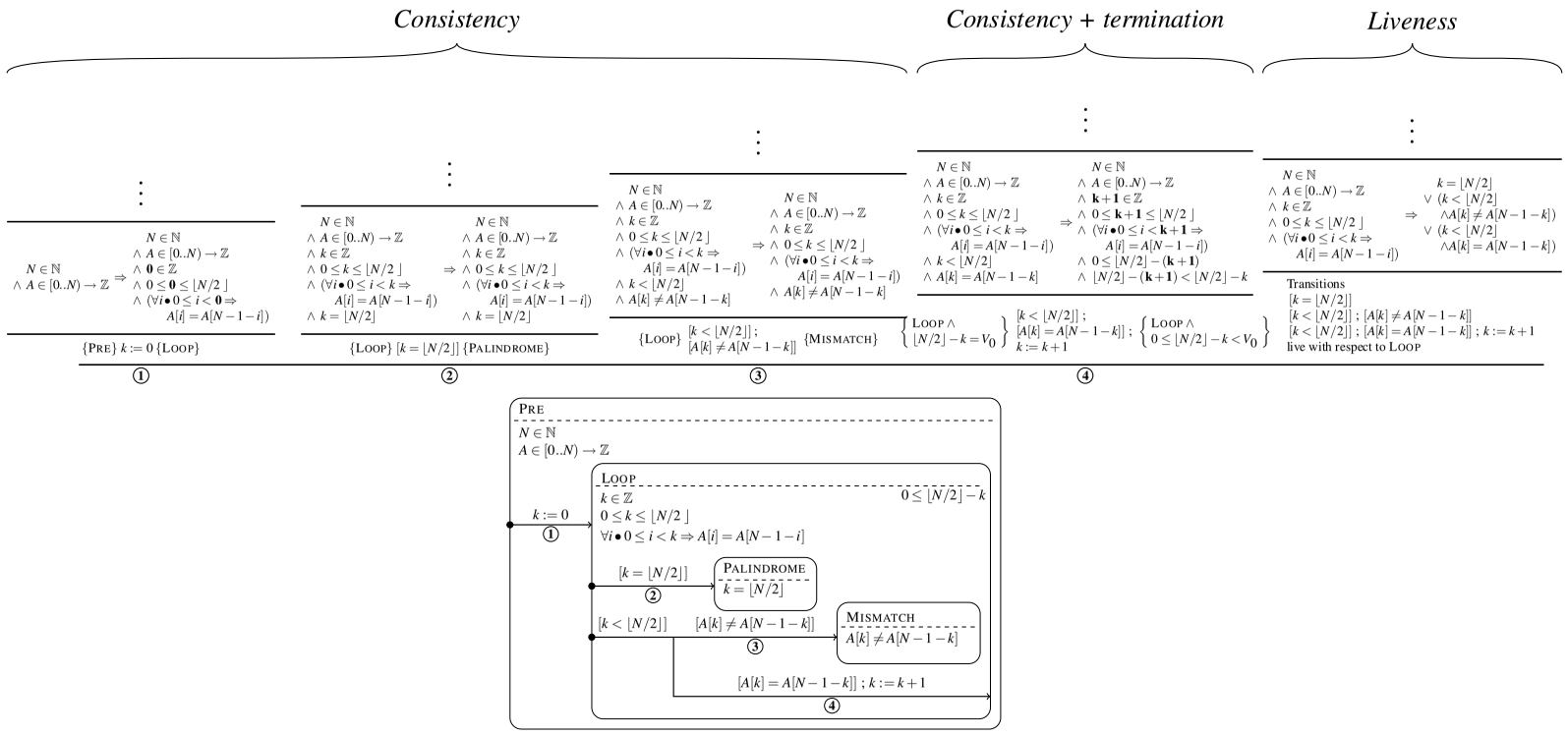


Figure 2.7: Proof tree outline for the palindrome program. Substituted subexpressions in consequents are in boldface.

operational semantics for invariant diagrams and proved that the proof rules are sound and complete with respect to the operational semantics; they also verified these rules in PVS.

2.6 Conclusions

Deductive verification of imperative programs is based on adding state assertions to select intermediate cutpoints. Assertions have no effect on the flow of control of the program, but decompose the verification into manageable tasks. Assertions can either be fitted to an already written program *(a posteriori* verification), or the program code can be given to fit the assertions (invariant-based programming). In this thesis we focus only on the latter approach.

Development of an invariant-based program comprises three stages: 1) understanding the problem and specifying the program; 2) sketching the algorithm and defining the structure of the invariants; and 3) implementing and verifying the transitions. Stage (1) involves building the background theory for the program and expressing the pre- and postconditions within this theory. In stage (2), the loop invariants and their nesting structure are defined. In stage (3) we implement the transitions and verify their correctness. Figures play a central role in all three stages.

There are three correctness conditions for invariant diagrams: consistency means that the transitions maintain the invariants; termination that the program does not run indefinitely; and liveness that the program does not "get stuck". An invariant diagram that preserves its invariants is consistent. In general, consistency is a basic condition that should always be proved, while termination and liveness may or may not be required depending on the operational interpretation of the diagrams. For example, we may be modeling a reactive system which is not intended to terminate, or which should allow indefinite blocking. The language presented in this thesis contains constructs to control the termination and liveness conditions generated.

We have given an example of the workflow of specification, implementation and verification in IBP. We note that all the conditions required to verify that the palindrome program is correct are present in the diagram. While the tabulated conditions in Figure 2.7 are verbose, deriving and proving them is straightforward. We can expect a reasonable theorem prover to discharge all the conditions in Figure 2.7 automatically.

Chapter 3

Tool-Supported Program Verification

This chapter focuses on the technical basis for program verification: specification languages, verification condition generators and theorem provers. We survey state-of-the-art deductive verification frameworks.

3.1 Verification workflow

Tools for deductive verification of imperative programs are typically designed around two central components:

1. A *verification condition generator* (VCG) to mechanically process the program and its specification into a set of *verification conditions* (VCs). The generated correctness conditions are logical formulas.

2. A *theorem prover* to try to discharge the generated conditions, and report the result back to the user. The theorem prover may benefit from additional input from the user (proof guidance).

The flow of data through these components is illustrated by Figure 3.1 below.

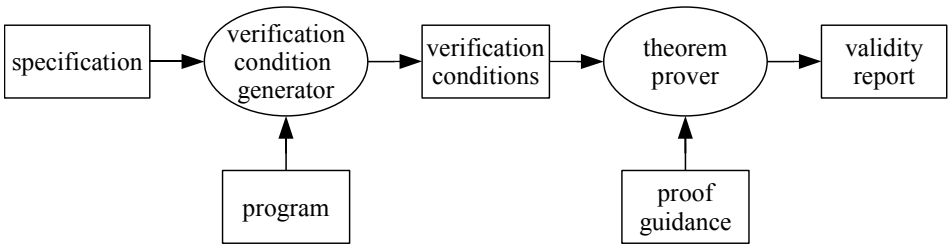


Figure 3.1: Basic program verifier architecture

The principle is that the VCs imply the (total or partial) correctness of the program. Thus, if the VCG is sound and all VCs can be discharged, the program satisfies its specification.

Similarly to debugging, the verification process does not follow a straight-line path to a correct program, but is rather an interactive, feedback-driven process. A program will contain errors and omissions caused by, among other factors, unfounded assumptions, insufficient understanding of the problem domain, mistakes in reasoning, and typos. To the effect of detecting such errors, verification is useful also when the theorem prover fails to prove a condition. Similarly to how a compiler aids in correcting syntactic errors by emitting messages and warnings indicating the cause and location of errors, a verification tool can provide fast feedback to improve the programmer's understanding of the program. The verification data flow annotated with this feedback loop is shown in Figure 3.2.

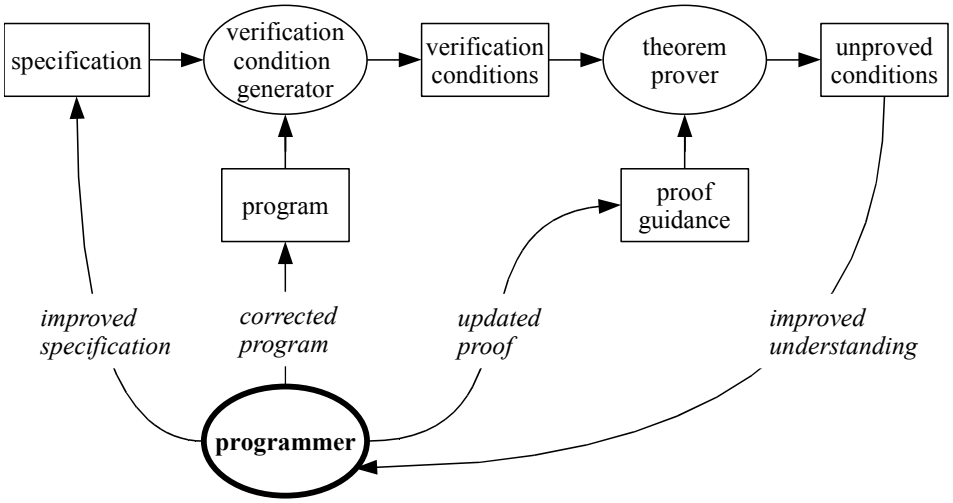


Figure 3.2: Verification workflow

A standard approach is to first attack the VCs with a fully automatic theorem prover to try to discharge as many of them as possible, hence decreasing the number of conditions to be considered by the programmer. A VC may be unprovable due to an error or omission in the program or specification, but it is also possible that the condition is valid but that the prover is not able to prove it, and/or is limited by time or memory constraints. Early in the development process the first case tends to be rather common, and the filtered conditions are often helpful for spotting the errors in the code. Interaction with a verification system can be an effective way of finding similar up-front errors as with unit tests, but with the advantage of being static and not requiring test data.

A frequent issue in later stages of the verification process is that some VCs are (evidently) true but the automatic prover cannot discharge them. Even if the

programmer just leaves these conditions unhandled, verification has provided value both in preventing bugs and in explicitly identifying a smaller set of assumptions on which the correctness of the program depends. However, for full verification the system may also provide support for proving the remaining conditions. The programmer then supplies additional guidance, such as a proof script, allowing the system to simplify the VCs into subconditions that it is able to discharge automatically.

3.2 Specification languages

A specification is a precise description of how a program should behave that allows for multiple alternative implementations. Languages for program verification hence include some form of specification notation. They may provide higher level modeling primitives such as sets and functions, modular specification methods (e.g., contracts), assertions, as well as various refinement mechanisms for relating implementation-level constructs to specification-level constructs. They may also provide *specification statements*, which may be composed like ordinary programming statements but which may not be implementable without further refinement.

Since the specification part of a language is used for describing programs rather than for providing instructions to a machine, it allows a much higher abstraction level than that of the implementation language. Some variant of mathematical logic underlie most specification languages. The logic has a direct effect on the expressiveness of the language and the degree of achievable automation. A less powerful logic makes a higher degree of automation possible, but restricts the expressiveness of specifications. For example, propositional logic is decidable, but is in practice not an expressive enough choice except for very limited specifications. A common choice is some variant of predicate logic, such as *first-order* or *higher-order logic*. In first-order logic variables can only range over a single universe of individuals; whereas in higher-order logic they can range over, e.g., functions and relations. First-order logic is much more useful for specification than propositional logic, but is in general undecidable. It may also be too weak; e.g., the transitive closure of a relation cannot be expressed in first-order logic. A higher-order logic is considered powerful enough for most specification needs.

Specifications are easier to write if the specification language integrates well with the implementation language. Hence, a language designed for verification defines the specification and programming constructs within the same syntactic structure; this is called a *wide-spectrum language* [39]. Alternatively, a pure implementation language may be extended with a specification notation, though such extensions may incur syntactic incompatibilites and language-within-language issues.

3.3 Semantics and embedding

Verification systems use a programming language semantics suitable for generating VCs, often some variant of *axiomatic semantics*. The axioms and inference rules are defined inductively over the syntax of the language, and the VC generation is directed by the syntactic structure of the program. Axioms and rules are defined to characterize the effect a particular statement has on assertions about the program state. For instance, the original Hoare logic [94] defines the following axiom for assignment statements and a consequence rule:

$$\frac{}{\{P[E/X]\}\, X := E\, \{P\}} \qquad \frac{P' \Rightarrow P \quad \{P\}\, S\, \{Q\} \quad Q \Rightarrow Q'}{\{P'\}\, S\, \{Q'\}}$$

The Hoare triple $\{P\}\ S\ \{Q\}$ is an assertion that a program S executed in a state satisfying P, if it terminates, does so in a state satisfying Q. X and E range over variables and expressions, respectively. $P[E/X]$ stands here for the syntactic substitution of the expression E for all free occurrences of variable X in P. The axiom states that any program of the form $X := E$ with precondition $P[E/X]$ and postcondition P is correct. Using Hoare logic, a VC for a general assignment program $\{P\}\, X := E\, \{Q\}$ can be derived as follows:

$$\dfrac{\begin{array}{c}\langle \text{proof} \rangle \\ \vdots \\ \mathbf{P \Rightarrow Q[E/X]}\end{array} \quad \dfrac{}{\{Q[E/X]\}\, X := E\, \{Q\}}\text{(axiom)} \quad \dfrac{}{Q \Rightarrow Q}\text{(axiom)}}{\{P\}\, X := E\, \{Q\}}\text{(consequence)}$$

Thus, in order to prove that the triple $\{P\}\, X := E\, \{Q\}$ correct, it suffices to prove that $P \Rightarrow Q[E/X]$ is true.

A mapping of terms in the programming language into terms in the underlying logic is called a *semantic function*. Distinction is made based on whether the programming language and semantic function are defined outside the underlying logic, so called *shallow embedding*, or within the logic, so called *deep embedding* [44]. In a deep embedding the abstract syntax of the embedded language is modeled as a type in the logic, and every program is a value of that type. This makes it possible to quantify over programs and prove general theorems about the embedded language in a deep embedding. A shallow embedding translates (outside of the logic) terms in the embedded language into terms in the logic. Hence one can only reason about the specific, translated programs. A shallow embedding is, however, significantly easier to implement and use, and allows sharing of values and types between the programming and theorem prover languages.

Axiomatic semantics is a substantial abstraction from the lower-level state-based *operational semantics* that more closely corresponds to how a program is executed by a computer. It is thus important that the verification semantics and the operational semantics are consistent. There are two aspects of this consistency.

Soundness means that every program verified according to the axioms executes correctly; *completeness* means that the axiomatic semantics is general enough to be used to verify any correctly executing program. Soundness is a vital requirement for any verification system. Completeness, however, while desirable is generally not attainable as there may be programs which produce the correct answer but cannot be verified in the system, e.g. due to missing intermediate assertions or loop invariants. It is a main design goal of a verification system to ensure that a sufficiently large set of programs is covered.

We finally note that for any axiomatic verification semantics, a strict mathematical assertion of soundness and/or completeness requires the underlying operational semantics to be fully axiomatized.

3.4 Theorem proving

Since program verification generates a large number of trivial VCs, filtering conditions through a powerful automatic theorem prover can significantly decrease the number of conditions that must be considered. For instance, a large industrial project carried out using the B-method [1] reported approximately 80 per cent of the VCs discharged automatically, and up to 90 per cent with tool adaptations [41]. Systems vary considerably in how the user interacts with the back-end theorem prover. Highly automated tools that do not support interactive proving may still allow the programmer to feed "facts" to the prover. These facts effectively become axioms, though they may also be verified by hand or in another system.

User-written proofs rely on a *proof checker* to assert the consistency of a proof with respect to some formal proof system. A chief design principle of theorem provers is that the checking is performed by a small trusted kernel encoding the proof rules of, e.g., sequent calculus. However, this requires proofs to be given at an extremely low level. Thus, in addition to checking there is a need for *proof tactics* and *strategies*, enabling proofs to be carried out at a higher level of abstraction, while remaining sound with respect to the formal system. The undecidability of any reasonably expressive logic dictates that full automation of proof construction is not achievable in the general, so interactive *proof assistants* enables the user to search for a proof interactively, in dialogue with the system. Such tools may provide advanced proof search heuristics to enhance and optimize proof construction.

Satisfiability Modulo Theories (SMT) solvers have evolved to be practical and highly efficient verification tool back-ends. SMT solvers combine boolean satisfiability (SAT) solving—typically some variant of the DPLL procedure [57]—with decidable theory fragments. The combination is usually based on the Nelson-Oppen method for cooperating decision procedures [125] or some development of this method. Common theories are linear arithmetic, uninterpreted functions with equality, and the theories of data structures such as arrays and bit vectors. SMT solvers are often designed to be used as back-ends for verification and constraint

solving systems, and are thus fully automatic and highly optimized for speed. The canonical use of an SMT solver in program verification is to prove that the negation of a VC is unsatisfiable. In general, these solvers are complete only for quantifier free logics; however, heuristics for quantifier instantiation make them useful in practice also for many conditions involving quantified formulas. Much research is focused on optimizing SMT solvers for program verification.

Since the proofs produced by automatic provers such as SMT solvers are seldom amenable to human inspection, the soundness of the verification hinges on the correctness of the prover itself. It has been argued that for verification to have a credible impact on the trustworthiness of software, programs should be accompanied by standardized representations of their correctness proofs such that another (independent) proof checker can be used to verify them. This raises the issue of proof interchangeability and open formats, where no consensus has been reached yet. Necula has introduced the concept of proof-carrying code [124], where potentially untrusted software is piggybacked with proofs of relevant safety properties. The idea is that proofs are stored in such a format that the execution environment can automatically check the validity of the proof as well as verify that the software conforms to its own safety rules.

3.5 Verification techniques and tools

Below we list some of the state of the art in techniques and tools for program verification.

3.5.1 Design by contract

Adding assertions to catch errors at run-time is common programming practice, and can be considered a first step towards embedding verification into the programming workflow. It has the advantage of being a light-weight addition feasible in any programming language without special support. Many languages support an `assert` statement or macro, such that assertion code can be easily disabled from the final product. The practice of documenting interfaces using pre- and postconditions and class/object invariants is known as *design by contract* [115]. Compared to assertions, design by contract is more involved, as it requires dealing with objects and relating values of variables in different states. The Eiffel language [114] is built from the ground up to support design by contract. The Java Modeling Language (JML) [108] adds design by contract capabilities to Java via annotations embedded in comments.

3.5.2 Extended static checking

Extended static checking (ESC) extends the scope of static checking from type correctness to properties such as the absence of null dereference, safety of array

indexing, and invariant maintenance. The goal of ESC is to find bugs, not full verification [61]. Most ESC tools are based on some variant of first-order logic, and a fully automatic first-order prover such as an SMT solver is typically used as the verification back-end. ESC has been a driver for research in automated theorem proving and SMT.

A tool for checking of Java programs against specification written in the JML called ESC/Java was originally developed by Flanagan et al. [79]. Development of this tool was later taken over by Cok and Kiniry to support the advances in JML and to address issues in the original implementation [53]. Leino et al. [60, 32] have developed the Boogie methodology for verifying programs written in, among other languages, a C# extension called Spec# [33].

3.5.3 Program verifiers

The first program verifiers were developed by King [106] and Good [82] in the late sixties. In 1973 Igarashi et al. [101] built a VCG for a subset of Pascal, which later evolved into the Stanford Pascal Verifier [109].

A number of tools target Java+JML [48]. LOOP [149] is a compiler that translates JML-annotated Java into VCs for (among other) the PVS theorem prover; the generated theorems are then proved using the proof assistant of PVS. Similarly, JACK [38] generates VCs for Coq and Simplify. Krakatoa [113] generates VCs for the verification framework Why [76], which supports several interactive and automatic theorem provers. The KeY System [40] is a verification system based on dynamic logic for a subset of Java called JAVA CARD. It is supported by a graphical theorem prover and integrates a number of SMT solvers, including Yices, as decision procedures.

The B-method [1] is one of the most successful industrial applications of formal methods, and is well known for having been used in the verification of a segment of the Paris metro system control software [41] and other large projects. B is based on set theory, and specifications are given as *abstract machines* with invariants and operations; the B tool assists in proving that the operations of a machine maintain the invariants, as well as in refining operations into executable code. Current development is centered on the Action Systems-based Event-B language [25], and the Rodin platform [2]. The Rodin platform is built on top of the Eclipse framework and integrates Event-B specification, specification animation, and a point-and-click proof assistant.

3.5.4 Theorem provers

The field of mechanized theorem proving has developed from an effort to mechanize mathematical and logical reasoning, as well as to support hardware and software verification. Notable early examples of theorem provers are NuPRL [54], LCF[116], and the Boyer-Moore prover Nqthm [46]. Based on LCF, Gordon

has developed the HOL system, an ML-based framework for proving program correctness in higher-order logic [83]. Nqthm was later developed into the program verification framework ACL2 [104]. HOL and ACL2 have both been used extensively in hardware verification. EHDM [142], developed at SRI, was the predecessor of PVS. Current state of the art interactive theorem provers include PVS [143] and Isabelle/HOL [128], both of which are based on classical higher-order logic. Another prover used for program verification is Coq, which is based on the calculus of inductive constructions [42]. This is just a partial list. The systems vary widely in methods of interaction and how proofs are built and represented; for instance, Isabelle proofs are based on forward reasoning while PVS proofs are goal-directed. A comparative overview of the major provers and their proof styles in the context of a concrete example is given by Wiedijk [154].

3.5.5 SMT solvers

First-order SAT and SMT solvers have developed into efficient, powerful tools for program verification. Simplify [62] has been used by a number of software verification tools, including ESC/Java. It is, however, no longer developed. Current state of the art includes Barcelogic [43], CVC3 [37], Yices [69], and Z3 [59]; this is just a partial list. A common standard input language for SMT solvers, the SMT-LIB language [36], is being developed to facilitate the writing of front-ends as well as the interchangeability of back-ends. All the listed solvers except Simplify support the SMT-LIB language (version 1.2). The international SMT-COMP competition [34] annually ranks the capacity and performance of SMT solvers based on a large collection of standard problems. SMT solvers have also been successfully integrated into several theorem proving systems as endgame provers. Examples include PVS and Yices, Isabelle and Yices [73], as well as HOL and Yices [153].

3.6 Summary

The architecture of a program verification system typically comprises a VCG and a theorem prover. The VCG translates a program and its specification into VCs which are sent to the theorem prover. Automatic tactics are used to discharge as many conditions as possible. Working with an automatic theorem prover in a programming context is a "static debugging" process, in which the output from the prover helps the developer in identifying errors in the program or specification.

A verification system should provide a means for describing what a program should do, in addition to how it does it. The specification part of the language must be expressive enough such that any relevant problem domain can be modeled. A higher-order logic is typically considered powerful enough for general-purpose specification.

A semantic function defines the translation of an imperative program and its specification into VCs. This consistency of a verified program hinges ultimately on the consistency of the verification semantics with respect to the environment in which the program will be executed.

Interactive theorem provers can be used for building and checking proofs of VCs. Proofs built in a theorem prover are guaranteed to conform to its underlying formal proof system. Decision procedures for common background theories implemented in SMT solvers have been developed to increase the degree of proof automation in practical program verification.

Chapter 4

PVS and Yices

This chapter is an overview of the PVS specification language and theorem prover. We describe the basics of the language as well as the most important proof commands. We also give an overview of the features of the SMT solver Yices, which is integrated as a decision procedure in PVS.

4.1 Introduction

PVS (originally an acronym for *Prototype Verification System*) is a specification and verification system developed by SRI International [143]. At the core, it consists of a higher-order specification language, a powerful interactive theorem prover, and an Emacs-based front-end [135]. Examples of auxiliary tools include an evaluation engine [145] and batch proving, scripting and checking [121]. PVS is open source and available under the GNU General Public License. It is implemented in Common Lisp and runs on the Linux, Solaris and Mac OS X platforms.[1]

Interaction with PVS is primarily via the Emacs front-end. PVS files are managed and edited as Emacs buffers. The interface provides commands to, among other tasks, parse and typecheck a file as well as to start an interactive session with the theorem prover. The prover can run in either interactive or batch mode. Interactive mode is used to create new proofs, while batch mode is used to replay existing proofs. The extension ProofLite [121] makes it possible to run scripted proofs without starting the Emacs front-end.

Commands to the prover are entered as S-expressions into an interactive prompt. Familiarity with Lisp is not required for normal use of PVS. However, when using the system in advanced ways—such as when writing custom prover strategies that query the internal structure of the sequent formulas or the prover state—it is often necessary to embed Lisp code into the proof commands. This also requires some degree of familiarity with the internals of PVS.

[1] PVS can be downloaded in source and binary form at `http://pvs.csl.sri.com/download` (link valid as of June 2010).

We proceed as follows. We first describe the PVS specification language. Following this, we give an overview of how proofs are built in PVS and how the PVS prover is controlled through commands and proof scripts. Finally, we summarize the functionality of the SMT solver Yices. We restrict the treatment throughout to features that are relevant to Socos and the case study in Chapter 8.

4.2 The PVS specification language

The PVS specification language [136] is a textual ASCII-based language. It is case-sensitive in identifiers but case-insensitive in keywords. For example, the following expressions are equivalent:

forall x : f(x) **or** F(x)
ForAll x : f(x) **OR** F(x)

The bold words are PVS keywords and may thus be written in any mixture of cases, whereas x, f and F are distinct identifiers. We will in the sequel use typographic mathematical symbols in place of the ASCII equivalents, rendering the above as:

$\forall \mathtt{x} : \mathtt{f}(\mathtt{x}) \vee \mathtt{F}(\mathtt{x})$

The typeset version of the PVS syntax should present no surprises. A glossary of the ASCII and typeset symbols used in this thesis is given in the Appendix on page 141.

4.2.1 Structure of specifications

The basic specification unit is the *theory*, which contains a number of *declarations*. Declarations assign names to types, constants, definitions, variables, axioms and theorems within the theory. Theories provide namespaces for declarations, as well as genericity and reusability through parameters. Declarations can be imported from one theory into another, enabling large modular theory hierarchies to be built. Importings must be acyclic.

PVS comes with an comprehensive library of theories called the *prelude* [133]. Prelude theories include functions, relations, orders, real numbers, integers, indexed sets, and so on. All prelude declarations are available in user created theories without being explicitly imported.

PVS supports specification of abstract datatypes—either at the top level or embedded within a theory—from which theories are automatically generated during typechecking [131]. Datatypes may be recursive and (like theories) parametric, allowing for specification of generic list- and tree-like structures.

4.2.2 Type system

The underlying logic of PVS is a simply-typed higher-order logic [132]. The type system provides base types such as `bool`, `nat`, `int` and `real`, and type constructors

to build new types from existing types. Types are closely related to sets: two types are equal if they denote the same set of values, and subtypes correspond to subsets. For example, `nat` is a subtype of `int`, `int` is subtype of `rational`, and `rational` is a subtype of `real`. Subtypes are introduced by *predicate subtyping* [141]; the subtype is defined by a predicate picking the elements from the supertype. For example, the type `nat` can be defined as a subtype of `int` using the following set comprehension-like notation:[2]

$$\texttt{nat} : \textbf{type+} = \{\texttt{i} : \texttt{int} | \texttt{i} \geq 0\}$$

The PVS design philosophy encourages the use of predicate subtyping. The syntax makes it easy to move from predicates to types: a type expression can be built from a predicate by enclosing the predicate in parenthesis. For example, the following introduces a new type `even_int` of even integers based on a predicate over integers called `even`? from the PVS prelude:

$$\texttt{even\_int} : \textbf{type+} = (\texttt{even?})$$

Predicate subtyping provides an expressive type system at the cost of rendering typechecking undecidable. Consequently, theorem proving is required to decide subtyping and/or type equality relations. PVS generates *type correctness conditions (TCCs)* during typechecking and tries to discharge them automatically using a default strategy. TCCs not discharged automatically can be proved interactively in the theorem prover. Above, the plus sign following the keyword is a type *judgement*: it tells the typechecker that the type `even_int` is non-empty, which results in the generation of an *existence TCC* requiring a witness for the type, and subsequently allows constants of type `even_int` to be declared without generation of an existence TCC. It is even possible to give the witness directly in the type declaration:

$$\texttt{even\_int} : \textbf{type+} = (\texttt{even?}) \textbf{ containing } \texttt{0}$$

In this case no TCC is generated, as the typechecker immediately can decide that `0` satisfies the `even`? predicate.

PVS supports uninterpreted types and subtypes. An uninterpreted type declaration introduces a new type that is disjoint from all other types except its own subtypes. An uninterpreted subtype declaration creates a subtype from a given type without further assumptions; i.e., two uninterpreted subtypes of a single supertype may or may not be disjoint.

Additional basic type constructors are the tuple (product type), function and record constructors. These constructors do not allow recursion (abstract datatypes are used in PVS to model and reason about recursive types such as lists and trees). A tuple constructor is of the form $[T_1, \ldots, T_n]$ where each T_i is a type expression. A

[2]For the sake of presentation, `nat` and other definitions in this section do not necessarily match their namesakes in the prelude exactly.

function type constructor is of the form $[D_1, \ldots, D_n \to R]$ where each D_i and R are type expressions. Specifically, the type $[T \to \texttt{bool}]$ is called a predicate over type T. Record types are similar to tuple types, but fields are labeled: $[\#l_1 : T_1, \ldots, l_n : T_n\#]$. The fields must be distinctly labeled, but in contrast to tuples the order of field declarations is irrelevant.

As an example, using records we can specify a type of dynamic arrays containing elements of type `T` with a field `len` for the number of elements and a field `elem` for accessing individual elements:

$$\texttt{vector} : \textbf{type+} = [\#\texttt{len} : \texttt{nat}, \texttt{elem} : [\texttt{nat} \to \texttt{T}]\#]$$

A powerful extension to predicate subtyping is *dependent typing*. A dependent type is a type that depends on a value. Dependent types are useful for succinctly encoding data invariants and dependencies directly in the types. With dependent typing, we can construct the following improved vector type:

$$\texttt{vector} : \textbf{type+} = [\#\texttt{len} : \texttt{nat}, \texttt{elem} : [\texttt{below}(\texttt{len}) \to \texttt{T}]\#]$$

Here `below` is a dependent type itself, defined in the prelude as follows:

$$\texttt{below}(\texttt{i} : \texttt{nat}) : \textbf{type} = \{\texttt{s} : \texttt{nat} | \texttt{s} < \texttt{i}\}$$

Now the domain of the `elem` function depends on the value of the `len` field, so that instead of covering all of `nat` it is limited to the integer interval $[0..\texttt{len})$. As is elaborated in [141], this definition provides a better notion of equality, since every vector value is now uniquely defined. This is also the array model which we use in our example in Chapter 8. A listing of the complete theory is given in the Appendix, page 146.

4.2.3 Variable and constant declarations

Variable declarations on the theory level bind names to types in the context of the theory. A name may then be used without explicit type annotation in binding expressions such as quantifiers and λ-abstractions.[3] For example, the declaration

```
u, v, w: var vector
```

introduces three variables `u`, `v` and `w` ranging over vectors. *Constant declarations* introduce named constants, which may be uninterpreted, as in:

```
x : T
vec1 : vector
```

[3] Variables declared at the theory level may in general not occur free in PVS terms. A notable exception is theorems, in which free variables are implicitly universally quantified.

In this case only the types of x and vec1 are known to PVS. From the previous declarations the type checker knows that both T and vector are non-empty. In general, declaring uninterpreted constants of types that are not declared as non-empty results in the generation of existence TCCs. Constants may also be interpreted, as in:

$$\texttt{vec2} : \texttt{vector} = (\#\texttt{len} := 1, \texttt{elem} := \lambda(\texttt{i} : \texttt{below}[1]) : \texttt{x}\#)$$

The above declares vec2 to be a one-element vector. Named functions may also be declared using the traditional name-argument binder notation. Also, multiple argument tuples may be sequenced to compactly define higher-order functions in a single declaration without use of λ-abstractions. For instance, the following declaration defines a curryfied concatenation function:

```
con(u)(v) : vector  =  (# len:=len(u)+len(v),
                          elem:=
                            λ(i:below[len(u)+len(v)]):
                              if i<len(u) then elem(u)(i)
                              else elem(v)(i-len(u)) endif #)
```

This declaration assigns con the type $[\texttt{vector} \to [\texttt{vector} \to \texttt{vector}]]$.

4.2.4 Formula declarations

Formula declarations associate names with closed boolean expressions. They are used to introduce formulas to be proved or invoked (as lemmas) in the prover. Three kinds of formulas can be added to a PVS theory: *axioms*, *assumptions* and *theorems*. A fourth type, *obligation,* is reserved for TCCs generated by the type checker.

Axioms are introduced with the keyword **axiom**. Injudicious use of axioms may introduce inconsistencies into the theory. Constants declarations are, on the other hand, guaranteed to add *conservative extensions* of the theory, meaning that they cannot give rise to inconsistency.

Assumptions are constraints on the theory parameters. They appear as axioms within the parametrized theory. Whenever the theory is imported into another theory, an instance of the assumptions for the actual parameters must be discharged.

The keyword **theorem**—or alternatively, **formula**, **fact**, **lemma** and several other available synonyms—introduces theorems that become proof obligations. For example, to add a lemma stating that con is associative we write:

$$\texttt{lem1}: \textbf{lemma}\ \forall(\texttt{u},\texttt{v},\texttt{w}) : \texttt{con}(\texttt{u})(\texttt{con}(\texttt{v})(\texttt{w})) = \texttt{con}(\texttt{con}(\texttt{u})(\texttt{v}))(\texttt{w})$$

The name lem1 is used to invoke the lemma from the theorem prover. Formula names exist in a separate namespace from constants and variables, but must be unique within a theory. PVS interprets a formula declaration containing free

variables as the universal closure of the free variables. We can thus state the above more compactly by omitting the $\forall(\mathtt{u},\mathtt{v},\mathtt{w})$ part. In the next we will see how theorems are proved in PVS.

4.3 Theorem proving in PVS

The PVS proof theory is based on *sequent calculus*. A PVS proof is a tree where each node is a sequent of the form

$$\gamma_1,\ldots,\gamma_n \vdash \delta_1,\ldots,\delta_m$$

where $\gamma_1,\ldots,\gamma_n$ are the *antecedent formulas* and $\delta_1,\ldots,\delta_m$ are the *consequent formulas*. At every point in the proof either $n>0$ or $m>0$. Proofs are carried out in a goal-directed style. The proof of a proposition α starts with the root sequent $\vdash \alpha$; the user then applies a proof rule that either proves the current sequent, or reduces it to a collection of subgoals. Each subgoal adds a branch to the proof tree. A branch is terminated by a step that proves the sequent. A proof is complete if every leaf sequent is proved. A sequent can be proved by reducing it to a form that is trivially true. A sequent is trivially true if for any γ_i, δ_j either γ_i is false, δ_j is true, or γ_i is syntactically equivalent to δ_j (modulo the names of bound variables).

A simple numbering scheme is used to identify sequent formulas in the prover: antecedents are labeled from the sequence of negative integers $-1,-2,-3,\ldots$ and consequents from the sequence of positive integers $1,2,3,\ldots$. Additionally sequent formulas may be given custom labels with the `label` command.

4.3.1 Commands

The logic of PVS is embodied in a small set of primitive inference rules (listed in, e.g., [146], Chapter 3). Every PVS proof corresponds to a sequence of applications of these rules. In practice, proofs are constructed using higher-level proof commands which encode the primitive inference rules. In total over a hundred commands are available; we list here a few that are relevant to this thesis. A complete list and the full documentation can be found in [146]. Many commands accept as the first parameter the number of the sequent formula to which they should be applied. For most commands this parameter may be omitted, in which case PVS either applies it to all formulas, or selects the formula based on command specific heuristics.

`beta` applies beta-reduction. For example, $(\lambda \mathtt{a},\mathtt{b}:\mathtt{a}+\mathtt{b})(1,2)$ is reduced to $1+2$.

`decompose-equality` decomposes equality between functions, records or tuples into a conjunction of equalities of the constituents. For example, if `f` and `g` are unary functions, $\mathtt{f}=\mathtt{g}$ is decomposed into $\mathtt{f}(\mathtt{x})=\mathtt{g}(\mathtt{x})$ where `x` ranges

over the type of the argument. For records and tuples, the command splits the proof tree for each component.

`expand` expands in the sequent the names given as parameters to their definitions, and then applies the `simplify` command.

`flatten-disjunct` performs disjunctive simplification of a sequent formula into a list of formulas. For example, $\alpha \vdash \beta \implies \gamma$ is reduced to $\alpha, \beta \vdash \gamma$. The depth to which formulas are flattened can be controlled with a parameter; by default the command iterates until flattening is no longer applicable.

`inst` instantiates universal quantifiers in the antecedent and existential quantifiers in the consequent.

`propax` discharges sequents that are trivially true, otherwise does nothing. PVS automatically applies `propax` to new sequents.

`skolem` introduces Skolem constants for existential quantification in the antecedent and universal quantification in the consequent. The variant `skolem!` of this command additionally invents names for the Skolem constants.

`split` branches a sequent into subproofs. For example, applying `split` on $\alpha \vdash \beta \wedge \gamma$ generates the sequents $\alpha \vdash \beta$ and $\alpha \vdash \gamma$. By default splits until the resulting formulas no longer are conjunctions, but the depth of splitting can be controlled with the `:depth` parameter.

`simplify, assert, ground, smash, grind` apply combinations of built-in decision procedures, automatic rewriting, quantifier instantiation, if-lifting, arithmetic and propositional simplification. Only `simplify` is primitive; the rest are strategies combining `simplify` and other proof commands. `grind` is the most general strategy: it applies automatic rewriting followed by repeated quantifier instantiation, if-lifting and simplification until the sequent is proved or a fixpoint is reached.

4.3.2 Example proof

Figure 4.1 shows a PVS proof of the formula `lem1`. The first step introduces skolem constants for the universal quantifier in the consequent. The second step uses `decompose-equality` to split the equality over `vector` into two separate branches for the components `len` and `elem`. Note that in the first branch, when expanding the definition of `con` the automatic simplification built into the `expand` command immediately simplifies the sequent to `TRUE`, which is then discharged automatically by PVS through `propax`.

The second branch applies `decompose-equality` again to transform function equality into range value equality. Names generated by the prover are suffixed

with ! and a positive integer. The next command is `expand`, which expands the definition of `con` and then simplifies. After this step, `smash` discharges the sequent.

A more compact proof of `lem1` applies `(grind)` in place of the `expand` commands in both branches. Since `grind` does both rewriting and simplification, it solves each branch in a single step. It is even possible to prove the lemma with a single command: `(grind-with-ext)`. This command is like `grind`, but also applies extensionality reasoning.

4.3.3 Strategy language

`smash`, `grind` and `grind-with-ext` are examples of *proof strategies* (tactics) for applying combinations of the primitive commands in an automated manner. They are useful for automating commonly occurring proof command patterns. PVS includes a small language for building custom proof strategies [134]. Below we list the main constructs in the strategy language; the full list can be found in [146].

`skip` does nothing.

`if` evaluates a Lisp expression and chooses a strategy based on the result. `(if` *cond* s_1 s_2`)` applies step s_1 if *cond* evaluates to true, otherwise applies s_2.

`let` allows embedding of arbitrary Lisp code into strategies. The command `(let ((`v_1 e_1`),...,(`v_n e_n`)` s `)` evaluates Lisp forms $e_1,\ldots,e_n$ and binds their values to the symbols $v_1,\ldots,v_n$ in step s.

`branch` applies a step and assigns a separate strategy to each subgoal. The command `(branch` s `(`s_1 ... s_n`))` applies for each $0 \leq i \leq n$ the step s_i to the i'th subgoal generated from applying s to the current sequent. If there are more than n subgoals, s_n is applied to the n'th as well as all subsequent subgoals.

`spread` like `branch`, but applies no step to extraneous subgoals.

`repeat*` repeats a step along all sub-branches until the step does nothing.

`then` applies a series of steps in sequence; if a step branches into several subgoals, the remaining steps in the sequence are applied along all branches.

`try` applies a step and based on the result chooses to apply one or another step. The command `(try` s_1 s_2 s_3`)` first applies s_1; if s_1 is successful and generates subgoals then s_2 is applied to each subgoal; if s_1 does nothing then s_3 is applied to the current sequent. If s_1 fails, failure is propagated to the parent proof state.

Strategies can be invoked in the prover. For example, the command

```
(then (skolem!)  (repeat* (decompose-equality)) (grind))
```

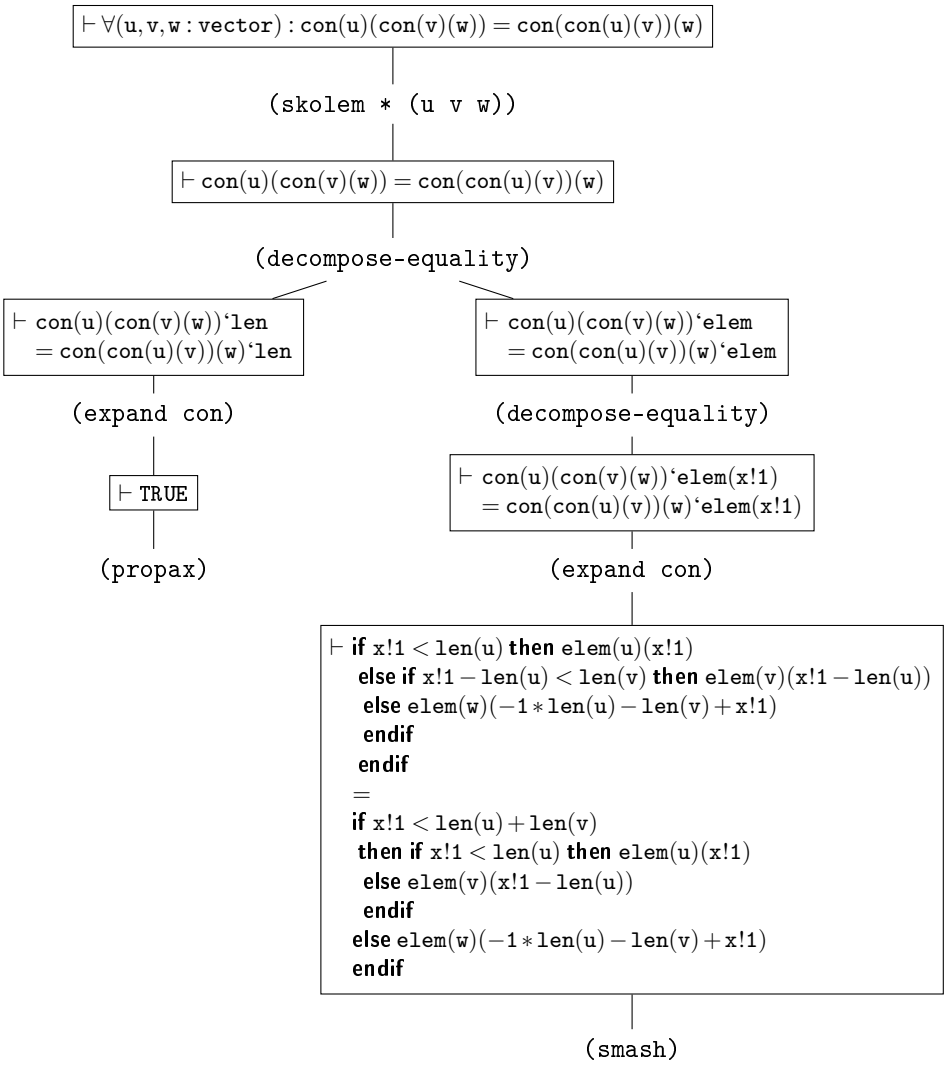


Figure 4.1: A PVS proof tree

discharges lemma `lem1` in one step. Additionally, PVS provides a Lisp macro called `defstep` for associating a name and a description with a strategy. Custom strategies should be defined in a Lisp file and stored together with the theories, or in a PVS package. These strategies may then be invoked via their names in the prover.

4.3.4 Proof scripts

The PVS prover records the proof scripts in separate text files, albeit in an internal format that is not intended to be human-readable nor easy to generate. A PVS extension, *ProofLite,* by Munoz et al. [121], allows proof scripts to be given as S-expressions inside the PVS theories. Using this notation we write the proof in Figure 4.1 as follows:

```
%|- lem1: proof
%|-   (skolem * (u v w))
%|-   (spread (decompose-equality)
%|-     ((expand con)
%|-      (then (decompose-equality)
%|-            (expand con)
%|-            (smash))))
%|- QED
```

ProofLite recognizes the character sequence `%|-` and translates the script into the internal proof format used by PVS (the `%` character normally starts a PVS comment that extends until end of line). In the remainder of the thesis we will present PVS proofs as ProofLite scripts.

4.4 Yices

Yices [69] is a freely available[4] SMT solver developed by SRI to support theorem proving, model checking, hardware verification and related domains. Yices decides the satisfiability—i.e., returns either `sat` or `unsat`—of first-order formulas containing uninterpreted functions with equality, linear real and integer arithmetic, scalar types, recursive datatypes, tuples, records, extensional arrays, fixed-size bit-vectors, quantifiers, and lambda expressions.

The implementation comprises a fast DPLL-based [57] SAT solver, a core theory solver, and a set of satellite solvers (for arithmetic, data types, arrays, etc.). The propositional structure of a formula is handled by SAT whereas (non-boolean) literals are handled by decision procedures for the corresponding theory fragment. The core solver uses congruence closure [126] to decide the satisfiability

[4] A binary can be downloaded from `http://yices.csl.sri.com` (as of June 2010).

of equations with uninterpreted functions, and communicates with the satellite solvers using an extension of the Nelson-Oppen procedure [125] for propagating equality between the solvers. The linear real arithmetic solver is based on the Simplex algorithm. Yices is implemented in C++, and is interfaced either via the *Yices input language* or through a C API.

The Yices input language supports quantifiers, but the decision procedure is incomplete if they are used. If quantifiers are used, Yices may additionally return `unknown`. Yices attemps to handle quantifiers with an E-graph matching-based [62] instantiation heuristic, complemented with Fourier-Motzkin elimination to simplify linear arithmetic expressions with quantifiers. It uses a specialized decision procedure for a fragment of universal quantification over array elements [47]; in this fragment it is possible to express, e.g., that a predicate is true for each element in an array, or that two arrays are equal on a common subrange.

Yices can be invoked directly from PVS as a decision procedure. A translation to the Yices input language has been implemented in PVS, but Yices is not, however, part of the PVS package and must be downloaded separately. If it is installed, the prover command `(yices)` invokes Yices as an *end-game strategy* in the PVS interactice prover. An end-game strategy either proves the sequent and terminates the branch, or does nothing. To check the validity of a sequent $\gamma_1, \ldots, \gamma_n \vdash \delta_1, \ldots, \delta_m$, the command checks the satisfiability of the formula

$$\gamma_1 \wedge \cdots \wedge \gamma_n \wedge \neg\delta_1 \wedge \cdots \wedge \neg\delta_m$$

using Yices. If the formula is unsatisfiable, the sequent is valid and is discharged. Otherwise, `(yices)` behaves as `(skip)`.

For large satisfiability problems, time and memory requirements of the algorithms implemented in Yices can grow beyond practical limits. In these cases it is possible to set an upper limit on the number of quantifier instantations, as well as a timeout such that the search terminates after a given number of seconds has elapsed.

4.5 Summary

PVS is a verification system comprising a specification language and an interactive theorem prover. Specifications are structured into theory hierarchies. Each theory contains a set of declarations. The basic declarations are type, variable, constant and formula declarations.

PVS is based on simply-typed higher-order logic. The base types are the built-in types, such as booleans and integers, and the uninterpreted types declared by the user. Type constructors include functions, tuples, records and abstract data types. Predicate subtypes and dependent types can be used to introduce arbitrary constraints on values. This makes typechecking undecidable in the general case; hence, PVS generates type correctness conditions (TCCs) during typechecking.

PVS proofs are based on sequent calculus. Formulas are proved in an interactive theorem prover. Each node and leaf in the proof tree is a sequent. The proof tree is extended by issuing commands to the prover. A proof is complete when each leaf in the tree has been proved. Strategies can be defined to automate the application of multiple proof commands.

ProofLite is a PVS package that allows proof scripts to be embedded within PVS theories in a human-readable form, and to be batch processed.

Yices is an SMT solver which is integrated as a decision procedure in PVS. The command `(yices)` either proves the current sequent, or does nothing.

Chapter 5

The Socos Language

This chapter defines the concrete invariant-based programming language used by Socos. We introduce a textual and a graphical syntax for the language.

5.1 Introduction

Socos provides a language for specifying sequential invariant-based programs. The language uses a subset of the PVS language. Expressions in the Socos language are PVS expressions, and the types of program variables are given as PVS types. The syntax and semantics of (type) expressions are well described in the PVS Language Reference [136], and hence, we do not go into the details of these here. Some of the syntactic apparel of PVS is carried over into the Socos language, such as the convention of enclosing blocks in **begin** . . . **end** brackets. The language combines the specification language of PVS with a concrete implementation language and can be considered a wide-spectrum language. In addition to standard program statements, such as assignments and tests, the language includes nondeterministic choice and nondeterministic assignment. The language has a basic procedure construct that allows decomposition of programs into subroutines.

We have deliberately kept the language small and simple. Only the minimal features needed to build procedural programs are included. The language does not include object-oriented constructs such as classes and inheritance, nor is there support for generic types.[1] Procedures are not values, and the parameter passing mechanism is restricted to pass by value or value-result. In particular, the language does not support pointers, pass by reference calls, a global heap or other mechanisms that may introduce variable *aliasing* (aliasing occurs when two separate identifiers are bound to the same mutable variable, so that updating via one identifier affects reading via the other).

The remainder of this chapter is structured as follows. First we give some

[1] While it is possible to build and reason about programs that use generic (uninterpreted) types, there is currently no support for instantiating them with concrete types.

conventions used throughout the chapter for defining the syntax of the language. We describe the lexical structure and the basic declarations. We then introduce the programming constructs in a top-down fashion, starting with contexts (the basic translation unit) and concluding with program statements. We end the chapter with a summary and a short discussion.

Notation. The language is presented in two dual notations: textual and diagrammatic. They are semantically equivalent; their relationship is the same as that between invariant diagrams and situation analysis [14, sec. 8]. The textual notation defines the nesting structure of situations but not the concrete graphical layout—i.e., the positions and sizes of the boxes. The two notations converge at the textual level: the syntax of all textual elements, such as constraints and statements, is identical in both notations.

The dual representation is pragmatic; textual notations are more readily handled in a syntax-directed context, and more amenable to automatic processing such as parsing and analysis. The textual notation is not intended to be used as a front-end programming language, although it is possible to do so. In Socos, the diagram representation of a program is converted into textual form as an intermediate step towards being processed into VCs. In the remainder of this thesis, we will present complete programs and examples as diagrams, while we use the textual notation in expositions that deal with fragments of the language, such as the verification semantics in the next chapter. The correspondence between the two notations is trivial, so this convention should cause no confusion.

We describe the syntax of the Socos language as a set of productions in BNF like form using the following conventions:

- Character literals are printed in `typewriter` font. `0x`H indicates an ASCII character with the hexadecimal value H.
- Reserved words and symbols are printed in **bold sans serif**.
- Syntactic variables are printed in *italics.*
- Productions are written in the form $P ::= S$.
- An optional part S of a clause is denoted $S^?$.
- Angle brackets are used to group clauses, as in $\langle S\ T \rangle^?$.
- For repetition of a clause S one or more times we write S^+; repetition zero or more times is denoted S^*. To indicate that the repeated elements are separated by a symbol , we write $S,^+$ and $S,^*$, respectively.
- Alternatives are separated with a bar, as in $S \mid T$.
- For elements with a diagrammatic representation, we give a graphical template to the right of the production.

5.2 Basic language structure

Below we briefly describe the basic structure of the language: lexical conventions, expressions, and constant and variable declarations.

5.2.1 Lexical conventions

The language follows the lexical conventions of PVS [136, Ch. 2]. Whitespace delimits tokens but is otherwise ignored, except in strings. Comments are introduced with the % character and extend until the end of the line. Numbers are nonempty sequences of digits. Strings are delimited with double quotes and may contain any ASCII character except the null character (the sequence `\"` is used to include a double quote in a string). Strings cannot contain comments. Identifiers are sequences that start with a letter and contain only letters, digits and the question mark character.

$$
\begin{array}{lll}
\textit{Number} & ::= & \langle \texttt{0} | \ldots | \texttt{9} \rangle^{+} \\
\textit{String} & ::= & \texttt{"} \ \langle \texttt{0x01} | \ldots | \texttt{0xFF} \rangle^{*} \ \texttt{"} \\
\textit{Id} & ::= & \langle \texttt{A} | \ldots | \texttt{Z} \mid \texttt{a} | \ldots | \texttt{z} \rangle \ \langle \texttt{A} | \ldots | \texttt{Z} \mid \texttt{a} | \ldots | \texttt{z} \mid \texttt{0} | \ldots | \texttt{9} \mid \texttt{?} \rangle^{*}
\end{array}
$$

Id above is the subset of PVS identifiers that do not contain the underscore character, which is reserved for identifiers generated by Socos to preclude name clashes with generated and user defined identifiers.

The structural constructs of the language introduce lexical scopes in which identifier bindings are restricted to one delimited subcomponent. Within a scope, Socos uses separate namespaces for the following categories of identifiers:

- constants and variables;
- types;
- contexts;
- procedures;
- situations.

Thus, it is possible to declare a constant with the same name as a situation within a single scope. Following the conventions of PVS, constants and variables may be overloaded provided that they can be disambiguated based on their types in the context where they appear. Identifiers in the other categories may not be overloaded.

5.2.2 Expressions and type expressions

There are two kinds of expressions, *Expr* and *TypeExpr*, corresponding to PVS expressions and type expressions, respectively. The exact syntax of these is documented in the PVS Language Reference [136]. *Expr* includes the basic numeric,

boolean and string expressions, as well as compound expressions such as operators, functions, binders and projections. *TypeExpr* includes the subtype, function, tuple and record type constructors.

5.2.3 Constant and program variable declarations

Constant declarations introduce new constants. Currently, non-recursive and recursive definitions are supported:

$$\mathit{Const} \quad ::= \quad \mathit{ConstDecl} \mid \mathit{RecursiveDecl}$$

These productions are the same as *ConstDecl* and *RecursiveDecl* of PVS, with the exception that constant names must be Socos *Id*s, and thus must not include the underscore character. Socos also does not currently support redefinition of PVS operators. This restriction is to simplify parsing, and may be lifted once the parser is improved.

The **pvar** keyword is used to introduce program variables. A declaration adding one or more program variables of the same type has the syntax:

$$\mathit{Pvar} \quad ::= \quad \textbf{pvar}\ \mathit{Id}^{,+} : \mathit{TypeExpr}\ ;$$

A program variable declaration has a strictly different interpretation than a PVS **var** declaration. The former adds a component to the state vector, whereas the latter binds a logical variable name to a type. Program variables may not be quantified over, and are the only identifiers that may appear on the left hand side of an assignment statement. The identifiers do, however, exist in the same namespace so it is an error to define a program variable with the same name as a logical variable in the same scope.

Constants and variables must be declared before they are used. However, in certain expressions identifiers of the form X_0 (where X is a program variable identifier) may be used. Such implicit *initial value-constants* are used to store the initial value of a mutable parameter of a procedure. Section 5.3.2 describes where they are available. In the typeset notation we render the initial-value constant X_0 as X_0.

5.3 Program constructs

We now describe the basic building blocks of Socos programs: contexts, situations, transition trees and statements.

5.3.1 Contexts

The context is the top level program construct and the basic translation unit. It defines a *verification context* for a set of procedures. A context declaration has the syntax:

Context ::= *Id* : **context**
begin
⟨**extending** Id^+;⟩?
$Importing^*$
⟨**using** ⟨*Id*|*String*⟩;⟩?
$Const^*$
$Procedure^*$
end *Id*

A context introduces a lexical scope: constants declared on the context level become available to all the contained procedures. A context may extend one or more other contexts with the **extending** keyword, inheriting their constants and procedures. Extension forms a hierarchy analogous to the PVS theory hierarchy, so extension declarations must not be cyclic.

Importing is identical to the PVS *Importing* clause and is used to import background theories, PVS theories containing definitions and lemmas pertaining to the verification of the procedures in the context. Importing a PVS theory makes its constants available in the context, and can also have an effect on the way VCs are handled, for instance if the theory contains auto-rewrite declarations [136, Sec 3.11]. We will use this mechanism in the case study in Chapter 8.

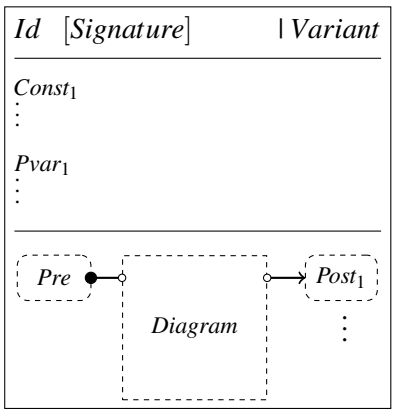
The **using** clause defines the proof command to be applied by default to the verification conditions. The argument is either one of the (transitively) extended contexts, in which case the command of that context is inherited; or it is a PVS proof command inside a string. If omitted, the command is inherited from the first extended context, or, if the context is not an extension, defaults to `(skip)`.

5.3.2 Procedures

The syntax for a procedure declaration is as follows:

Procedure ::=
Id [*Signature*] :
procedure
$Pre^?$
$Post^?$
⟨** *Variant* ;⟩?
begin
$Const^*$
$Pvar^*$

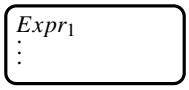
Diagram
end *Id*



$$
\begin{array}{lll}
\textit{Signature} & ::= & \langle \langle \textit{Id} : \textit{TypeExpr} \rangle^{,+} \rangle^{?} \\
 & & \langle \textbf{valres}\ \langle \textit{Id} : \textit{TypeExpr} \rangle^{,+} \rangle^{?} \\
 & & \langle \textbf{result}\ \langle \textit{Id} : \textit{TypeExpr} \rangle^{,+} \rangle^{?}
\end{array}
$$

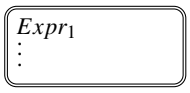
$$
\textit{Variant} \quad ::= \quad \textit{Expr}
$$

$$
\textit{Pre} \quad ::= \langle \textbf{pre}\ \textit{Expr}; \rangle^{+}
$$

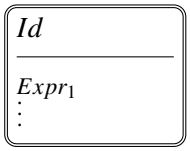


$$
\textit{Post} \quad ::= \textit{AnonPost}^{+} \mid \textit{NamedPost}^{+}
$$

$$
\textit{AnonPost} \quad ::= \quad \textbf{post}\ \textit{Expr};
$$



$$
\begin{array}{l}
\textit{NamedPost} \ ::= \\
\quad \textit{Id} : \langle \textbf{post}\ \textit{Expr}; \rangle^{+}
\end{array}
$$



$$
\textit{Diagram} \quad ::= \textit{Situation}^{*}\ \textit{Trs}^{?}
$$

The procedure construct is standard, with the addition that the specification allows multiple, labeled postconditions. The procedure executes in its own statespace, consisting of the mutable parameters and local variables. A *procedure call* is semantically equivalent to 1) assigning to each of the formal parameters the value of its corresponding actual parameter, 2) executing the body of the procedure, and 3) copying back the values of the mutable parameters to the caller, while discarding the values of local variables. Procedure calls are described in Section 5.3.4.

Each procedure in a context must have a unique name. The *signature* of a procedure is a list of its formal parameters, describing each parameter's *name*, *type* and *kind*. Parameter names should be distinct. The type of a parameter may be any PVS type, however the type may currently not be dependent on the value of another parameter (constraints on the parameters should be specified in the procedure precondition). The three parameter kinds are constant, value-result, and result. Constant parameters are used for passing values to a procedure: in a call, the actual constant parameters are evaluated and bound to the corresponding formal parameters. These bindings remain unchanged during execution of the procedure body, in contrast to the *value parameter* model, which allows the binding to be updated in the procedure body. An advantage of constant parameters is that they are a semantically simpler concept, and can be used meaningfully in postconditions. For value-result parameters, the values of the actual parameters are copied to the formal parameters, and when the procedure returns the final values of the formal parameters are copied back to the actual parameters. Result parameters are used for passing a store for the result only. Reference parameters are currently not

supported.

Variant, *Pre*, and *Post* constitute the specification part of the procedure declaration. A variant must be provided for (mutually) recursive procedures if termination is to be verified. The variant is an integer expression over the constant and value-result parameters. Its lower bound is implicitly zero. We describe how variants are used to prove termination of recursive procedures in the next chapter. The precondition of the procedure is the conjunction of all **pre**-prefixed predicates. Preconditions are over constants and value-result parameters. If *Pre* is omitted, the precondition defaults to `true`. A procedure may declare postconditions in three different ways:

- No *Post*. This is equivalent to a single anonymous postcondition `true`.
- A single anonymous postcondition.
- One or more uniquely labeled postconditions. The predicate associated with a label is the conjunction of the **post**-prefixed predicates appearing after the label.

A procedure with more than one postcondition is called a *multi-exit procedure*; typical use for multi-exit procedures is to distinguish between normal and exceptional returns. The postcondition predicates is over constants, parameters and *initial-value constants*. For each value-result parameter V, the initial-value constant `V_0` refers to the value of V prior to execution of the procedure body. This is a notational shorthand to avoid having to introduce a new identifier for this purpose.

The procedure body introduces a lexical scope containing constant and variable declarations, and an invariant diagram *Diagram*. *Diagram* is the actual procedure implementation, consisting of an initial *transition tree*, where execution of the procedure body starts, and a (possibly empty) set of situations. The syntax of transition trees will be described in Section 5.3.4.

We make some comments regarding the graphical notation for procedures. The outermost box represents a scope for the invariant diagram nested within, binding the parameters, constants and local variables. Due to their roles as initial and final situations, the pre- and postconditions are drawn as rectangular regions similar to situations, adjacent to or inside the procedure diagram (but never nested inside situations). Preconditions additionally have a slightly thicker outline and postconditions a double outline to set them apart from the intermediate situations. The initial transition tree of a procedure is drawn from the precondition, extending into the diagram. Exit transitions (defined in Section 5.3.4) are drawn as arrows to the postconditions. If the pre- or postcondition is omitted, the frame of the procedure box is used to anchor these transition arrows.

Figure 5.1 shows an example specification of a procedure called `find` for determining whether an integer element `x` exists in the constant vector `a`, and in the positive case also returning an index containing `x`. The procedure has two

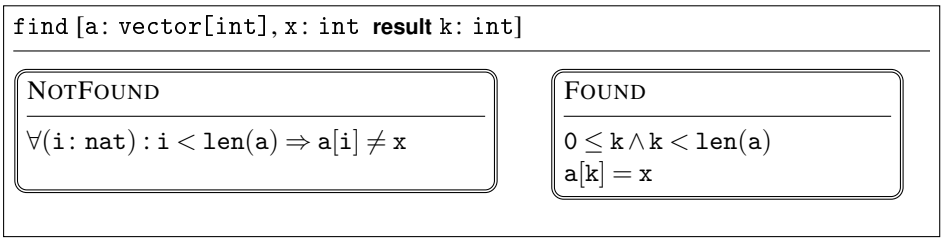


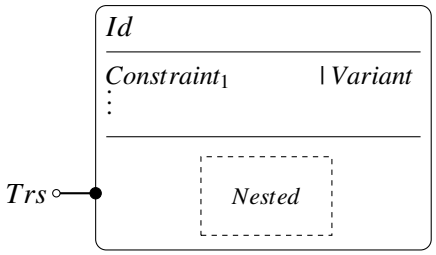
Figure 5.1: Specification of procedure `find`

postconditions: NOTFOUND and FOUND. If `x` is present in `a`, the procedure implementation is supposed to exit in FOUND after having set `k` to some index that contains `x`; if `x` is not present in `a`, the implementation should exit in NOTFOUND.

5.3.3 Situations

The syntax of situations is as follows:

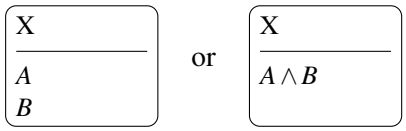
Situation ::=
Id : **situation**
begin
⟨∗ *Constraint* ;⟩∗
⟨∗∗ *Variant* ;⟩?
Nested
Trs?
end *Id*



Nested ::= *Situation*∗
Constraint ::= *Expr*

Situations within a procedure must be distinctly named. Each situation contains a (possibly empty) sequence of *constraints*, an optional variant, a (possibly empty) sequence of nested situations, and an optional transition tree.

A situation corresponds to a named predicate over the state vector of the program variables in the containing procedure. Each constraint corresponds to a data invariant over the state vector. Constraints must be boolean expressions over the program variables. The predicate of a top level situation is the conjunction of its sequence of constraints; the predicate of a nested situation is the conjunction of its sequence of constraints and the predicate of its enclosing situation. Writing constraints as separate clauses has no semantic significance, but provides a way of decomposing verification of transitions to the situation. Each constraint becomes a separate proof goal. For example, declaring a situation as either



associates the same predicate with X, but the the first case gives rise to two separate proof goals, A and B, while in the second case we get a single proof goal, $A \wedge B$. If no constraint is given, the situation predicate is `true`. The variant, if given, should be an integer function from the state vector. The lower bound of the variant is assumed to be zero.

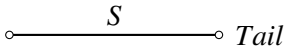
Execution from a situation starts at the transition tree Trs. A situation that can be reached from the initial transition tree is *reachable*. A reachable situation without a transition tree is a final situation.

5.3.4 Transition trees

A transition tree is a tree in which each leaf is either a **goto** or an **exit** command, and each intermediate node is either a **choice** command, an **if** command, or a **call** command. An edge in the tree may additionally be labeled with a statement S; the syntax of statements is given in the next section. A transition is a unique path from the root to a leaf. The syntax of transition trees is:

Trs ::= $\langle S\,; \rangle^?\, Tail$

$Tail$ ::= $Goto | Exit | Choice | If | Call$

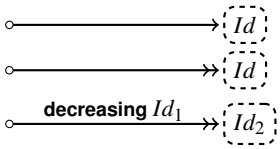


$Goto$ ::=

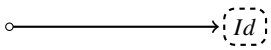
goto Id

| **decreasing goto** Id

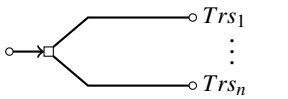
| **decreasing** Id_1 **goto** Id_2



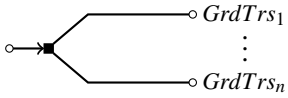
$Exit$ ::= **exit** $Id^?$



$Choice$ ::= **choice**
Trs^+
endchoice

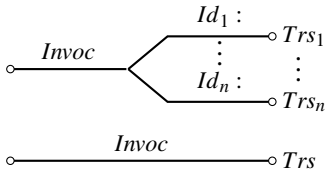


If ::= **if**
$GrdTrs^+$
endif



$GrdTrs$ ::= $[Expr]\,;\, Trs$

$$
\begin{array}{lcl}
\mathit{Call} & ::= & \textbf{call}\ \mathit{Invoc} \\
& & \quad \langle \mathit{Id}\ :\ \mathit{Trs} \rangle^{+} \\
& & \textbf{endcall} \\
& | & \mathit{Invoc};\ \mathit{Trs}
\end{array}
$$



$$\mathit{Invoc} ::= \mathit{Id}(\langle \mathit{Expr}^{,+}, \rangle^{?} \langle \mathit{Id}^{,+} \rangle^{?})$$

goto and **exit** identify the targets of the individual transitions. The former jumps to an intermediate situation in the procedure, while the latter exits to a postcondition. In the textual syntax we identify the target by name, while in the diagrams we indicate the target by drawing the transition arrow to the edge of the target situation. The keyword **exit** must be given without a parameter in procedures with a single, unnamed postcondition. For procedures with named postcondition, the parameter should identify a postcondition. In the diagrammatic notation we draw an exit transition to the edge of a postcondition.

The keyword **decreasing** is used to declare that a transition decreases the variant of the situation given as parameter. This mechanism defines cutpoints for verifying that transition cycles terminate; we describe how it is used in Section 6.5 in the next chapter. **decreasing** can also be given without a situation identifier, in which case the variant is the target situation. All transitions in a tree sharing a common target situation must supply identical parameters to **decreasing**. This requirement ensures that all transitions between the same two situations decrease a common variant, which is required for the termination verification method to be sound. In diagrams, we draw transitions marked as **decreasing** using a double arrowhead.

choice is nondeterministic choice, while **if** is a guarded choice. The former nondeterministically picks one of the branches for execution. The latter nondeterministically picks a branch whose guard is true for execution, and aborts if all guards are false. In the diagrammatic notation the type of choice is indicated at the branching point with a white or a black square, respectively.

Figure 5.2 shows a linear search implementation of the procedure `find`, containing the situation LOOP and two transition trees. The transition tree from LOOP contains two **if** branches. The transition back to the intermediate situation LOOP has been indicated to decrease the variant of LOOP.

call is a procedure call. The construct consists of an invocation clause, which identifies the procedure to be called and supplies the actual parameters, and a branching part, which connects the exits of the procedure to transition trees. The second version of procedure call omits the branching part and applies only to procedures with a single unlabeled postcondition. The invocation clause should provide actual parameters to match the signature of the called procedure in the standard positional way. For each formal constant parameter, an expression of the

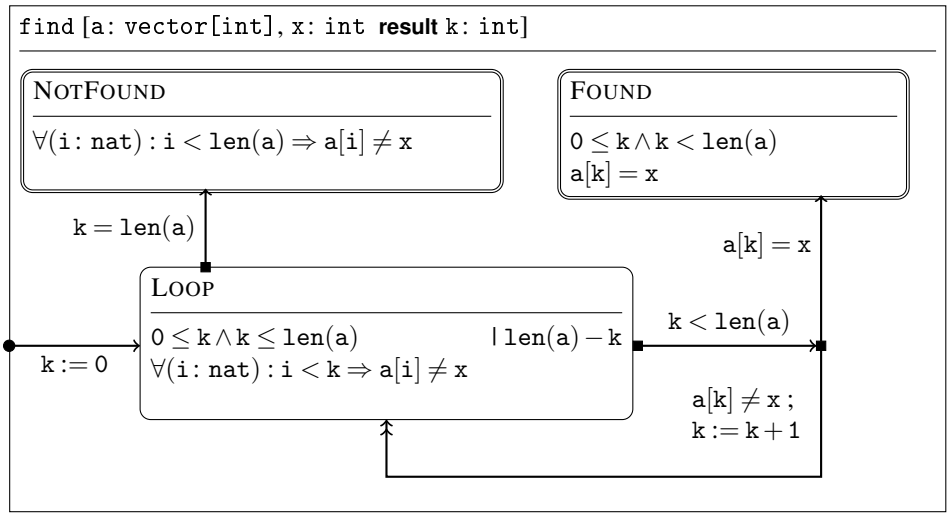
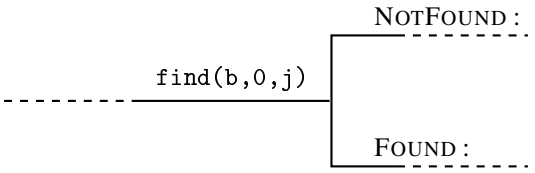


Figure 5.2: Implementation of procedure `find`

same type or a subtype must be supplied. For each formal value-result parameter, a program variable of the same type must be supplied. For each formal result parameter, a program variable of the same type or a supertype must be supplied. The branching part should include at most one branch for each postcondition in the called procedure. For example, a call of the procedure `find` with actual parameters `b`, `0` and `j` is drawn as follows:



To be considered live, a call should provide a transition tree for each named postcondition (this is described further in Section 6.6 in the next chapter).

5.3.5 Statements

The language includes five distinct single-exit program statements: *assume*, *assert*, *assignment*, *havoc*, and *sequential composition.* The syntax of statements is as follows:

$$
\begin{array}{lcll}
S & ::= & [Expr] & \text{(assume)} \\
 & | & \{Expr\} & \text{(assert)} \\
 & | & Id^{,+} := Expr^{,+} & \text{(assignment)} \\
 & | & Id^{,+} := ? & \text{(havoc)} \\
 & | & S_1 ; S_2 & \text{(sequential composition)}
\end{array}
$$

Assume and assert both require the argument to be a predicate. Assume is a specification statement; it is *enabled*—i.e., executable—only from states satisfying the predicate. From such states execution of the assume statement succeeds without changing the state. Assume blocks if the predicate is false. Assert evaluates its argument; if true, it succeeds without changing the state; if false, the program fails.

The assignment statement updates a list of values to a list of program variables. An assignment statement is well-formed only if the number of left hand side identifiers is equal to the number of right hand side expressions, and every element on the left hand side is a program variable identifier. The same variable may occur multiple times on the left hand side. The operational interpretation of an assignment $V_1, \ldots, V_n := E_1, \ldots, E_n$ is that the expressions $E_1, \ldots, E_n$ are first evaluated, and then the result values are assigned to $V_1, \ldots, V_n$ in left to right order. Havoc assigns each variable in the list nondeterministically an arbitrary value from the type of the variable. Sequential composition of two statements executes the first statement followed by the second statement.

5.4 Summary and discussion

We have in this chapter defined dual textual and diagrammatic syntaxes for the concrete invariant-based programming language used by the Socos tool. The language is intended to have a simple verification semantics and so excludes many features that would be expected from a full fledged implementation language, such as global and reference variables, object orientation, and pointers.

The top level construct defines the verification context (constants, background theories, default VC discharge command) for a set of procedures. Each procedure is specified by a contract consisting of a precondition and one or more postconditions. The implementation of a procedure is given as an invariant diagram with nested situations connected by transitions. Invariants are predicates over a state vector consisting of the procedure's mutable parameters and local variables. A transition tree is a multi-exit statement rooted in a situation and with each leaf ending in a situation or a postcondition. The commands **choice**, **if** and **call** add branching points to the transition tree. Statements are labels on the tree edges.

Program variables can range over any type that can be specified in PVS and expressions are PVS expressions, allowing the full power of the specification language to be used. This gives a large degree of freedom in choosing the level of abstraction for programs. For instance, if we are verifying a program over integers on a general algorithmic level we may want to use the mathematical (unbounded) `int` type, while if we are checking a program near machine level, we may want to use a bounded integer type corresponding to a specific word size. A trade-off for the generality is that every Socos program cannot be considered executable as such. To ensure that a program can be executed, we must check that all program variables can be represented in the execution environment and that all expressions occurring

in statements can be evaluated in finite time. There is currently no support for such checking in Socos. A more precise language definition should restrict expressions in statements to a subset of the PVS language that can be evaluated. Such a subset has been identified for the PVS ground evaluator [145], which translates PVS expressions into Common Lisp. The ground evaluator also provides a way of defining custom translations to Lisp called *semantic attachments* for terms that are not evaluable in their general form.

A similar approach was taken in Socos_1 for translation of invariant-based programs into Python [21]. Socos_1 provides a mechanism for defining Python evaluators for operators that are not generally executable—such as quantifiers—but for which meaningful special cases can be identified—such as quantification over an integer range. However, there is no unified semantics of the types used for reasoning about the program and the runtime Python types. For Socos_2, we are considering building an execution environment based on the PVS ground evaluator; this is, however, future work.

Chapter 6

Verification Methodology

This chapter discusses the mechanics of establishing correctness of an invariant diagram and gives a basis for VC generation. We describe the consistency, liveness and termination conditions on invariant diagrams and how Socos derives them.

6.1 Introduction

Chapter 2 described three types of correctness conditions associated with invariant diagrams: consistency, liveness and termination. This chapter goes into the semantic characterization of these conditions for the language of Chapter 5 and presents a concrete method for verifying that a program satisfies them. The goal is to define a verification method that is sound—an incorrect program should not be allowed to pass the verification process—and "adequately" complete—a sufficiently large class of programs should be verifiable. The methodology should support incremental program development. Building a correct program in one step is infeasible. It must be possible to develop correct programs incrementally, such as by verifying one transition before adding the next, or strengthening an existing invariant-based on the feedback from the verification process. It should also be possible to postpone verification of termination and liveness until the basic consistency of the program has been established. The tool should thus provide facilities for checking programs at various "degrees of partial correctness".

Verification in Socos is based on checking each procedure separately. Hence the VCs are interpreted in the context of a single statespace corresponding to the procedure currently being verified. We use the standard technique of abstracting procedure calls into assert and assume statements based on the specification.

The remainder of the chapter is organized as follows. Section 6.2 introduces the notations used in this and the next chapter. Sections 6.3–6.6 define a semantic function based on weakest preconditions for translating invariant-based programs into VCs. The chapter ends with a summary and discussion of related work.

6.2 Notation

Rewrite rules. We express the translation of Socos programs into VCs as collections of rewrite rules over fragments of the language. We give the rules such that the transformation is terminating and deterministic.

The rule $P \xrightarrow{R} T$ denotes that the result of applying translation R to a term matching the pattern P produces the term T. The pattern P is a binder for a set of syntactic variables, which become available in the right hand side T. We denote with S^R the term produced by applying rule R to term S.

Many rules deal with lists of terms. We use interchangeably the notations T and $T_1, \ldots$ and $T_1, \ldots, T_n$ for a list of terms, picking the notation that best suits the exposition. If T and T' are lists, then T, T' stands for the term list obtained by concatenating T and T'.

We use the wildcard symbol $[\ldots]$ in patterns to match uninteresting sequences of subterms. For instance, the pattern "$P\ [\ldots]$" matches any term starting with a subterm matching P.

Environment. This chapter describes the verification of a single invariant diagram in the environment established by the containing procedure. The environment consists of a state vector σ and the following injective mappings from identifiers to terms:

- J_X stands for the predicate of situation X, i.e., the conjunction of the constraints of X and the predicate of its parent situation;
- V_X stands for the variant of situation X;
- $\mathsf{P}_P(C, V)$ gives the precondition of procedure P applied to constant parameters C, value-result parameters V;
- $\mathsf{Q}_{P,X}(C, V\_0, V, R)$ gives the postcondition of procedure P labeled X applied to constant parameters C, initial values V_0, value-result parameters V and result parameters R;
- $\mathsf{W}(C, V)$ stands for the variant of the current procedure (set of mutually recursive procedures);
- T_X stands for the type assigned to program variable X.

Additionally, we write P_0 for the precondition of the current procedure, $\mathsf{Q}_{0,0}$ for the default postcondition of the current procedure, $\mathsf{Q}_{0,X}$ for the labeled postcondition X in the current procedure, and $\mathsf{Q}_{P,0}$ for the default postcondition of procedure P. The concrete details of these mappings are revealed in Chapter 7 where we define the translation of Socos programs into PVS theories.

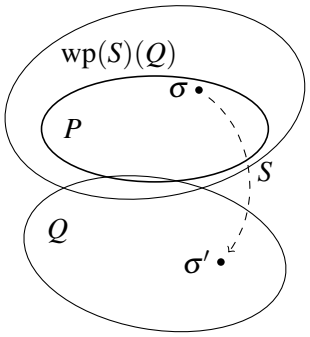


Figure 6.1: Weakest precondition

6.3 Consistency

Back [16, Sec. 4] defines consistency of an invariant diagram as the property that "each transition establishes its target when executed from its starting situation." Operationally, consistency of a transition from situation A to situation B means that every execution trace from a state satisfying J_A should *not fail*, and should terminate in a state satisfying J_B. A transition fails if an assert statement evaluates to false. We make a strict distinction between failure and nontermination: a statement always either fails or terminates. Diagrams with loops and transitions containing recursive procedure calls, on the other hand, may exhibit nontermination. When verifying consistency we only demonstrate the absence of failure. Thus, consistency can be seen as a partial correctness assertion.

Socos uses weakest preconditions as the basis for generating VCs. The weakest precondition of a statement S, denoted $\mathrm{wp}(S)$, is a predicate transformer that for a predicate Q gives the most general (weakest) precondition under which S is guaranteed to establish Q. In other words, execution of S in a state σ such that $\mathrm{wp}(S)(Q)(\sigma)$ holds terminates in a state σ' such that $Q(\sigma')$ holds. This is illustrated in the set diagram of Figure 6.1. A single-exit statement S is consistent if and only if its precondition P is at least as strong as $\mathrm{wp}(S)(Q)$:

$$\forall \sigma \bullet P(\sigma) \implies \mathrm{wp}(S)(Q)(\sigma) \tag{6.1}$$

Formula 6.1 is the familiar correctness condition for Hoare triples of the form $\{P\}\,S\,\{Q\}$. The weakest preconditions for the statements in Section 5.3.5 are the standard ones; we postpone their definitions until the next chapter, where they will be treated in conjunction with the reduction of VCs into proof goals.

A transition tree is consistent if and only if its constituent transitions are all consistent. We recall that a transition tree is a multi-exit statement with intermediate nodes of type *Choice*, *If* or *Call*, and leaves of type *Goto* or *Exit*. Of the intermediate node types, we consider here only *Choice*, as *If* and *Call* will be defined in terms of *Choice* and other primitive statements in sections 6.4 and 6.6,

respectively. We also consider only *Goto* leaves; *Exit* is treated in Section 6.6.

Extending Formula 6.1 to transitions trees, we introduce a semantic translation rule cc (for *consistency condition*) such that a transition tree Trs is consistent from situation X if:

$$\forall \sigma \bullet \mathsf{J}_X(\sigma) \implies Trs^{\mathsf{cc}}(\sigma) \tag{6.2}$$

Hence, cc should give a weakest precondition for every transition in the tree Trs to establish its target. We define cc for **goto** and **choice** as follows:

$$S \,;\, \textbf{goto}\ X \quad \xrightarrow{\mathsf{cc}} \quad \mathrm{wp}(S)(\mathsf{J}_X)$$

$$S \,;\, \textbf{choice}\ Trs_1 \ldots Trs_n\ \textbf{endchoice} \quad \xrightarrow{\mathsf{cc}} \quad \mathrm{wp}(S)(Trs_1^{\mathsf{cc}} \cap \cdots \cap Trs_n^{\mathsf{cc}})$$

The first case establishes the connection between situations and postconditions, while the second case expresses the notion that a transition tree is consistent if and only if each subtree is consistent. When the statement S is absent from the left hand side, we have the simpler condition:

$$\textbf{goto}\ X \quad \xrightarrow{\mathsf{cc}} \quad \mathsf{J}_X$$

$$\textbf{choice}\ Trs_1 \ldots Trs_n\ \textbf{endchoice} \quad \xrightarrow{\mathsf{cc}} \quad (Trs_1^{\mathsf{cc}} \cap \cdots \cap Trs_n^{\mathsf{cc}})$$

Formula 6.2 is the basic template for generating VCs. Socos generates exactly one condition of this form for each situation that has a transition tree, and one for the initial transition of the procedure. Showing that a procedure is consistent means proving that each condition is true. This method allows for a concise way of building VCs generation that match closely the transition tree from which they were generated, and has the advantage that the weakest precondition for each statement is calculated only once. However, since Formula 6.2 describes the correctness of a transition tree rather than individual transitions, it is not suitable for sending to a theorem prover as such. Reduction of Formula 6.2 into smaller proof goals is done with PVS proof commands as described in the next chapter.

6.4 Liveness

Liveness means that execution of the diagram, if terminating, terminates in one of the intended final situations, i.e., in one of the postconditions. A procedure is live if the following conditions both hold: 1) there is an initial transition tree, and from every situation that is reachable from the initial transition tree at least one of the postconditions is reachable; 2) for every reachable transition, each command in the transition is able to proceed from any state it may be reached by. Condition (1) ensures that the program does not get stuck in an intermediate situation. Condition (2) ensures that the execution of a transition can run to completion and proceed to the target situation.

Condition (1) is checked by reachability analysis of the program graph. To establish condition (2), the transition statements must be checked for enabledness. In general, for a statement to be enabled in any state it must satisfy Dijkstra's "excluded miracle" law [66]:

$$\forall \sigma \bullet \neg \mathrm{wp}(S)(\emptyset)(\sigma)$$

Here the symbol $\emptyset$ denotes the universally false predicate (empty set). A statement that does not satisfy the above is called *partial*. Thus, for a partial statement S there exists some state σ' from which S cannot progress to any state. The behavior of a partial statement in such states is sometimes interpreted as "invoking a miracle." Of the statements given in Section 5.3.5, only the assume statement is partial; more precisely, the assume statement $[G]$ is enabled only in those states in which G is true, so programs that contain assume statements are not necessarily live.

Socos analyzes programs syntactically for liveness and prints a warning that the program may not be live if condition (1) is not true, or if an assume statement is used in a reachable transition. Programs which are not live may still be verified for consistency, as lack of liveness is not necessarily an error but rather an incompleteness in the program. The incompleteness may be deliberate, e.g., the checked program is a stepping stone towards a final, live version. On the other hand, it may also be due to a genuine mistake. In either case the tool points out to the programmer is this really what is desired.

When liveness is desired, **if** rather than **choice** should be used to enclose guarded transitions. It has the following consistency condition:

$$\begin{array}{l} S\,;\ \textbf{if} \\ \quad [G_1];Trs_1 \\ \quad \vdots \\ \quad [G_n];Trs_n \\ \textbf{endif} \end{array} \quad \xrightarrow{\text{cc}} \quad \begin{array}{l} \langle\, S;\{G_1 \vee \cdots \vee G_n\};\ \textbf{choice} \\ \quad [G_1];Trs_1 \\ \quad \vdots \\ \quad [G_n];Trs_n \\ \textbf{endchoice}\,\rangle^{\text{cc}} \end{array}$$

The collection of guarded transitions is translated into an assertion of the disjunction of the guards followed by a **choice**. The assertion is placed just before the choice, and hence the program aborts if all guards are false; this results in a consistency condition that at least one guard is enabled being generated. We note that the terms $[G_1] \ldots [G_n]$ on the left hand side are transition guards, whereas on the right hand side they are assume statements.

In summary, to ensure that a program is live the programmer should 1) ensure that a postcondition can be reached from every reachable situation, 2) avoid assume statements; and 3) use **if** commands for conditional transitions. During program development, assume statements are frequently used to collect explicit assumptions at specific points in a transition tree, since these assumptions may simplify the proof search for the automatic prover. However, Socos does not address the problem verifying that a program containing assume statements is live. We note

in this respect that an assume statement whose precondition is sufficiently strong that it can be proved (automatically) to always be true, can be replaced with an assert statement. Thus, requirement (2) does not restrict the set of live programs that can be verified. The effect of (3) is that transition trees are instrumented with assertions such that the consistency condition is provable only if at least one guarded transition is enabled. A program that does not satisfy these liveness conditions may still be checked for consistency.

6.5 Termination

Nontermination in invariant diagrams can be due to either unbounded iteration through cycles in the transition graph, or infinitely recursive procedure calls. We describe iteration here; recursion is treated in the next section in the context of procedures and procedure calls. Methodologically, termination is handled similarly to liveness in the sense that generation of termination conditions is optional. Socos analyzes the program graph to determine if termination conditions should be generated. If termination is to be checked, Socos instruments the transitions that are part of a cycle with assertions, such that their consistency implies termination. If termination is not to be checked but the program contains cycles, Socos warns that the program may not be terminating.

To ensure that an invariant diagram is terminating, we must prove that no situation in the diagram is visited often; i.e., that the transition relation is well-founded. This can achieved by showing that every cycle in the transition graph strictly decreases a variant, which is bounded from below. A straightforward method is to generate a verification condition for each cycle in the diagram. However, this is unappealing since it violates the locality principle of IBP by requiring reasoning about entire cycles rather than individual transitions. Instead, the following method which is based on identifying the *strongly connected components* in the diagram is used. A graph is strongly connected if there is a path between any pair of nodes. The strongly connected components of a graph is the disjoint set of its maximal, strongly connected subgraphs. If each strongly connected component is reduced into a node while maintaining the edges between components, the resulting graph is acyclic. We can hence decompose termination of the program into termination of its individual strongly connected components.

In Socos_1 termination is proved by showing that each cycle in a strongly connected component is cut by a transition that decreases the variant [22]. It requires the following constraints to be satisfied:

1. All situations in a strongly connected component share a common variant.
2. Every cycle in the component is cut by a transition that *strictly decreases* the variant.
3. No other transition in the component increases the variant.

The programmer marks a set of transitions, and the tool generates VCs that each marked transition decreases the variant, and additionally that each unmarked transition in the same component does not increase the variant. We note that this method is less general than reasoning about entire cycles, since it does not allow a variant to be increased in any transition, even if the variant is necessarily decreased by the cycle as a whole.

While the method works for simple programs, conditions (1)–(3) are too strong to handle some common cases. In particular, condition (1) does not allow for decomposition of termination proofs within a strongly connected component. This means that nested loops must share a common variant, whereas it is often more natural to give separate variants for the outer and inner loops. Examples of invariant diagrams with multiple variants can be found in, e.g., [14, 15] and [16]; we consider here the simplified example in Figure 6.2.

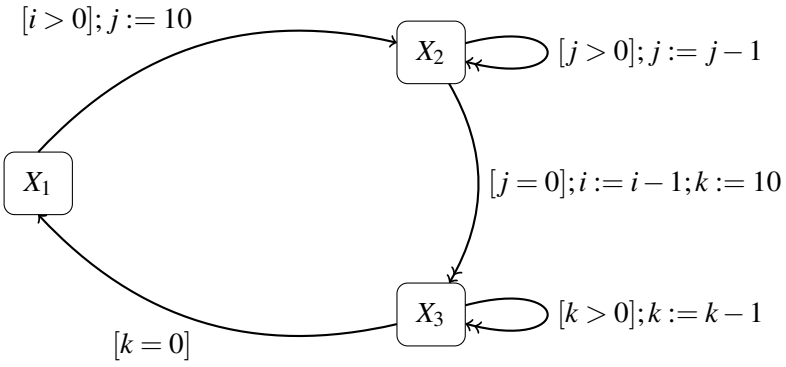


Figure 6.2: A terminating program

The program comprises three loops, each one decreasing the counters i , j and k, respectively. It is appealing to assign variant i to the outer loop involving all situations, and variants j and k respectively to the two inner loops involving single situations. While we can easily formulate a common variant (say $i + j + k$) that is decreased by each of the double-headed arrows (condition (2)), finding one that additionally satisfies condition (3) for the transition between situations X_1 and X_2 seems hopeless.

We generalize the above method slightly by allowing each situation in a strongly connected component to declare a separate variant. The basic idea is that a component is split into subcomponents which are separately proved to terminate using conditions (1)–(3). However, since variants can interfere with each other—decreasing one may increase another—this decomposition must not be arbitrary.

The extended verification method is based on the following observation. Suppose that in a strongly connected component there is a situation X such that every cycle through X strictly decreases the associated variant V_X. As V_X is bounded

from below, this precludes all computations where the situation X or any of the transitions that decrease V_X occur infinitely often. Having established this, we can reduce the problem of showing that the component terminates by deleting all transitions that strictly decrease V_X, and again decomposing the resulting graph into its strongly connected components. This effectively prunes X along with any other situation that can reach itself *only* by decreasing V_X, since such situations are no longer part of a cycle (the cycles have been cut). We then re-iterate the method on the resulting strongly connected components. If the original component can be reduced completely using this method, no situation may occur infinitely often, and thus the component is terminating.

We describe in the following a recursive algorithm to determine based on the given **decreasing** assertions if a decomposition is possible, and additionally to mark transitions with **nonincreasing** assertions. The latter is an internal marking of transitions to record the variants that may not be increased; they are not part of the language and the programmer cannot add these assertions explicitly.

1. Partition the diagram into its strongly connected components, discarding situations that are not involved in a cycle as well as transitions between components.

2. For each strongly connected component C:

   (a) Select a situation X in C, such that each cycle from X back to X is cut by a **decreasing** X transition. If there is no situation satisfying this criterion, signal failure.

   (b) Annotate all transitions in C not marked by **decreasing** X with **nonincreasing** X.

   (c) Remove all transitions marked as **decreasing** X from C, and apply this algorithm recursively on the resulting graph.

We note that the using a common variant for each component is a base case of the algorithm. We also note that there may be several acceptable choices in step (2a); the algorithm is nondeterministic in this case. This is handled pragmatically by processing situations in the order they are declared, making the selection deterministic. A more general option that could be considered is to build all possible **nonincreasing** markings and generate a VC corresponding to their disjunction. However, this would add complexity to the conditions by introducing reasoning about multiple transitions in the same condition, violating the locality principle of IBP.

Figure 6.3 illustrates the decomposition of the program in Figure 6.2, annotated with the variants suggested above and with **decreasing** assertions added. The first step of the above algorithm selects and prunes X_1, which satisfies the cycle cutting criterion, and annotates every transition in the diagram with **nonincreasing** X_1.

The remaining two components consist each of a single situation (X_2 and X_3, respectively) and are base cases.

The generation of termination conditions is directed by the presence of **decreasing** assertions: if no **decreasing** assertions are present in a strongly connected component, Socos shows a warning that the component may not be terminating. If at least one **decreasing** assertion is present, the above algorithm is used to look for a decomposition. If none can be found, Socos reports an error. Otherwise, the transitions are marked with **nonincreasing** as described above, and additionally each transition tree in the component is prefixed with the assignment statement

$$X_1\_0, \ldots, X_n\_0 := \mathsf{V}_{X_1}, \ldots, \mathsf{V}_{X_n}$$

where each situation X_i appears in at least one **decreasing** or **nonincreasing** clause in the transition tree, and X_i_0 is an integer variable recording the value of the variant at the root of the transition tree. The following correctness condition is then used for the marked transitions

$$\begin{array}{lll} S; & \textbf{decreasing } X & \\ & \textbf{nonincreasing } Y_1, \ldots, Y_n & \xrightarrow{\text{cc}} \\ & \textbf{goto } Z & \end{array} \qquad \begin{array}{ll} \langle\ S; & \{0 \le \mathsf{V}_X < X\_0\}; \\ & \{0 \le \mathsf{V}_{Y_1} \le Y_1\_0\}; \\ & \vdots \\ & \{0 \le \mathsf{V}_{Y_n} \le Y_n\_0\}; \\ & \textbf{goto } Z \quad \rangle^{\text{cc}} \end{array}$$

to ensure that the variant at the end of a transition is bounded from below by zero and less than/less than or equal to the initial value.

6.6 Procedures

The verification method for procedures is based on standard techniques, so we will only give a brief overview here. Execution of a procedure body starts in an initial situation satisfying the precondition, and ends in a final situation satisfying one of the postconditions. A procedure call is verified using the pre/postcondition abstraction for the procedure in the usual way. The trickiest part is properly handling the substitution of the actual parameters into the specification; a good exposition of the technique is given in [84, Ch. 12]. The method implemented in Socos is the same, but with the restriction that mutable value parameters and reference (var) parameters are not supported, and with the (straightforward) extension to procedures with multiple postconditions and multi-exit procedure calls. We also address termination of recursive calls.

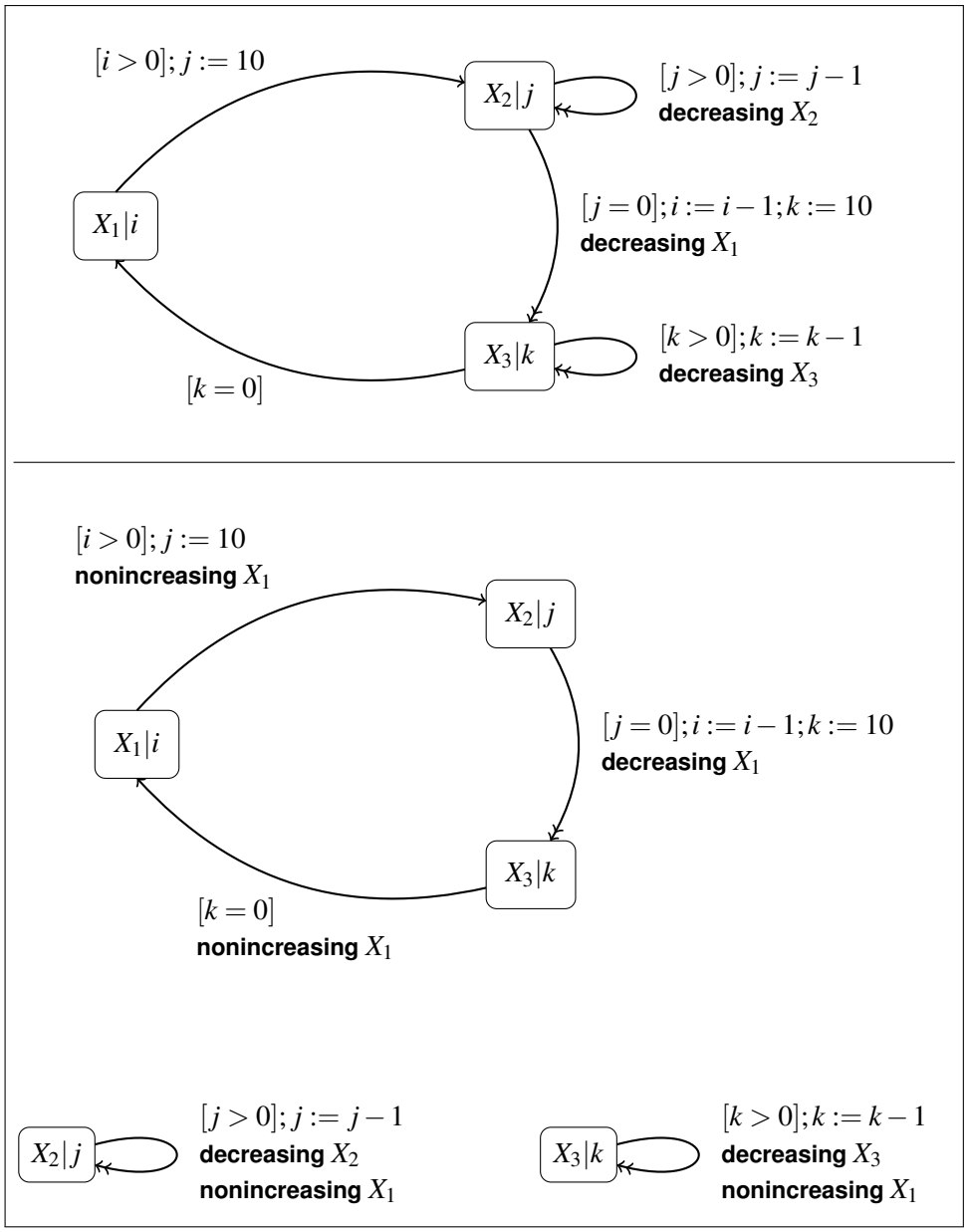


Figure 6.3: The **decreasing**-marked program of Figure 6.2 (above) and its decomposition (below)

6.6.1 Procedure body verification

The initial and final situations are derived from the pre- and postconditions of the current procedure in the following way. The state vector of the procedure consists of the formal value-result and result parameters and the local variables; constant parameters, being immutable, are not included in the state vector. Let C, V and R be the formal parameters of kinds constant, value-result and result, respectively. Let further $L_1, \ldots, L_m$ be the postcondition labels. We recall from Section 5.3.2 that C, V and R are pairwise disjoint; and that in addition to global constants, precondition P_0 may only contain identifiers from C and V whereas postconditions $\mathsf{Q}_{0,0}, \mathsf{Q}_{0,L_1}, \ldots, \mathsf{Q}_{0,L_m}$ may contain identifiers from C, V, R and the initial value constants $V_1\_0, \ldots, V_n\_0$. Initially the precondition is assumed to be true, V_i is equal to V_i_0 for each $1 \leq i \leq n$, and nothing is known about the values of result parameters or local variables. We define a set of indexed final situations, in each of which the corresponding postcondition holds. The initial and final situations are thus characterized by the predicates:

$$
\begin{array}{lcl}
\mathrm{I} & = & \mathsf{P}_0(C,V) \cap V_1 = V_1\_0 \cap \cdots \cap V_n = V_n\_0 \\
\mathsf{F} & = & \mathsf{Q}_{0,0}(C, V\_0, V, R) \\
\mathsf{F}_{L_1} & = & \mathsf{Q}_{0,L_1}(C, V\_0, V, R) \\
\vdots & & \\
\mathsf{F}_{L_m} & = & \mathsf{Q}_{0,L_m}(C, V\_0, V, R)
\end{array}
$$

And consequently we have the following consistency conditions for *Exit* leaves:

$$
\begin{array}{rcl}
\textbf{exit} & \xrightarrow{\mathsf{cc}} & \mathsf{F} \\
S\ ;\ \textbf{exit} & \xrightarrow{\mathsf{cc}} & \mathrm{wp}(S)(\mathsf{F}) \\
\textbf{exit}\ X & \xrightarrow{\mathsf{cc}} & \mathsf{F}_X \\
S\ ;\ \textbf{exit}\ X & \xrightarrow{\mathsf{cc}} & \mathrm{wp}(S)(\mathsf{F}_X)
\end{array}
$$

6.6.2 Procedure call verification

We now consider a call of procedure P with the actual parameters E, X and Y supplied for the formal constant parameters C, value-result parameters V and result parameters R respectively. E is a list of expressions, while X and Y are variable identifier lists with potentially common members. The call is non-recursive if the call graph starting from P does not include the current procedure; it is recursive if the call graph contains the procedure. The consistency condition for the non-

recursive procedure call is:

$$\begin{array}{l} \textbf{call } P(E,X,Y) \\ \quad L_1 : Trs_1 \\ \quad \vdots \\ \quad L_n : Trs_n \\ \textbf{endcall} \end{array} \xrightarrow{\text{cc}} \begin{array}{l} \langle \quad \{\mathsf{P}_P(E,X)\}; X',Y' :=?; \\ \quad \textbf{choice} \\ \qquad [\mathsf{Q}_{P,L_1}(E,X,X',Y')]; \\ \qquad X,Y := X',Y'; Trs_1 \\ \qquad \vdots \\ \qquad [\mathsf{Q}_{P,L_n}(E,X,X',Y')]; \\ \qquad X,Y := X',Y'; Trs_n \\ \quad \textbf{endchoice} \quad \rangle^{\text{cc}} \end{array}$$

$$P(E,X,Y);Trs \xrightarrow{\text{cc}} \begin{array}{l} \langle \quad \{\mathsf{P}_P(E,X)\}; X',Y' :=?; \\ \quad [\mathsf{Q}_{P,0}(E,X,X',Y')]; \\ \quad X,Y := X',Y'; Trs \quad \rangle^{\text{cc}} \end{array}$$

where the labels $L_1,\ldots,L_n$ are disjoint and each label L_i is a postcondition of P. X',Y' are pairwise disjoint lists of distinct variables such that for each X'_i and Y'_j, $\mathsf{T}_{X'_i} = \mathsf{T}_{V_i}$ and $\mathsf{T}_{Y'_i} = \mathsf{T}_{R_i}$. The identifiers X',Y' are picked so as to be fresh for the transitions trees $Trs_1 \ldots Trs_n$.

The assertion obliges the caller to ensure that the precondition, with actuals substituted for the corresponding formal value(-result) parameters, holds. X' and Y' are introduced for the return values of the procedure. For each branch, the matching postcondition, over E, X, X' and Y', is added as an assumption, and then the return values are assigned back to the actual (value-)result parameters. We note that if the same variable is passed for multiple formal value-result or result parameters, it will hold the return value assigned to the last such formal parameter in the procedure's signature due to our semantics of the multiple assignment statement.

If the labels $L_1 \ldots L_n$ do not cover all postconditions of P, Socos warns that the procedure may not be live.

6.6.3 Recursive procedures

Recursion occurs when a procedure calls itself in the body, either directly (single recursion) or by calling another procedure that eventually calls it back (mutual recursion). Similarly to how loops are handled, we analyze the procedure call graph and detect the strongly connected components to identify the procedures that are involved in a recursion.

Socos currently use the following simple method for verifying termination of recursive procedure calls. All procedures in a component for which termination is to be verified are required to share both a common signature and a common variant. This constraint allows the procedures to be considered to operate in a common statespace, simplifying the verification methodology. If termination verification for a component is not intended—i.e., no variant is given for any

of the participating procedures—the constraint on the signatures does not apply. If termination verification is intended, with W being the shared variant, C the formal constant parameters, and V_0 the initial-value constants for the value-result parameters, the assert statement

$$\{\, 0 \leq \mathsf{W}(E,X) \wedge \mathsf{W}(E,X) < \mathsf{W}(C,V\_0) \,\}$$

is prepended just before the precondition assertion in the VC for the non-recursive case above. Thus, at the point of each call, the caller is additionally obliged to ensure that the variant in terms of the actual parameters is bounded from below and has decreased since the entry to the procedure. This assertion ensures that the shared variant decreases with each call and that the recursion eventually terminates.

We note that this method in its present form works well for single recursion, but imposes significant restrictions on mutually recursive procedures. In addition to requiring all participating procedures to share a common signature and invariant, it also requires the variant to be decreased by *every* call (cf. the termination rules for loops). These constraints are significant and have implications on the kind of programs that can be expressed in Socos at present if termination is to be verified. In particular, the rules enforce a very tight coupling between the mutually recursive procedures. Better approaches exist in the literature; we discuss some of these in the next section.

6.7 Summary and discussion

Weakest preconditions give a semantic basis for a syntax-directed translation of programs into logic VCs. We have in this chapter given a verification schema for Socos programs based on weakest preconditions. A context is verified by checking that each procedure is consistent, live and terminating. Consistency means that every transition establishes its target, and that no assertion fails. Liveness means that a program does not get stuck in an intermediate situation and that each transition can proceed to its target. This is ensured by checking that the transition graph from the initial situation does not contain dead ends, that no assume statements are used, and that all exits are handled in procedure calls. Termination means that there are no infinite loops. Termination condition generation is enabled by annotating the situations with variants and decreasing declarations such that every cycle is cut by a transition that decreases a variant. For recursive procedures, each recursive call must decrease a variant that is shared among all procedures involved in the recursion.

Consistency is a property that should always be verified: every transition added to the diagram must be consistent. A program can be verified to be consistent even if it is incomplete (i.e., not live and/or not terminating). Incomplete programs may serve as stepping stones towards a final, complete program.

6.7.1 Related work

Many verification tools use weakest precondition semantics as a basis for transforming programs into verification conditions suitable for checking in an automatic theorem prover. For instance, in the Boogie methodology [32] a BoogiePL [60] program containing specifications, code and background theory is first converted using weakest preconditions into VCs, which are then sent to, e.g., the Z3 SMT solver [59]. BoogiePL is, however, an intermediate language that is not intended to be used for programming as such but rather for encoding the verification semantics of industrial languages (as has been done for, e.g., C# in the form of Spec# [33]).

Statements with multiple exits have traditionally been studied in the context of exceptions [10, 107]. We have not pursued this research direction here. In an unpublished report, Back and Karttunen [24] extend the weakest precondition to a function of multiple postconditions and give a semantics for goto-programs. While a weakest precondition calculus for multi-exit statements would allow us to state and analyze the correctness conditions for Socos programs more precisely, development of such a calculus was considered to be outside the scope of this thesis.

Back and Preoteasa [29] have given a formal operational semantics for invariant diagrams and defined with respect to this semantics a set of sound and complete Hoare-like proof rules. The rules allow decomposition of conditions on diagrams into conditions on transitions. Their work provides a formal proof theory for invariant based programming. However, we have not yet done a careful analysis of how the Socos verification methodology maps to their semantics.

Podelski and Rybalchenko [139] have given in the context of an automata-theoretic framework a proof rule for decomposing a termination verification problem into smaller independent verifications based on the notion of disjunctive well-foundedness. Their approach is based on identifying a transition invariant—a binary relation over program states that includes the reachability relation—that it is disjunctively well-founded, meaning that it is a finite union of well-founded relations (disjunctive well-foundedness is a weaker property than well-foundedness). In their case proving termination amounts to establishing well-foundedness of the subrelations of the transition invariant.

Homeier [99, Ch. 7] describes a method for verifying termination of mutually recursive procedures. There every recursion cycle from a procedure back to itself is shown to make progress (decrease a variant); if this recursion contains intermediate cycles through other procedures (called *diverted* recursion), additional VCs are generated to ensure that the intermediate cycle does not interfere with the progress. In contrast to the Socos method, progress is aggregated along the paths in the call graph and a VC is generated for every cycle, making the method more general at the cost of locality. We note that in his master's thesis [138], Plüss sketched an IBP adaptation of Homeier's method for mutually recursive procedures with multiple exits, but this has not been implemented.

6.7.2 Final remarks

Our liveness semantics corresponds to the standard way a program is executed, in the sense that from each intermediate state during execution of a transition there must be at least one next state to proceed to. In other words, once a transition has been triggered, it must run to completion. As an alternative, one can consider a semantics in which a command that cannot proceed causes the program to backtrack to a previous state to try an alternative branch. If this backtracking is repeated until transitions are exhausted, liveness of a transition tree can be defined as the property that one of the leaves is eventually reached. Under such an interpretation of liveness the transition tree

if
  $[A]$; $[B]$; **goto** X
  $[C]$; **goto** Y
endif

would be live from all states satisfying $(A \wedge B) \vee C$. This interpretation allows for a larger class of live programs, although the operational semantics can be considered rather esoteric. It is not supported by the tool.

We acknowledge that the requirements on mutually recursive procedures with regards to termination proofs are presently too strong. In general, it seems difficult to give a simple and at the same time sufficiently general verification method for programs with mutually recursive procedures. One possibility is to extend the verification method for termination of iterative programs to mutually recursive procedure calls; e.g., each call statement could be annotated with decreasing/nonincreasing assertions and VCs generated accordingly. However, this does not address the restriction that recursive procedures must have a shared statespace.

A present limitation is that the range of the variant function must be the non-negative integers. For increased generality, it should be possible to use any well-founded set, but support for this has not yet been implemented.

Chapter 7

Verifying Socos Programs in PVS

This chapter describes the translation of invariant diagrams into PVS theories. We also describe how the generated verification conditions are handled.

7.1 Introduction

The first step of automation of most formal methods consists of applying some form of general, highly automated proof strategy to simplify the VCs. The result of the simplification is a set of undischarged VCs, which are reported to the programmer so that errors and omissions can be identified, and/or a more specific proof strategy attempted. In an IBP tool this verification machinery should satisfy the following desiderata:

- The interface should be lightweight and easy to interact with. It should be possible to send an invariant-based program under development for checking with a simple command, and the checker should report the results immediately.
- The system should be general purpose; it should not be limited to any specific program domain.
- Conditions that were automatically discharged should not be displayed at all, maintaining the impression that the program *is* the proof.
- Conditions that were not automatically discharged should be shown to the programmer without syntactic clutter introduced in a translation process; e.g., they should contain the same identifiers, operators and term structure.
- Conditions should be presented in a sufficiently succinct way to assist the user in correcting the program.
- When automation fails, it should be possible to interactively prove individual conditions.

The above have fundamental implications on how VCs should be processed. Firstly, the process should be completely automatic and the conditions generated in a format that is amenable to automatic verification. Secondly, the translation function from programs to VCs should maintain the connection between the program and the generated conditions. Thirdly, the conditions must be processed at a fine level of granularity. When verifying that a transition establishes its target situation, the tool should filter out all constraints that were discharged automatically, and highlight those that were not.

In general, it is possible to achieve higher degree of automation by using assumptions about the domain. In PVS it is standard practice to develop a library of strategies together with a theory library to achieve a increased automation for proofs in the domain of the library. An IBP tool should also provide means for building, extending and applying *background theories* in relevant programming domains. Background theories can introduce definitions, automatic rewrites, lemmas and strategies that are useful both for formulating the program and its invariants as well as for improving automation of the verification.

Another desirable property of the translation process is that the generated theories are human readable. While the intention is that the end user works at the level of diagrams, not PVS theories, the generation of readable VCs has some advantages. Firstly, looking at the generated conditions can be helpful in understanding the program under development. Secondly, since the VC generator is under development (and has not been verified correct itself) there is value in that the generated conditions can be scrutinized.

The remainder of the chapter is organized as follows. Section 7.2 describes the translation of contexts, procedures and situations into PVS theories. Section 7.3 describes how the generated conditions are handled. In Section 7.4 we consider the type correctness conditions of the generated theories. Section 7.5 discusses background theories and gives a simple example. The chapter ends with summary, conclusions and discussion of related work.

7.2 Translation into PVS theories

During verification of one context, SOCOS generates a hierarchy of unparametrized PVS theories as follows:

- A single `ctx` theory containing the importing and constant declarations is generated for the context being verified. This theory imports the `ctx` theory for each extended context.
- A `spec` theory, importing the main `ctx` theory, is generated for each procedure. The `spec` theory contains the visible part of a procedure specification—its signature, variant and pre- and postconditions.
- An `impl` theory is also generated for each procedure. This imports the `spec`

theory of the procedure as well as the `spec` theories of each called procedure. It contains PVS declarations for the state vector, the situations, and the VCs for all transition trees.

Only `impl` theories contain VCs. We note that an `impl` theory depends on the specification of the procedure, the specifications of each called procedure, and each `ctx` theory in the image of the contexts of these specifications under the reflexive transitive closure of the extends relation. An example translation is shown in Figure 7.1. The left hand side shows a program consisting of two contexts, of which one contains two mutually recursive procedures. The right hand side shows the generated PVS theories and the importing relation.

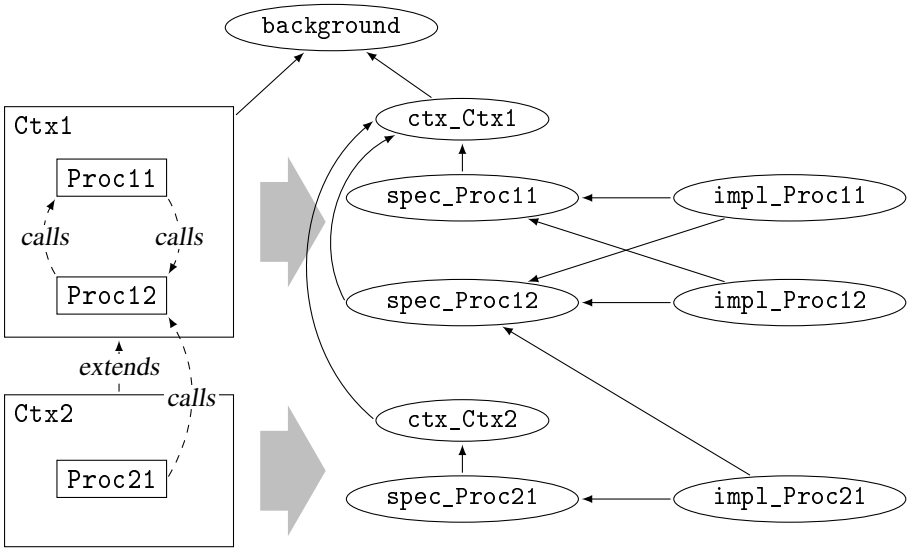


Figure 7.1: Example PVS translation of two contexts. Contexts and procedures are drawn as boxes, theories as ellipses, calls and extends relations as dotted arrows, and importing relations as solid arrows

The `spec` and `impl` theories contain straightforward translations of the pre-conditions, postconditions and situations into PVS predicates. We give a brief summary here; the syntactic translation rules are listed in the Appendix, Section A.2. For a procedure P we have the translated predicates:

$$\begin{array}{lcllcl} \mathsf{P}_P & = & \texttt{spec\_}P\texttt{.pre\_} & \mathsf{J}_X & = & \texttt{sit\_}X \\ \mathsf{Q}_{P,0} & = & \texttt{spec\_}P\texttt{.post\_} & \mathsf{I} & = & \texttt{sit\_ini\_} \\ \mathsf{Q}_{P,X} & = & \texttt{spec\_}P\texttt{.post\_}X & \mathsf{F} & = & \texttt{sit\_fin\_} \\ & & & \mathsf{F}_X & = & \texttt{sit\_fin\_}X \end{array}$$

Generated identifiers include the underscore character to avoid name clashes with user defined identifiers. The specification predicates are prefixed with the `spec` theory identifier for disambiguation since multiple `spec` theories may be imported.

The components of the state vector, i.e., the mutable parameters and local variables, are declared as logical variables in the `impl` theory. Constant parameters and initial-value constants are declared as constants in the theory. Since the PVS **importing** statement does not import variable names, there is no clash with names from the `spec` theories. The predicates on the right hand side above are all over the state vector. The rest of the `impl` theory contains VCs, one for each situation that has a transition tree, and additionally one for the initial transition tree (if present). These are described in the next section.

7.3 Verification conditions

Each situation that has a transition tree generates a VC consisting of two components: a lemma declaration with Formula 6.2 as the proof obligation, and a ProofLite proof script associated with the lemma. Given a situation X, the state vector σ, and a transition tree Trs rooted at X, the following transformation is applied to the triplet $X, \sigma \bullet Trs$ to generate the two components:

$$
\begin{aligned}
X, \sigma \bullet Trs \xrightarrow{\text{vc}} \; & \texttt{vc\_}X : \textbf{lemma}\ \forall \sigma : \texttt{sit\_}X(\sigma) \implies \langle Trs^{\text{cc}}(\sigma)\rangle^{\text{pvs}} \\
& \texttt{\%|-vc\_}X\texttt{ : proof} \\
& \texttt{\%|- (skolem-2)} \\
& \texttt{\%|- (flatten-disjunct + 1)} \\
& \texttt{\%|- } \langle Trs^{\text{cc}}(\sigma)\rangle^{\text{prf}} \\
& \texttt{\%|-qed}
\end{aligned}
$$

The transformations pvs and prf are defined over the structure of consistency conditions. The former expands the wp function and beta-reduces until all applications of a predicate to σ are of the form $\texttt{sit\_}X(\sigma)$. The latter builds an S-expression to be inserted into the proof script associated with the VC for the transition tree. The proof script first eliminates the top level quantifier. The command `skolem-2` is a simple adaptation of the PVS `skolem` command: instead of generating new names for the Skolem constants, it reuses the names from the binding expressions; this makes the VC slightly easier to read.[1] The proof script then flattens the sequent, such that the antecedent of the implication becomes the antecedent of the sequent. At this point in the proof the sequent is of the form $\alpha \vdash \beta$, where α is the precondition and β is the consistency condition of the transition tree.

We recall from Chapter 6 that the consistency condition for a transition tree is a term of the form $\text{wp}(S)(Q_1 \cap \cdots \cap Q_n)$ where each Q_i is either a term of the same form or a situation predicate. The transformations pvs and prf defined over this structure are given in Table 7.1.

[1] PVS does not allow a constant that is in the current scope to be used as a skolem constant. Hence, skolemization using the variable names may fail if, e.g., a constant with a conflicting name is being imported via a background theory. For such names `skolem-2` falls back to the automatic name generation of `skolem!`.

1.	$\texttt{sit\_}X(\sigma)$	$\xrightarrow{\text{pvs}}$	$\texttt{sit\_}X(\sigma)$
		$\xrightarrow{\text{prf}}$	`(expand-defs` *defs-sexp*`)` `(check-report` *strat-sexp*`)`
2.	$(Q_1 \cap \cdots \cap Q_n)(\sigma)$ $(n>1)$	$\xrightarrow{\text{pvs}}$	$(\langle Q_1(\sigma)\rangle^{\text{pvs}}) \wedge \cdots \wedge (\langle Q_n(\sigma)\rangle^{\text{pvs}})$
		$\xrightarrow{\text{prf}}$	`(branch (split-n` n`)` `(then` $\langle Q_1(\sigma)\rangle^{\text{prf}}$`)`$\ldots$`(then` $\langle Q_n(\sigma)\rangle^{\text{prf}}$`))`
3.	$\text{wp}(S_1;S_2)(Q)(\sigma)$	$\xrightarrow{\text{pvs}}$	$\langle \text{wp}(S_1)(\text{wp}(S_2)(Q))(\sigma)\rangle^{\text{pvs}}$
		$\xrightarrow{\text{prf}}$	$\langle \text{wp}(S_1)(\text{wp}(S_2)(Q))(\sigma)\rangle^{\text{prf}}$
4.	$\text{wp}([E])(Q)(\sigma)$	$\xrightarrow{\text{pvs}}$	$(E) \implies (\langle Q(\sigma)\rangle^{\text{pvs}})$
		$\xrightarrow{\text{prf}}$	`(flatten-disjunct + 1)` $\langle Q(\sigma)\rangle^{\text{prf}}$
5.	$\text{wp}(\{E\})(Q)(\sigma)$	$\xrightarrow{\text{pvs}}$	$(E) \wedge (\langle Q(\sigma)\rangle^{\text{pvs}})$
		$\xrightarrow{\text{prf}}$	`(branch (split-assert)` `((then (expand-defs` *defs-sexp*`)` `(check-report` *strat-sexp*`))` `(then` $\langle Q(\sigma)\rangle^{\text{prf}}$`)))`
6.	$\text{wp}(X := E)(Q)(\sigma)$	$\xrightarrow{\text{pvs}}$	$(\lambda\,(X_1 : \mathsf{T}_{X_1}, \ldots, X_n : \mathsf{T}_{X_n}) : \langle Q(\sigma)\rangle^{\text{pvs}})(E)$
		$\xrightarrow{\text{prf}}$	`(beta +)` $\langle Q(\sigma)\rangle^{\text{prf}}$
7.	$\text{wp}(X :=?)(Q)(\sigma)$	$\xrightarrow{\text{pvs}}$	$(\forall\,(X_1 : \mathsf{T}_{X_1}, \ldots, X_n : \mathsf{T}_{X_n}) : \langle Q(\sigma)\rangle^{\text{pvs}})$
		$\xrightarrow{\text{prf}}$	`(skolem-2)` $\langle Q(\sigma)\rangle^{\text{prf}}$

Table 7.1: Rules for transformations pvs and prf. The pattern on the left hand side is common for each rule pair

The pvs rules convert the predicates into PVS terms and apply the standard weakest precondition transformation on the statements defined in Section 5.3.5. The proof script generated by the prf rules builds a proof tree with a single consequent at every node, and which branches in synchrony with the correctness condition. The prf transformation depends on two S-expression parameters: *strat-sexp*, the default strategy to be applied to the leaves of the proof tree, which is either defined in the current context or inherited from other contexts; and *defs-sexp*, listing the definitions relevant to the sequent (specifically pre- and postconditions, situations, and procedure variants). In addition to `skolem-prefer` the prf rules in Table 7.1 use four Socos specific strategies: `expand-defs`, `check-report`, `split-n`, and `split-assert`. They are described below. These strategies are based on the PVS strategies combined with queries into the proof state; they do not define new proof rules nor manipulate the proof state, and hence constitute a conservative extension of PVS.

Rules (1) apply to the leaves of the correctness condition, at which the consequent is a single situation or assertion. The command `expand-defs` expands all situation, pre-/postcondition and variant definitions in the sequent, and additionally if the consequent is a sequence of constraints splits the constraints into separate proof branches. Each proof goal produced by `expand-defs` is of the form $\alpha_1, \ldots, \alpha_n \vdash \beta$, where β is a target situation constraint or an assertion, and each formula α_i originates from a source situation constraint, an assume statement, or an assertion. `check-report` applies *strat-sexp* to each goal, and then prints the status of the proof.

Rules (2) reduce an intersection of predicates into a conjunction of formulas, and split the proof tree such that each conjunct becomes a separate branch. `split-n` is an iterated version of `split`: it splits the top-level conjunction into two branches, and then repeatedly splits the right branch for a total of at most $n-1$ iterations. Hence, a sequent of the form $\Gamma \vdash \alpha_1 \wedge (\alpha_2 \wedge (\alpha_3 \wedge \ldots))$ is split into the goals $\Gamma \vdash \alpha_1, \Gamma \vdash \alpha_2, \Gamma \vdash \alpha_3 \ldots,$.

The remaining rules give the weakest preconditions of single exit statements.

Rules (3) convert a composition of statements into a nested formula and a sequence of proof commands.

Rules (4) convert an assume statement into an implication and applies `flatten-disjunct`, promoting the antecedent of the implication into an antecedent sequent formula. The first parameter to `flatten-disjunct` indicates that only consequent formulas should be flattened; the second parameters limits the depth to one level.

Rules (5) branch off an assertion of E from the main proof tree. The command `split-assert` splits a sequent of the form $\Gamma \vdash \alpha \wedge \beta$ into the two goals $\Gamma \vdash \alpha$ and $\Gamma, \alpha \vdash \beta$. The first goal becomes a verification target, to which the commands `expand-defs` and `check-report` are applied. Remaining conditions in the second branch get α as an additional antecedent.

Rules (6) apply substitution. The substitution is applied through a lambda binding expression, and the proof script applies *beta-reduction* to the consequent. A side effect of `beta` is that it rewrites all redexes in the consequent to their reduced form, including those that occur inside expressions and invariants. This means that redexes in user defined expressions will be shown in the reduced forms in the unproven sequents reported by Socos. This is not ideal, since it may cause conditions being shown in a different form than was given by the user, contradicting the requirement that they should be unaffected by the translation process. Unfortunately, PVS does not provide a command for restricting beta reduction to specific redexes. However, in practice this case has not been a problem.

By executing the proof script produced by the above rules in batch mode, a transcript of the proof run is produced. The commands `expand-defs` and `check-report` instrument the transcript with text strings indicating the status of the proof after *strat-sexp* has been applied to a proof goal. This information is used by Socos to aggregate the result of the verification.

7.4 Type correctness

PVS generates type correctness conditions (TCCs) to ensure that a theory is well typed. A PVS theory is not consistent until all its associated TCCs have been discharged. Since type checking in PVS is undecidable in the general, discharging the TCCs can require theorem proving. By default, PVS automatically applies the strategy `tcc`, which is based on `grind`. TCCs that are not discharged by this strategy can be proved interactively. The assumptions available in a TCC depend on the context in which it is generated; e.g., given the formula $P \Rightarrow Q$, type conditions on the subformula Q may be proved under the antecedent P. The semantics of PVS type correctness is described in detail in [132]. We note here that the type conditions associated with the PVS files generated by Socos subsume the following type conditions on Socos programs:

- All expressions that occur in a program are consistently typed.
- Precondition constraints are boolean expressions over the constant and value parameters; postcondition constraints are boolean expressions over parameters and initial-value constants.
- Situation constraints and assume/assert arguments are boolean expressions over the state vector. Every constraint in a situation or condition is type-checked in the context of the preceding constraints; for nested situations, the predicate of the immediate outer situation is included in the type checking context.
- Assignments are type consistent; the expression assigned to a variable has a type that is from the type of the variable.
- An actual constant parameter has a type from the formal parameter's type. An actual value-result parameter has a type that is equivalent with the type of the formal parameter. An actual result parameter has a type that includes the type of the formal parameter.

We further note that for an expression that occurs inside a transition tree, the source situation predicate as well as the preceding assertions and assume statements are included in the type checking context. This allows, for instance, a guarded transition (tree) to be type checked under the assumption of the guard. Since the PVS embedding requires every assignment to be type correct, we use predicate subtyping to encode "strong" invariants over the program variables—i.e., constraints on the values that must hold after each assignment statement.

The types of program variables should be non-empty. In general, it is best to use types with non-emptiness judgments to avoid generation of extraneous TCCs. Procedures with parameters over possibly empty types can be specified, but the implementation of such procedures will generate unprovable TCCs if the type

of an in-parameter (constant or value-result) may be empty. If the type of any one variable in the state vector is empty, the program does not have a meaningful interpretation, as its statespace is empty.

7.5 Background theories

Background theories are PVS theories imported into verification contexts. They serve two main purposes:

- to introduce domain-specific definitions, notations and types that simplify the specification and implementation of a program; and
- to support domain-specific strategies for discharging VCs.

Thus, background theories facilitate not only the coding of the program, but also the verification of the program. As an example we have defined a small background theory for variable size arrays, `vector`, and a strategy, `endgame`. The theory is listed in the Appendix, Section A.3. It includes the record type `vector` from Chapter 4, and adds an array read operator, denoted $\mathtt{a}[\mathtt{i}]$ (ASCII: $\mathtt{access}(\mathtt{a},\mathtt{i})$), as well as an update operator, denoted $\mathtt{a}[\mathtt{i} \leftarrow \mathtt{x}]$ (ASCII: $\mathtt{update}(\mathtt{a},\mathtt{i},\mathtt{x})$). The second argument `i` of both operators is type restricted to be a legal index, and the type of $\mathtt{update}(\mathtt{a},\mathtt{i},\mathtt{x})$ is the arrays with the same length as `a`. The operators satisfy the following basic read-over-write properties of arrays:

1. $\mathtt{a}[\mathtt{i} \leftarrow \mathtt{x}][\mathtt{i}] = \mathtt{x}$
2. $\mathtt{i} = \mathtt{j} \vee \mathtt{a}[\mathtt{i} \leftarrow \mathtt{x}][\mathtt{j}] = \mathtt{a}[\mathtt{j}]$

The `endgame` strategy is implemented with the following Lisp code:

```
(defstep endgame (&optional (lemmas nil))
  (let ((introduce-lemmas
         `(then ,@(loop for l in lemmas
                        append `((lemma ,l))))))
    (then
      (skosimp*)
      (auto-rewrite-defs :always? t)
      (assert)
      introduce-lemmas
      (yices)
      (fail)))
  "End-game strategy"
  "Invoking yices, supplying lemmas: ~{~a~^, ~}")
```

The strategy applies skolemization and simplification, followed by `auto-rewrite-defs` and `assert` to expand all relevant definitions in the

sequent, loads the supplied formulas into the antecedent with `lemma`, and finally invokes `yices`. Definitions for which auto-rewriting is disabled are not expanded in the second step; they become uninterpreted constants in the logical context of Yices. The optional lemmas supplied as parameters to `endgame` appear as axioms to Yices. Supplying the above read-over-write properties allows `endgame` to prove basic conditions involving `vectors`. For example, Figure 7.2 shows a procedure which reverses the order of an array. In this case, the command `(endgame :lemmas (update_prop_1 update_prop2))` discharges all VCs associated with the procedure automatically.

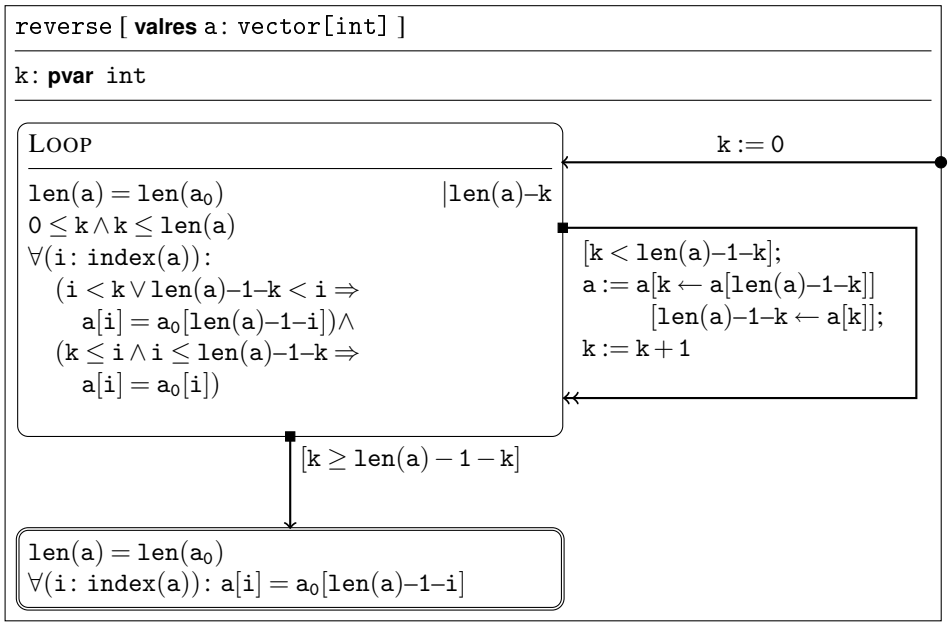


Figure 7.2: A program to reverse an array

Additional operators for manipulating arrays, concatenation, slicing, and so on can then be added as they are needed; theorems about these operators can be introduced to achieve proof automation for programs using them. In the next chapter, we will extend the background theory with a set of definitions useful for reasoning about sorting of integer `vectors`.

Any number of lemmas can be given to `endgame` as parameters; they are introduced into the proof as consequents. However, this mechanism must be used judiciously, as adding a lemma increases the size of the decision problem to which Yices is applied for each VC in the verified context. A possible consequence is that a verification which was previously quickly decided, now runs for an unacceptable amount of time.[2] Hence, parameters are best suited for basic properties required

[2]By default, Socos runs Yices for a maximum of 10 seconds per condition. This can be tuned with the parameter `-tm` in the PVS global variable `*yices-call*`.

to discharge most conditions; more specific properties required in only a few transitions are typically better handled with rewrite rules that translate the condition into a simplified form before applying Yices. Preprocessing steps are easily added by redefining the default strategy to

```
(then S (endgame parameters))
```

where *S* is the pre-processing step.

7.6 Summary and discussion

We have described how a program consisting of a set of mutually recursive procedures, where each procedure is implemented as an invariant diagram, is translated into a PVS theory hierarchy. Each procedure generates a specification and an implementation theory, the former containing the pre- and postconditions of the procedure, and the latter containing the situations and VCs. The implementation theory for a procedure imports the specification theories of the procedure itself as well as all called procedures. Additionally, the contexts and background theories are imported. Socos sends the generated theories to PVS for typechecking and verification. A program is correct when all VCs and TCCs in the theory hierarchy have been discharged.

VCs are generated from the transition diagram in a syntax directed process. Each transition tree generates one PVS lemma. The translation is based on the consistency conditions for transition trees and on the standard weakest precondition semantics of statements. For each condition, a PVS proof script to reduce the condition into fine grained proof goals is generated. Each constraint in a situation or specification becomes a separate proof goal. A default strategy is applied to each goal. Proof goals that are not directly discharged are reported to the programmer, and may be helpful when correcting the program. Conditions that were not discharged automatically can be proved interactively in PVS.

The verification method described here is very open ended. Background theories and user defined strategies give access to the full verification arsenal of PVS. Background theories can be used to define domain-specific notations and types, and to improve automation of the verification. The task of the default strategy is to discharge as many of the valid leaf sequents of the proof tree as possible. The `endgame` strategy invokes Yices to discharge the VCs, and provides a basic mechanism for feeding assumptions into the decision procedure. Yices was picked because it is already integrated into PVS. Our experience is that it is well suited as a default catch-all endgame prover for Socos. It handles linear constraint and basic quantifier reasoning, and in practice discharges the simpler conditions such as loop conditions and conditions on array bounds reasonably quickly. While Yices is limited to first-order problems, it can still be useful for discharging conditions involving higher-order properties described by uninterpreted functions.

7.6.1 Related work

A number of existing verification tools use PVS as a back-end. Owre [130] gives a recent survey of some of the major tools.

Muñoz [120] has implemented an embedding of the B Abstract Machine Notation into PVS. The tool is a compiler (called PBF) that takes as input an abstract machine and generates an embedding of it into a PVS theory. In this approach, soundness of a B machine is implied by the type correctness of the generated theory.

Another example is the Loop compiler for JML-annotated Java [149]. It is based on a coalgebraic semantics of Java, and includes a VC generator that converts a Java program into a PVS theory hierarchy.

The Why VC generator [76], part of the Why/Krakatoa/Caduceus framework [77], can use a number of back-end proof tools, including PVS and Yices.

7.6.2 Future work

The open-endedness of the system vs. its robustness is an issue that remains to be considered. With the current embedding it is possible to alter the interpretation of conjunction, equality and other boolean relations in the generated predicates by redefining the corresponding operators (e.g., within a background theory). The notion of correctness under such alterations is not well defined. Due to the prototype nature of the tool, this issue has not been a priority. Hence background theories must at the present be conservative in the sense that they should not redefine types and constants used in the theory generation. This includes the `bool` and `int` types and the operators $=$, $\wedge$, $\vee$, $\Rightarrow$ and $\leq$. Straightforward solutions are to fully name qualify the operators in generated theories, or to reserve a set of aliased operators for Socos. On a more general level, the boundaries between user definable parts and the background framework should be more clearly defined.

One possibility that has not yet been explored is that of extracting useful information about the proof state when a catch-all strategy fails to discharge a goal. In particular *counterexamples* could provide useful feedback to the programmer when verification fails.

Chapter 8

An Exercise in Tool-Supported IBP

This chapter exemplifies the Socos verification methodology. We build a verified implementation of the heapsort algorithm, while using a background theory to achieve a high degree of proof automation.

8.1 Introduction

Heapsort [155] is an in-place, comparison-based sorting algorithm from the class of selection sorts. It achieves $O(n \log n)$ worst and average case performance by storing the unsorted elements in a binary max-heap structure, allowing for constant time retrieval of the maximal element and logarithmic time restoration of the heap property after the maximal element has been removed. By storing the heap in the unsorted portion of the array, heapsort uses only a small constant amount of additional memory.

The algorithm implemented in this chapter is very close to the one given by Cormen et al. in [55, Ch. 6]. It comprises two main loops in sequence. The first loop builds a max-heap out of an unordered array by extending a partial heap one element at a time, starting from the end of the array. The second loop deconstructs the max-heap one element at a time, maintaining a sorted subarray after the heap. Each iteration extends the sorted portion by swapping the root of the heap with the last element of the heap, and then restores the heap property for the next iteration.

Heapsort is an interesting exercise in IBP for a number of reasons. Firstly, even though an implementation of heapsort can be given rather compactly, its verification involves a set of nontrivial invariants and proofs. Secondly, it is not so easy to write a correct implementation of heapsort up front. In particular, there is a corner case that is easily missed. Thirdly, as with other sorting algorithms the specification of heapsort involves the notion of permutation. While it is straightforward to give a general definition of permutation, reasoning in the actual verification in terms of the

definition is infeasible. In Socos, we can use background theories to supply more useful abstract properties for the automatic tactic, while maintaining soundness with regard to the definitions.

8.2 Specification

We focus in our example on sorting arrays of type `vector[int]` (the `vector` theory is listed in the Appendix, page 146). The postcondition should guarantee that the array 1) is arranged in non-decreasing order, and 2) preserves the values from the original array. We introduce a predicate `sorted` to express property (1) in a new PVS theory called `sorting`:

```
sorting : theory
begin
  importing vector[int]
  a, b, c : var vector
  sorted(a) : bool = ∀(i, j : index(a)) : i < j ⇒ a[i] ≤ a[j]
  ⋮
```

In the sequel we use `sorting` as a background theory for our sorting program, extending it with additional definitions as needed.

Formalizing property (2), we define a binary predicate `perm` over vectors as follows, asserting the existence of a bijective mapping over the indexes:

```
perm(a, b) : bool = ∃(f : (bijective?(index(a), index(b)))) :
                      ∀(i : index(b)) : b[i] = a[f(i)]
```

While this definition of `perm` is mathematically concise, reasoning in terms of the definition gives little room for automation since it requires demonstration of a bijection. Higher order quantifiers render Yices incomplete, and we will also have little luck with the catch-all strategies of PVS (such as `grind`) proving even a basic property such as reflexivity. Instead, when verifying sorting algorithms which work on pairs of elements it is more fruitful to consider permutation as the smallest equivalence relation that is invariant under the pairwise exchange of elements. Proceeding in this direction, we introduce and prove the following properties of `perm` in PVS:[1]

```
perm_len : lemma perm(a, b) ⇒ len(a) = len(b)
perm_ref : lemma perm(a, a)
perm_sym : lemma perm(a, b) ⇒ perm(b, a)
perm_trs : lemma perm(a, b) ∧ perm(b, c) ⇒ perm(a, c)
```

[1] As `perm` may be useful also in contexts outside of sorting of integer arrays it may be better to build a separate, generic background theory for permutations. For brevity we include everything in a single theory here.

The lemma perm_len allows the prover to infer that permutations have equal length. This ensures that a valid index in an array is also a valid index in any permutation of the array. The lemmas perm_ref, perm_sym and perm_trs state that permutation is an equivalence relation. Proving these lemmas is a straightforward exercise in PVS, involving in each case finding the right instantiation of the bijection f and applying basic properties of bijections from the PVS prelude.

We introduce a function swap for exchanging the elements at indexes i and j, while keeping the remainder of the elements in the array unchanged:

$$\texttt{swap}(\texttt{a}, (\texttt{i}, \texttt{j} : \texttt{index}(\texttt{a}))) : \{\texttt{b} | \texttt{len}(\texttt{b}) = \texttt{len}(\texttt{a})\} = \\ \texttt{a}[\texttt{i} \leftarrow \texttt{a}[\texttt{j}]][\texttt{j} \leftarrow \texttt{a}[\texttt{i}]]$$

The property that swap maintains the length is encoded in a predicate subtype. All array manipulations in the heapsort program will be pairwise swaps, so the endgame strategy only needs to know the following about swap: its effect on subsequent array reads, and that swap maintains permutation. We state these properties as follows:

$$\texttt{swap\_acc} : \textbf{lemma} \\ \forall(\texttt{a}, (\texttt{i}, \texttt{j}, \texttt{k} : \texttt{index}(\texttt{a}))) : \texttt{swap}(\texttt{a}, \texttt{i}, \texttt{j})[\texttt{k}] = \texttt{a}[\ \textbf{if}\ \texttt{k} = \texttt{i}\ \textbf{then}\ \texttt{j} \\ \textbf{elsif}\ \texttt{k} = \texttt{j}\ \textbf{then}\ \texttt{i} \\ \textbf{else}\ \texttt{k}\ \textbf{endif}\ ]$$

$$\texttt{swap\_perm} : \textbf{lemma} \\ \forall(\texttt{a}, (\texttt{i}, \texttt{j} : \texttt{index}(\texttt{a}))) : \texttt{perm}(\texttt{a}, \texttt{swap}(\texttt{a}, \texttt{i}, \texttt{j}))$$

The proofs of these lemmas are trivial. The first property follows directly from the definitions of update and access; it is proved with (grind). The second lemma is proved by instantiating the existential quantifier and demonstrating bijectivity.

Finally, to support automatic reasoning in terms of the above more abstract properties of perm and swap rather than the definitions, we add the following declaration to the background theory:

auto_rewrite- perm, swap

This prevents occurrences perm and swap from being expanded into their definitions, and hence they will be treated as uninterpreted functions by Yices when the endgame strategy is invoked. Next, we introduce a new context heapsort that imports our background theory and applies endgame with the six properties we have defined so far:

```
heapsort : context
begin
    importing sorting;
    using "(endgame :lemmas (perm_len perm_ref perm_sym
                             perm_trs swap_acc swap_perm))";
    ⋮
```

The remainder of the context `heapsort` will be procedures, which we present as invariant diagrams. The complete `heapsort` context is listed in textual format in the Appendix, page 148.

8.3 Situation structure

We introduce a procedure `heapsort`, which given a value-result parameter `a` of type `vector[int]`, should achieve the postcondition $\texttt{sorted}(\texttt{a}) \wedge \texttt{perm}(\texttt{a}, \texttt{a}_0)$. We design `heapsort` around two situations called BUILDHEAP and TEARHEAP. The former builds the heap out of the unordered `a` by moving in each iteration one element of the non-heap portion of `a` into its correct place in the heap portion; the latter then sorts `a` by selecting in each iteration the first (root) element from the heap portion and prepending it to the sorted portion of the array. TEARHEAP is not entered until BUILDHEAP has completed, so we use the loop variable `k` in both loops. In both situations `k` will be in the range $[0..\texttt{len}(\texttt{a})]$, and the invariant $\texttt{perm}(\texttt{a}, \texttt{a}_0)$ should also be maintained by both situations.

In the first stage, BUILDHEAP, the heap is extended leftwards one element at a time by decreasing `k`. The portion to the right of `k` satisfies the following *max-heap property*: an element at index i is greater than or equal to both the element at index $2i+1$ (the "left child") and the element at index $2i+2$ (the "right child"). Figure 8.1 shows the invariant of BUILDHEAP and the loop transition. The loop terminates when `k` reaches zero. For each iteration, after `k` has been decremented the new element at position `k` must be "sifted down" into the heap to re-establish the max-heap property. We introduce a new procedure, `siftdown`, for this purpose. The parameters to `siftdown` are the left and right bounds of the heap, as well as the array itself. We implement and verify `siftdown` in the next section.

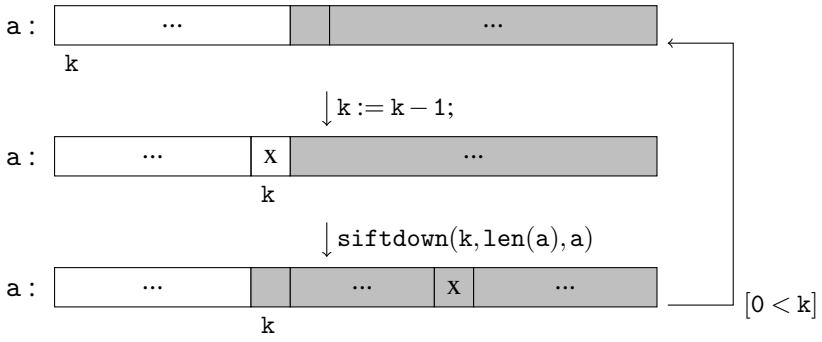


Figure 8.1: Building the heap. The shaded portion satisfies the max-heap property

We now formalize the heap property in the background theory. We extend the `sorting` theory with functions `l` and `r` for the index of the left and right child respectively, and a predicate `heap` expressing that a subrange of an array satisfies

the max-heap property:

$$
\begin{aligned}
&\mathtt{l}(\mathtt{i}:\mathtt{nat}):\mathtt{nat} = 2\times\mathtt{i}+1\\
&\mathtt{r}(\mathtt{i}:\mathtt{nat}):\mathtt{nat} = 2\times\mathtt{i}+2\\
&\mathtt{heap}(\mathtt{a},(\mathtt{m},\mathtt{n}:\mathtt{nat})):\mathtt{bool} = \mathtt{m}\le\mathtt{n}\wedge\mathtt{n}\le\mathtt{len}(\mathtt{a})\wedge\\
&\qquad\qquad (\forall(\mathtt{i}:\mathtt{nat}):\mathtt{m}\le\mathtt{i}\Rightarrow\\
&\qquad\qquad\quad (\mathtt{l}(\mathtt{i})<\mathtt{n}\Rightarrow\mathtt{a}[\mathtt{i}]\ge\mathtt{a}[\mathtt{l}(\mathtt{i})])\wedge\\
&\qquad\qquad\quad (\mathtt{r}(\mathtt{i})<\mathtt{n}\Rightarrow\mathtt{a}[\mathtt{i}]\ge\mathtt{a}[\mathtt{r}(\mathtt{i})]))
\end{aligned}
$$

We then have that BUILDHEAP should maintain $\mathtt{heap}(\mathtt{a},\mathtt{k},\mathtt{len}(\mathtt{a}))$. When the loop terminates, $\mathtt{heap}(\mathtt{a},0,\mathtt{len}(\mathtt{a}))$ should hold.

In situation TEARHEAP, which is entered after BUILDHEAP has completed, we again iterate leftwards, this time maintaining the heap to the left of k, and a sorted subarray to the right of k. The loop is iterated while k is greater than one—when the heap portion contains a single element, this element must be a smallest element, so the array is already sorted at that point. In each iteration, k is first decremented, then the element at index k element is exchanged with the element at index 0 (the root of the heap) to extend the sorted portion. Since the leftmost portion may no longer be a heap, this is followed by a call to `siftdown` starting from index 0 to restore the heap property. Additionally, to infer that the extended right portion is sorted, we also need to know that the array is *partitioned* around k, i.e., that the elements to the left of k are smaller than or equal to the elements to the right of (and at) k. An informal diagram for the TEARHEAP situation and the intermediate states in the loop transition is shown in Figure 8.2. In this figure we have indicated with sloping that a portion is sorted in non-decreasing order.

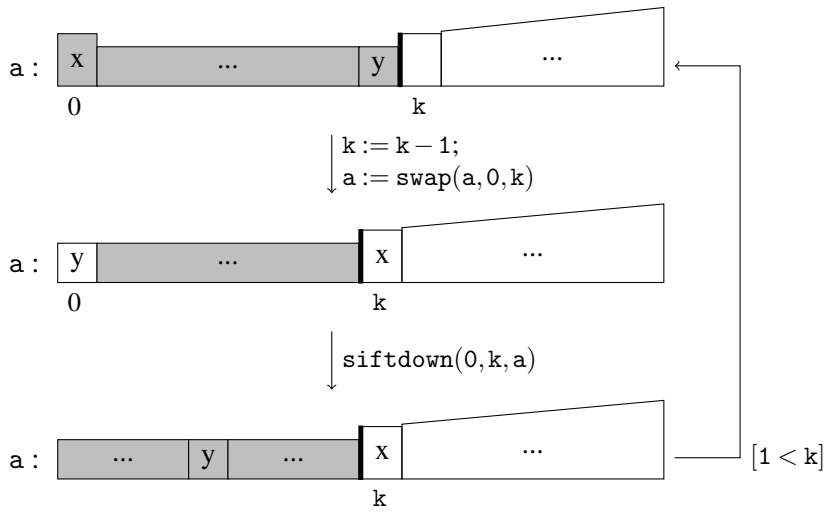


Figure 8.2: Sorting the array. The shaded portion satisfies the max-heap property, the sloping portion is sorted, and the array is partitioned around k

To be able to express the constraints of TEARHEAP concisely we introduce two predicates into the background theory; one expressing that the rightmost portion of an array is sorted, and one that an array is partitioned around a given index:

$$\texttt{sorted}(\texttt{a}, (\texttt{n} : \texttt{upto}(\texttt{len}(\texttt{a})))) : \texttt{bool} = \\ \forall(\texttt{i}, \texttt{j} : \texttt{index}(\texttt{a})) : \texttt{n} \leq \texttt{i} \wedge \texttt{i} < \texttt{j} \Rightarrow \texttt{a}[\texttt{i}] \leq \texttt{a}[\texttt{j}]$$

$$\texttt{partitioned}(\texttt{a}, (\texttt{k} : \texttt{upto}(\texttt{len}(\texttt{a})))) : \texttt{bool} = \\ \forall(\texttt{i}, \texttt{j} : \texttt{index}(\texttt{a})) : \texttt{i} < \texttt{k} \wedge \texttt{k} \leq \texttt{j} \Rightarrow \texttt{a}[\texttt{i}] \leq \texttt{a}[\texttt{j}]$$

With these declarations added to the background theory, we can now give a first situation structure for the procedure `heapsort`. A partial invariant diagram is shown in Figure 8.3. Since CONSTRAINTS is also over the local variable `k`, the postcondition cannot be nested inside CONSTRAINTS; hence we have repeated the constraint $\texttt{perm}(\texttt{a}, \texttt{a}_0)$ in the postcondition.

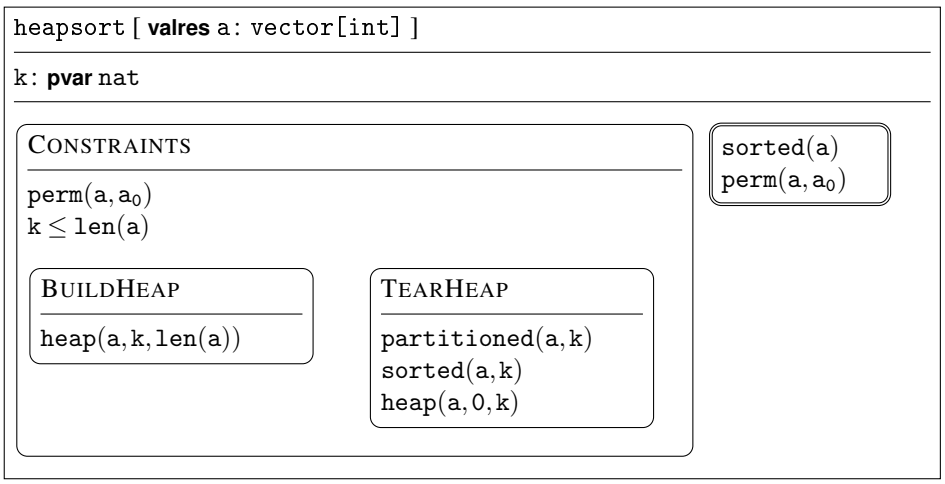


Figure 8.3: Heapsort situations

8.4 Loop initialization and exit transitions

We proceed by adding and verifying the transitions that are not cyclic, i.e., the initial and final transitions, and the transition between BUILDHEAP and TEARHEAP. Since these transitions do not depend on the `siftdown` procedure, they can be checked with Socos immediately. We first consider the initial transition to BUILDHEAP. One possibility is to initialize the loop counter with $\texttt{len}(\texttt{a})$. However, we can do better by noting that $\texttt{heap}(\texttt{a}, \texttt{m}, \texttt{n})$ is true for any `m` and `n` such that $\lfloor \texttt{len}(\texttt{a})/2 \rfloor \leq \texttt{m} \leq \texttt{n} \leq \texttt{len}(\texttt{a})$, and hence that the right half of the unsorted array already satisfies the max-heap property. This is because the elements in the rightmost half are on the last level of the heap and do not have any children within

the range $[\mathtt{m}..\mathtt{n})$. We can confirm this hypothesis by adding the following initial transition

$$\mathtt{k} := \mathtt{floor}(\mathtt{len}(\mathtt{a})/2)$$

and asking Socos to check `heapsort`. Our tool responds that all transitions are consistent, but since the diagram is incomplete, it also points out that the procedure may not be live. We proceed by adding the two exit transitions: from BUILDHEAP to TEARHEAP, and from TEARHEAP to the postcondition. The updated diagram is shown in Figure 8.4. Rechecking the program, Socos confirms that the program is consistent (it is obviously not live, since the transitions do not cover all the possible values of `k` due to the loop transitions being missing). However, before we can add the loop transitions, we need to actually implement and prove the `siftdown` procedure.

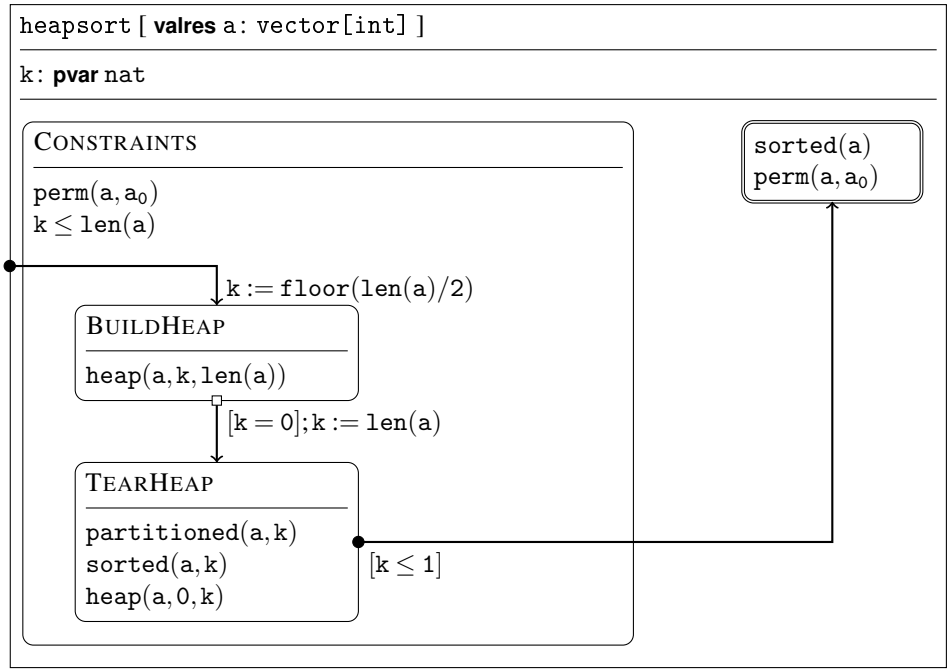


Figure 8.4: `heapsort` with acyclic transitions in place

8.5 The `siftdown` procedure

The parameters to the procedure `siftdown` are the left bound `m`, the right bound `n`, and the array `a`. Assuming the subrange $[\mathtt{m}+1..\mathtt{n})$ satisfies the heap property, `siftdown` should ensure upon completion that the subrange $[\mathtt{m}..\mathtt{n})$ satisfies the heap property, that the subranges $[0..\mathtt{m})$ and $[\mathtt{n}..\mathtt{len}(\mathtt{a}))$ are unchanged, and that the updated array is a permutation of the original array. A pre-post specification is

given in Figure 8.5. Predicate $\texttt{eql}(\texttt{a},\texttt{b},\texttt{i},\texttt{j})$ is defined in the `vector` theory (page 146) and is an abbreviation for the property that two arrays are elementwise equal on the largest common subrange of `a` and `b` entirely within the range $[\texttt{i}..\texttt{j})$.

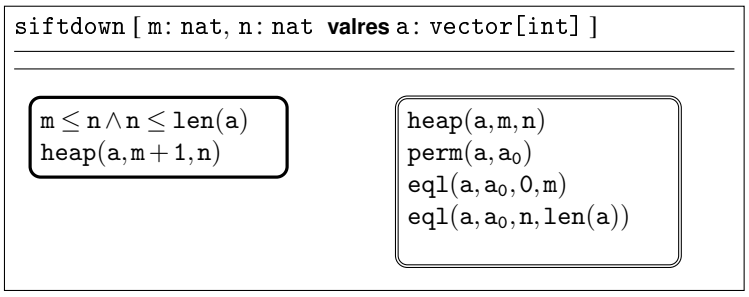


Figure 8.5: `siftdown` specification

The procedure `siftdown` achieves its postcondition by "sifting" the first element in the range downward into the heap until it is either greater than or equal to both its left and right child, or the bottom of the heap has been reached. When either condition is true, the heap property has been restored. Each iteration of the loop swaps the current element with the greater of its children, maintaining the invariant that each element within the heap range, except the current one, is greater than or equal to both its children. The loop statement, using a counter `k` pointing to the current element, is given in Figure 8.6 together with an illustration of the loop invariant. In this figure circles represent elements within the heap range. A shaded circle indicates that an element is known to be greater than or equal to its children. The dashed lines indicate that the parent of `k` is also be known to be greater than or equal to `k`:s children. This part of the invariant is required to prove that the max-heap property holds for the new parent of `k` after swapping. That it is maintained follows from the fact that the child selected for swapping is known to be greater than or equal to its children.

The procedure should return when either the values of both children are less than or equal the current element, or there are no more children within the range of the heap. More precisely, the loop should exit to the postcondition when the following condition holds:

$$\texttt{n} \leq \texttt{r}(\texttt{k}) \vee (\texttt{a}[\texttt{l}(\texttt{k})] \leq \texttt{a}[\texttt{k}] \wedge \texttt{a}[\texttt{r}(\texttt{k})] \leq \texttt{a}[\texttt{k}])$$

Figure 8.7 shows a diagram with an intermediate situation SIFT and the entry, loop and exit transitions in place. We have used *If* commands (square dot) in both choice points to enforce liveness conditions, and we have also provided the variant $\texttt{n} - \texttt{k}$ for situation SIFT and marked both loop transitions as decreasing.

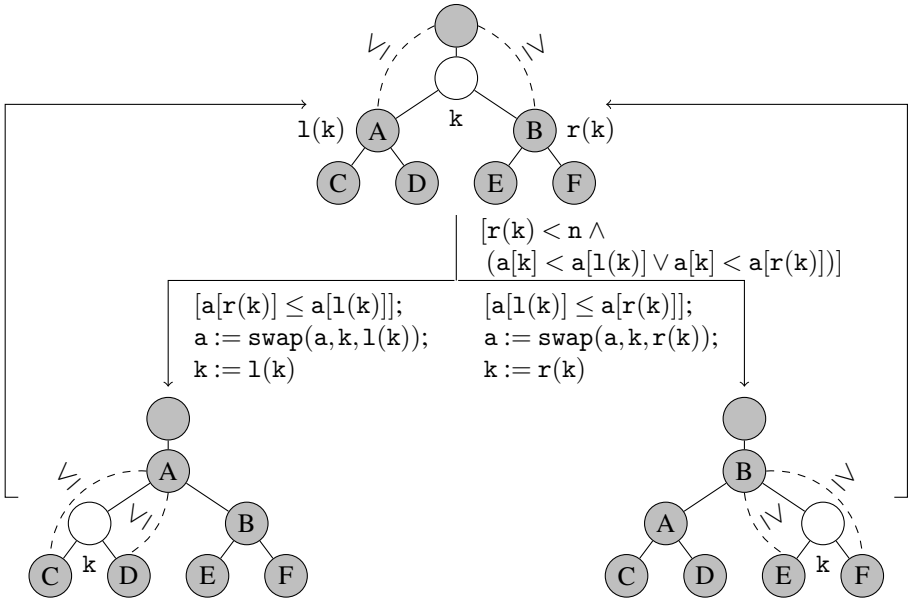


Figure 8.6: The `siftdown` loop invariant

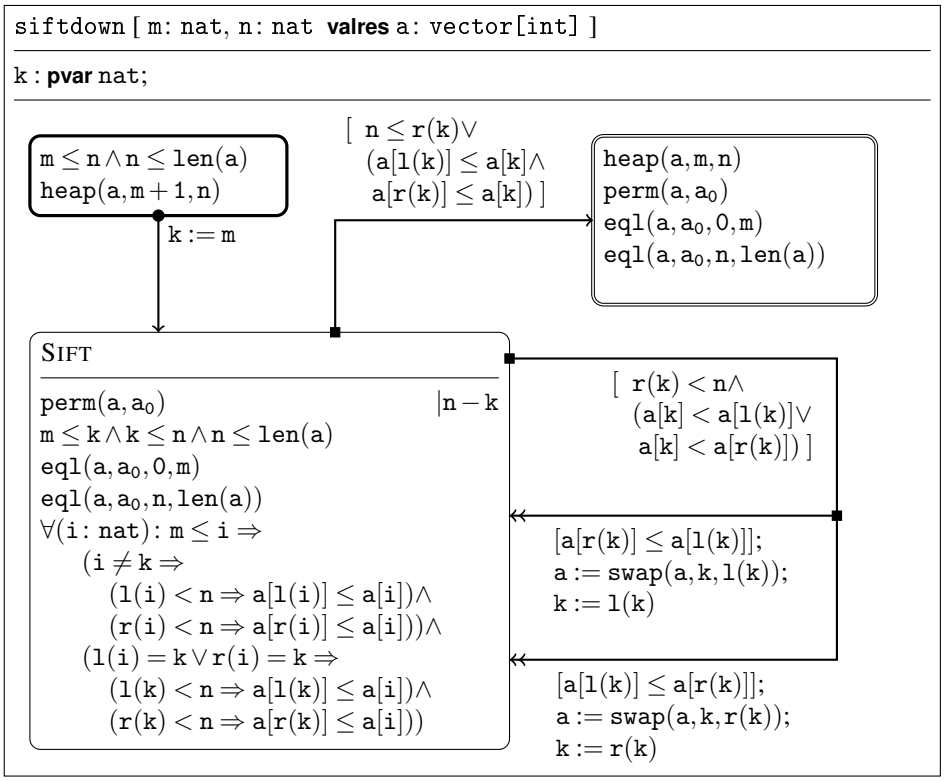


Figure 8.7: A first attempt at `siftdown`

If we try to verify the program, Socos informs us that all transitions except the exit transition were proved; the remaining condition is shown in Figure 8.8.

```
procedure 'siftdown', constraint(s) in transition from
'Loop' to exit:
    [-1]  n <= r(k) OR
           (a[l(k)] <= a[k] AND a[r(k)] <= a[k])
    [-2]  (n <= r(k) OR
            (a[l(k)] <= a[k] AND
              a[r(k)] <= a[k]))
           OR
           (r(k) < n AND
             (a[k] < a[l(k)] OR a[k] < a[r(k)]))
    [-3]  (perm(a, a_0))
    [-4]  m <= k and k <= n and n <= len(a)
    [-5]  eql(a, a_0, 0, m)
    [-6]  eql(a, a_0, n, len(a))
    [-7]  FORALL (i: nat):
            m <= i =>
             (i /= k =>
               (l(i) < n => a[l(i)] <= a[i]) AND
                (r(i) < n => a[r(i)] <= a[i]))
              AND
              ((l(i) = k OR r(i) = k) =>
                (l(k) < n => a[l(k)] <= a[i]) AND
                 (r(k) < n => a[r(k)] <= a[i]))
      |-------
    [1}   (heap(a, m, n))
```

Figure 8.8: Unproven condition for the exit transition from SIFT (the sequent is shown in ASCII, but we have rendered $\texttt{access}(\texttt{a},\texttt{i})$ as $\texttt{a}[\texttt{i}]$ for brevity)

The tool was unable to prove that $\texttt{heap}(\texttt{a},\texttt{m},\texttt{n})$ is established by the exit transition. By inspecting the assumptions it is easy to see that the condition is actually not provable. The reason is an omission of a corner case in the program in Figure 8.7. The problem is that when $\texttt{n} = \texttt{r}(\texttt{k})$, nothing is known about the relation between $\texttt{a}[\texttt{k}]$ and $\texttt{a}[\texttt{l}(\texttt{k})]$. This case occurs when the left child of the current element is the last element in the heap range, and the right child falls just outside of the heap range. To confirm this guess, we can strengthen the first disjunct of the exit guard to $\texttt{n} < \texttt{r}(\texttt{k})$ and re-check. Now, the exit transition is proved consistent, but the liveness assertion for the first branch from SIFT now fails since the case $\texttt{n} = \texttt{r}(\texttt{k})$ is no longer handled.

We resolve the issue by restoring the first disjunct of the exit guard to $\texttt{n} \leq \texttt{r}(\texttt{k})$, and instead handling the corner case in a separate branch of the exit transition which swaps elements $\texttt{k}$ and $\texttt{l}(\texttt{k})$ if $\texttt{a}[\texttt{k}] < \texttt{a}[\texttt{l}(\texttt{k})]$ before exiting to the postcondi-

tion. The updated program can be seen in Figure 8.9. This diagram is a correct implementation of `siftdown`, and now all the VCs (as well as the TCCs) are discharged automatically by `endgame`.

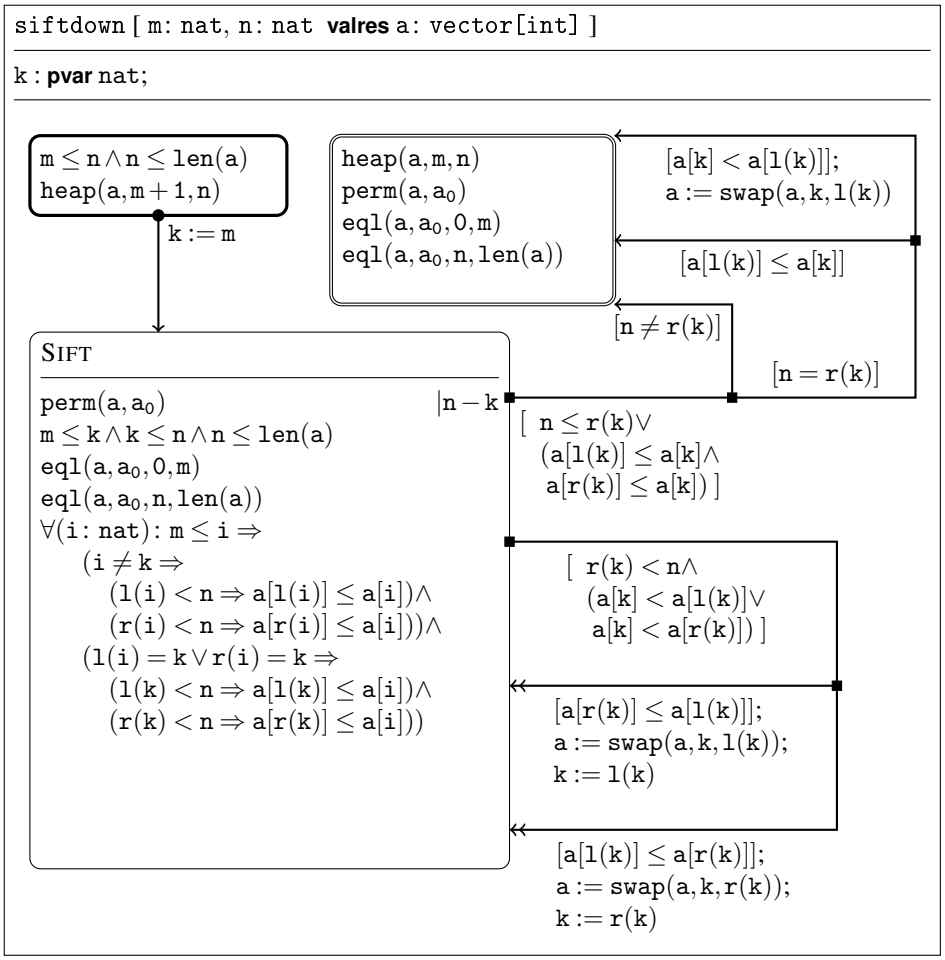


Figure 8.9: Final `siftdown` program, with corrected exit transition. The corner case $\mathtt{n} = \mathtt{r}(\mathtt{k})$ is handled in a separate branch of the exit transition.

8.6 Completing `heapsort`

Using the `siftdown` procedure to implement both missing loop transitions, we can complete the procedure `heapsort`. Figure 8.10 shows the program from Figure 8.4 completed with loop transitions, procedure calls and variants.

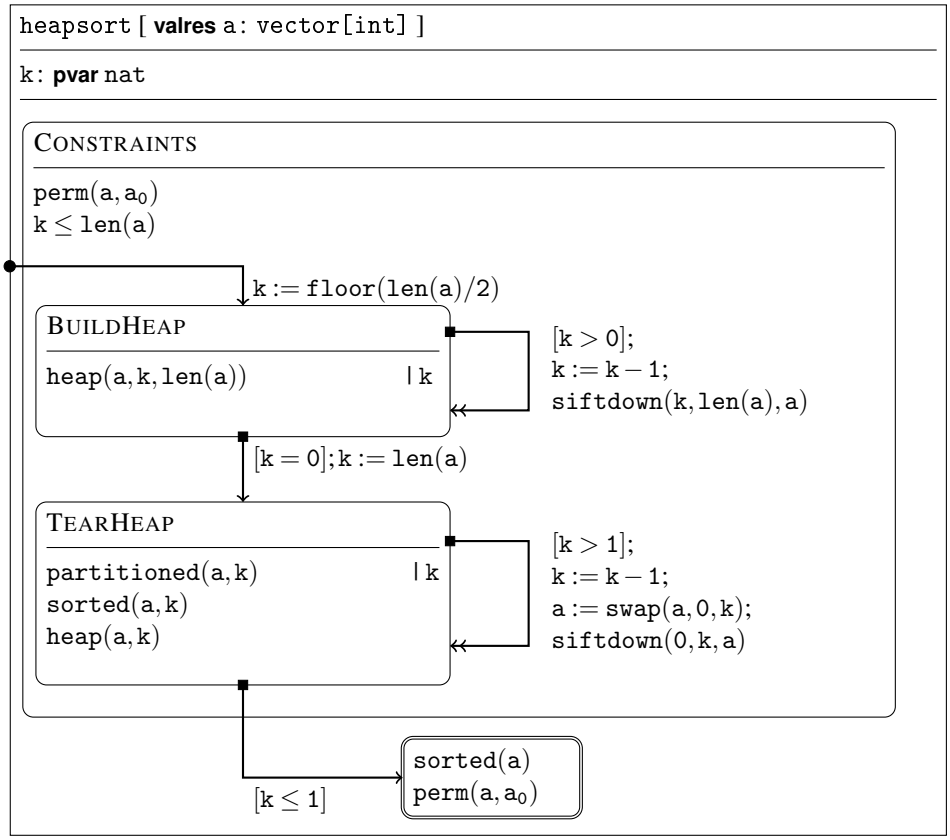


Figure 8.10: `heapsort` with loop transitions in place

If we ask Socos to check the program in Figure 8.10, it discharges all termination and liveness conditions automatically. Socos also proves all transitions except the TEARHEAP loop transition consistent. For this transition, the prover has trouble showing that the loop transition maintains `partitioned`. The unproven condition is listed in Figure 8.11. In this condition the name `a_1` denotes the value of `a` returned by `siftdown`. It seems that the condition is hard to prove because of the way we have defined the postcondition of `siftdown`. This conjecture is correct. `siftdown` manipulates the leftmost portion of the array, and the properties of `perm` given to the automatic prover cannot be used to infer that `partitioned` is maintained throughout the procedure call. In fact, this condition cannot be proved without using the definition of `perm`.

Further analysis reveals that proving the condition actually requires two nontrivial properties: 1) that the root of a max-heap is the maximal element; and 2) that if `partitioned` holds for an index and an array, it also holds for a permutation of the array where the portion to the right of the index is unchanged. One alternative is to start proving this condition directly in PVS. However, it is better to first make properties (1) and (2) explicit in the program by adding assertions to the loop

```
procedure 'heapsort', constraint(s) in transition from
'TearHeap' to 'TearHeap':
    [-1]  0 <= k - 1
    [-2]  k - 1 < k
    [-3]  (heap(a_1, 0, k - 1))
    [-4]  (perm(a_1, swap(a, 0, k - 1)))
    [-5]  (eql(a_1, swap(a, 0, k - 1), 0, 0))
    [-6]  (eql(a_1, swap(a, 0, k - 1), k - 1, len(a_1)))
    [-7]  0 <= k - 1
    [-8]  k - 1 <= len(swap(a, 0, k - 1))
    [-9]  (heap(swap(a, 0, k - 1), 0 + 1, k - 1))
    [-10] k > 1
    [-11] ((k > 1 OR k <= 1))
    [-12] (perm(a, a_0))
    [-13] k <= len(a)
    [-14] (partitioned(a, k))
    [-15] (sorted(a, k))
    [-16] (heap(a, k))
      |-------
    [1]   (partitioned(a_1, k - 1))
```

Figure 8.11: Unproven condition for loop transition from TEARHEAP

transition. In this case we update the TEARHEAP loop transition statement to the following:

$$[\mathtt{k} > 1]; \mathtt{k} := \mathtt{k} - 1; \{\forall(\mathtt{i} : \mathtt{index}(\mathtt{a})) : \mathtt{i} \leq \mathtt{k} \Rightarrow \mathtt{a}[\mathtt{i}] \leq \mathtt{a}[0]\};$$
$$\mathtt{a} := \mathtt{swap}(\mathtt{a}, 0, \mathtt{k}); \{\mathtt{partitioned}(\mathtt{a}, \mathtt{k})\}; \mathtt{siftdown}(0, \mathtt{k}, \mathtt{a})$$

Re-checking the program, we are now left with two simpler conditions: the first assertion, and the same condition as in Figure 8.11 but with the added assertions as additional antecedents. The second assertion is discharged automatically. The first assertion requires an induction proof, but is straightforward. Proving that $\mathtt{partitioned}(\mathtt{a\_1}, \mathtt{k}-1)$ is a consequence of $\mathtt{partitioned}(\mathtt{swap}(\mathtt{a}, 0, \mathtt{k}-1), \mathtt{k}-1)$ and the antecedents in Figure 8.11 is much more involved, requiring reasoning in terms of the definition of permutation and applying properties of bijective functions.

The background theory listed in Section A.3 includes two additional lemmas, `heap_max` and `perm_partitioned`, which we have proved in PVS and which can be directly used to discharge the above conditions. The automatic tactic is not, however, as such able to find the right instantiations to discharge the conditions automatically. This is because the definitions of `heap` and `partitioned` are expanded, making it more difficult to find a match. If we turn off auto-rewriting for these definitions in the above branches of the proof and then apply `endgame`, both goals are discharged automatically.

8.7 Summary and discussion

In this chapter we have shown how PVS and Socos can be used to build a correct invariant-based sorting program. First a basic background theory about sorting and permutations was built. After that the invariants were identified and the situation structure of the program was defined. The transitions were then added one by one, with consistency being maintained throughout the development process. The product of the example is a fully mechanized proof of the consistency, liveness and termination of the implementations of `heapsort` and `siftdown`. With the basic background theory in place, all transitions except one were proved automatically. When the properties `heap_max` and `perm_partitioned` had been identified and added to the background theory, full automation could be achieved.

An SMT solver such as Yices is powerful enough to automatically discharge most of the simpler conditions. By pinpointing problematic conditions, the output from the verification is useful even when the automatic strategy is unable to discharge a valid condition. For such conditions we have a number of alternatives:

- Add an assume statement to achieve consistency at the cost of liveness. This may be a valid alternative if full formal verification is not necessary or we are, e.g., satisfied with using testing for the parts that could not be verified.
- Add an assert statement to isolate a specific condition on which the correctness proof hinges. This condition can then be handled using one of the alternatives listed here.
- Prove the condition in the PVS proof assistant. This method is mainly useful for verifying the final version of a program, and has the disadvantage that proofs must be rechecked when the program is changed.
- Add a lemma to the background theory. This allows the lemma to be reused in other contexts, and to be automatically applied by Yices. This option must be used judiciously, since sending a large number of lemmas to Yices may cause to it to hit its time and/or quantifier instantiation bounds.
- A final option is to identify and add intermediate situations. This may be merited if the intermediate states are complex and further decomposition of the proof is desired. The disadvantage is that the program structure becomes more complicated.

We have shown how our tool integrates the specification, implementation and verification activities into a single workflow. The possibility of verifying incomplete procedures allows a very fine-grained, incremental correct-by-construction approach. Our experience is that careful formulation of the invariants, as well as introduction of useful theorems that can be reused, are key to the verification process and allow a high degree of automation to be achieved. The heapsort

example used background theories extensively to make automatic verification involving non-trivial properties manageable. It extended the basic `vector` theory with properties of sortedness, heaps and permutations, and a set of lemmas useful for automatically discharging simple conditions. The correctness of the `heapsort` procedure is based on these rather general lemmas, which once proved, could also be reused in other verification contexts. The actual application of the lemmas to verify individual transitions was completely automatic.

Our experience is that an automated workflow based on PVS and Yices provides substantial aid in the construction and verification of invariant-based programs. We have so far primarily focused on verification of algorithmic programs, which are typically rather small but may be tricky to specify and/or verify. Examples include sorting, searching, graph algorithms and dynamic programming. In these cases, a high degree of automation generally requires well developed background theories. Simpler programs and toy examples can usually be verified using only the basic system, and in this case the conditions are quite often simple enough that an SMT solver can discharge them all. Verifying such programs is thus plausible even for users not familiar with theorem provers and interactive proof assistants. This gives opportunity for applications of the tool in teaching.

Chapter 9

Case study: Socos in Teaching

This chapter presents a descriptive case study to evaluate IBP as well as the use of Socos in the context of a course given to first and second year students in 2007. We discuss the setup of the case study, the data collection methods used, and the obtained results.

9.1 Introduction

Considering that logic and discrete mathematics are fundamental in the skillset of the software engineer in the same way that continuous mathematics are essential in the traditional engineering disciplines [93], one would expect this to be reflected in the education of future software engineers. However, prominent CS academics such as Dijkstra [67] and Gries [85, 86] have long criticized the diminishing role of mathematics in CS education. While a traditional CS program typically includes several courses in logic and discrete mathematics, the theory learned in these courses is often not put into practice in the programming courses [137]. Moreover, the role of mathematics in CS may be deliberately downplayed by faculty [105].

In IBP, programming is addressed as a hands-on mathematical problem-solving activity; in particular, mathematical reasoning is integrated seamlessly into the process of producing a correct implementation from a specification. The IBP workflow of constructing correct programs is quite similar to that of constructing mathematical proofs. One first writes down the specification of the program; the pre- and postcondition correspond to the assumptions and goals of the proof, respectively. Formulating the specification in mathematical logic requires reasoning in—and potentially developing—the appropriate background theory for the program domain. In the following step, one selects a concrete algorithm for implementing the specification and describes its intermediate situations using the same mathematical logic. This corresponds to a proof strategy that breaks down the proof into a set of lemmas. Finally, one checks the individual lemmas to complete the proof.

IBP has been studied in teaching experiments by Back [16] since 2004. The initial experiments were in the form of ad-hoc programming sessions in which a pair or small group of programmers with no prior experience of IBP solved a programming problem on the whiteboard while being closely monitored by researchers. The feedback from these initial experiments was promising and prompted the expansion of the method into the classroom. The first organized teaching of IBP was in 2005 as a segment of an advanced course aimed at graduate students. Back has since then founded the IMPEd resource center [49] to improve the mathematics and programming education in the Åbo Akademi CS undergraduate program as well as in Finnish high schools. The center promulgates a CS undergraduate program that includes a stack of courses in mathematical logic, structured derivations and IBP. As a pilot study, the IBP course was given in spring 2007 as an elective course for first and second year students. Since then IBP has become the primary vehicle at Åbo Akademi University for teaching introductory formal methods: the 2007 course was re-iterated in spring 2008, and in 2009 it became a compulsory course in the undergraduate program. This chapter describes the 2007 pilot study [19], and focuses on the use of Socos in the course.

The remainder of this chapter is structured as follows. Section 9.2 discusses the general role of tools in formal methods courses. Section 9.3 describes the course syllabus. Section 9.4 defines the study: objectives, methods, and results. Section 9.5 discusses the results. The chapter ends with a summary, discussion of related work, and conclusions.

9.2 Tools in formal methods education

Verification tools are often introduced to students only in advanced level courses, where the focus is on industrial strength formal methods, model checking, or theorem proving. The value of integrating formal methods tools into the CS curriculum has been recognized by several educators [6, 74, 156], with a key point being that students should be made aware of existing tools. Since these tools require considerable expertise, learning the tool itself is usually the main objective in such courses. Verification tools can also play a role in the curriculum as a support for teaching and learning. Tools provide immediate feedback, and students are motivated by interactions that allow them to work in small increments with continuous feedback. The edit–compile–debug regimen taught in traditional programming courses surely encourages learning–by–doing: the feedback obtained from bouncing a program against the compiler or a test case allows the program to be "explored", improving the programmer's understanding of it. Furthermore, tools identify typos and careless errors, freeing up time and mental capacity for other tasks.

A tool is useful in practice even if its technical underpinnings are not understood in detail by its user. When teaching programming, we do not expect students to

produce working, or even syntactically correct, programs without the aid of a compiler. We let them experiment and learn by interacting with a compiler, and we do this before teaching compiler technology. Similarly, when teaching verification it is not reasonable to expect a student to laboriously develop, in full mathematical detail, a calculational style proof of a program without any intermediate feedback in the process. Our hypothesis is that a tool such as Socos can provide value even if the students are not familiar with the underlying theorem prover technology.

9.3 Undergraduate course in IBP

The first undergraduate course on IBP was given in the third period (January–March 2007) of the academic year 2006–2007 at Åbo Akademi University. The course was elective, and targeted mainly first and second year computer science and computer engineering students. 5 ECTS credits were awarded for completion of the course.

The overall design goals of the course were as follows. Firstly, it should serve to decrease the gap between practical programming and formal methods. This means that correctness checking should be introduced from day one as an integral part of the program development process. Secondly, the course should present correctness concepts in a way that is not off-putting to students. It should in particular abstain from highly formal descriptions and instead introduce IBP in a hands-on, example-driven fashion using only the minimal background theory necessary to reason about a specific problem domain. Thirdly, it should strive to inculcate into students that mathematical reasoning is a practical, powerful tool that they can apply in their later software engineering careers.

Familiarity with formal methods, verification tools or even programming was not required to take the course. The only prerequisite was basic knowledge of predicate logic. All students who had taken the standard introductory logic course given to all CS students at the beginning of their first study year were eligible to take the course.

9.3.1 Syllabus

The syllabus of the course comprised the following segments, delivered over eleven lectures and six problem sessions: 1) introduction to IBP, 2) calculational proofs, 3) background theory (arrays, sortedness, permutations), and 4) tool-supported verification using the Socos prototype. Both lectures and sessions were 90 minutes each. Segments (1) to (3) were introduced without tool support and comprised eight lectures and four sessions in total. Due to unfamiliarity of the target audience with mathematical modeling and proofs, a large share (five lectures) of the time was allocated to segments (2) and (3). Segment (4) included two lectures of Socos introduction and practical demonstration, and two problem sessions.

All problems sessions involved invariant diagram construction and verification. Problems were given in increasing order of difficulty. The first exercises consisted of filling in invariants and proving transitions for already given programs, while later exercises only gave an informal problem description and required the student to formulate the pre- and postconditions as well as construct and verify the solution program. Problem sessions for segments (1) to (3) took place in classrooms and involved teacher assistance, but included homework assignments to be solved independently and handed in at the next session. The students wrote their solutions using pen and paper. The Socos problem sessions took place in a computer lab in which Socos was available on all computers, and solutions were handed in electronically. Teacher assistance was available also in the lab.

The course finished with a written exam, in which no software or other support material was allowed. To pass the course, a student should achieve at least 50 per cent of the score of each individual graded homework assignments, as well as 50 per cent on the final exam.

9.3.2 Use of Socos

We felt it was important to introduce IBP without tool support so that students could focus on the fundamentals and not be overwhelmed with several concepts in the beginning of the course. But it is evident that once the method has been learned and exercised a few times using pen and paper, the burden of hand proofs quickly becomes significant even for relatively simple programs. The student must laboriously write down and check verbose conditions—many of which are identical to or minor variations of already proven ones. Due to the absence of immediate feedback, careless errors may not be detected until several pages of proofs have been written. At this point, increasing the scope of problems without providing any kind of automation only subjects students to repetitive drudgery. The method could then be experienced as frustrating, motivation would falter, and students may leave with a potentially long-lasting bad taste of verification. The other option, limiting the scope of problems, on the other hand risks implanting into students that verification is limited to toy examples.

In our setting there was also a practical aspect to the problem. The exercise segment of an introductory CS course at Åbo Akademi University involves a string of 5–10 problem sets that are to be completed and handed in by the student. The problem sets are usually given weekly and each set is either solved during a 90 minute classroom session, given as a homework assignment with a tight deadline (one week), or as a combination. Problem sets are expected to require at most a few hours work per set to complete for the average student. Given the time-demanding nature of writing down proper proofs this significantly limits the scope of the problems that can be given.

Our approach was to increase the difficulty of the problems while at the same time “easing in” tool support, with the goal of keeping the workload roughly

unchanged, but letting the students solve harder problems. Students would first be subjected to a demonstration of a Socos verification of a known problem, i.e., one they had already solved using pen and paper. In the next stage, students would solve a simple problem on their own to get accustomed to the tool and its capabilities. Given that they at this point have some experience of hand verification, we expected them to welcome the addition of automation. Finally, we ramped up the difficulty of the problems for the final assignment.

A side benefit of integrating Socos into the course was that the constructed programs could be executed. Constructing a program that can be executed is likely more motivating than writing programs on paper.

Note on version. While the technical description and examples presented so far in this thesis have pertained to Socos_2, the case study in teaching was carried out using Socos_1 and so the examples in this chapter will be presented in the context of that version. The main difference on the user level is the notation—Socos_1 uses a syntax based on the MathEdit mathematics editor [18], whereas Socos_2 is based on the PVS language. The Socos_1 notation is less rich than the PVS based notation, and is Unicode-based rather than ASCII-based. It includes the standard logical connectives, quantifiers, arithmetic operators, and constant and variable definitions. The back-end verifier also differs—Socos_1 uses Simplify, whereas Socos_2 uses PVS and Yices. Simplify [62] is a validity checker that supports linear arithmetic, interpreted functions with equality, arrays (maps) and quantifier instantiation based on E-graph-matching. Simplify belongs to a generation of validity checkers that has now been succeeded by SMT solvers; it is, however, still a powerful proof tool and well capable of handling most of the simple conditions generated from the examples in this course. Socos_1 supports the definition of uninterpreted functions and axioms, which are sent to Simplify, but it does not interface with PVS theories.

9.4 The study

The study was part of an overall inquiry into the educational merits of IBP verification education, which we described in the report [19]. We focus here only on the use of Socos in the course. The study follows the principles of descriptive case study in computer science education research [78, pp. 47-48]. We address the following research questions:

1. *What educational value does Socos provide?*
2. *If students have knowledge of IBP but no prior tool experience, are they able to quickly start using Socos?*
3. *Given experience of pen and paper verification, are students able to effectively tackle bigger problems using the tool?*

4. *Does the tool add perceived value to the course from the point of view of the students?*

5. *Does the tool add perceived value to the course from the point of view of the teacher?*

6. *Is the Socos tool considered easy or difficult to use? How can we improve it?*

7. *How can we streamline Socos integration into teaching?*

9.4.1 Methodology

The undergraduate course was elective and the CS student body at Åbo Akademi University is rather small. Altogether ten students were active in the Socos part of the course. Due to the small number of participants, we adopted a multi-faceted approach to data collection to improve the trustworthiness of the results. Data was collected based on classroom observation, analysis of homework assignments, and a post-course questionnaire.

Classroom observation. The teacher assisting in the Socos problem sessions monitored the students while they were working with the tool, taking notes of common problems and other issues.

Homework assignments. Students sent solutions to the Socos homework assignments directly to the teacher as e-mail attachments for grading. The results presented are based on assignments handed in by 7 students.

Questionnaire. The post-course questionnaire included five multiple choice questions using Likert-type scales and one open ended question asking the students their opinions about Socos and the course in general. Answers were anonymous. The results presented are based on the analysis of 10 questionnaires.

9.4.2 Problem sessions

We introduced the tool segment of the course with a brief overview of the principles of automatic proving. The goal here was to give a "feel" for the capabilities of the automatic prover (Simplify), so as not to raise unrealistic expectations. Students were introduced in an informal way to the expressiveness and limitations of first-order logic, as well as to capabilities of Simplify. We did not include PVS at all in the course.

Students started the first problem session by individually going through an introductory tutorial to Socos. Actual problems were then given in two sets of three problems each. Students had one week to complete each set. The first set was not graded, and model solutions were presented in class by the teacher. The

second set was handed in and graded, with solutions made available to students on line after the deadline. Students were instructed to supply manual proofs for transitions which could not be proved by Simplify. Such proofs could be entered into a free-text field in Socos and saved together with the program, and we also allowed students to manually prove either the actual condition or a "helper" lemma sent to the theorem prover. Computers with Socos installed were accessible in the lab at all times but students were encouraged to be present at the sessions to benefit from the teacher assistance.

Ungraded problems. The following problems were given in the ungraded set, in increasing order of difficulty:

UP1) Finding the index of the largest element in a non-empty integer array.

UP2) Reversing an array in-place.

UP3) Raising an integer to the power of a positive integer using an efficient algorithm. Students were given a background theory with the following properties: $x^0 = 1$, $x^e = x^{e/2}x^{e/2}$ and $x^e = x^{e-1}x$.

Graded problems. Grading for the Socos assignments was based on the number of verified transitions and how precisely the pre- and postconditions were expressed. The maximal score was set for each problem based on its difficulty. A student's score was calculated from the number of proven VCs on a *pro rata* basis. Erroneous pre-/postconditions were penalized with a 60 per cent deduction of the score for the problem in question; this was intended to encourage care with this rather important part of the solution as well as to discourage simplifying the problem by changing the specification. The problems given in the graded set and their maximal scores were as follows:

GP1) Summing the elements of an integer array (5 points).

GP2) Binary search to determine if an integer value exists in a sorted array (5 points).

GP3) Sorting an integer array using a freely chosen algorithm ("simpler" quadratic algorithms such as insertion sort earned at most 10 points, whereas "harder" efficient algorithms such as quicksort earned at most 15 points). Only selection sort was excluded from the choices, since it was a familiar problem from the pen and paper part of the course. Students were in advance given a background theory with an equivalence relation *permutation* over arrays defined, together with an already verified swapping procedure maintaining *permutation*.

The maximum total score was 20 points, of which 10 or more were required to pass. Extra points earned by selecting a harder algorithm in GP3 could be used to make up for lost points for GP1 and GP2, but not to raise the total score tally above 20. Ten students attended the Socos part of the course regularly, and seven students handed in solutions to the graded problems within the time frame. Their scores are summarized in Table 9.1. The three remaining students out of the ten active participants did not hand in solutions in time. In one of these cases the student had been absent from the introductory sessions and subsequently lacked basic knowledge of the tool. These students were allowed to participate in the written exam, and were later given a new set of Socos problems to solve in order to earn the final credits for the course.[1] While these students later passed the course they were not accounted for in this study.

Student	*GP1. Array sum (5 p)*	*GP2. Binary search (5 p)*	*GP3. Sorting (10/15 p)*	*Total*
S1	1	1	2 (IS)	4
S2	1	5	5 (IS)	11
S3	4	3	5 (IS)	12
S4	5	5	10 (IS)	20
S5	3	1	9 (IS)	13
S6	3	4	10 (QS)	17
S7	5	5	10 (MS)	20
Average	**3.14**	**3.43**	**7.29**	**13.86**

Table 9.1: Student scores for graded problem set
IS=insertion sort, QS=quicksort, MS=mergesort

The table shows that six students (S2–S7) passed the assignment, while one (S1) failed. In GP3, five students chose to implement insertion sort, while S6 opted for the more difficult quicksort and S7 for mergesort.

9.4.3 Classroom observation

As we were interested in evaluating the Socos workflow and identifying potential usability issues, the teacher assisting the students in the lab sessions actively monitored the students' use of Socos. We did not use a systematic data collection

[1] At Åbo Akademi University, it is common to allow failed students to compensate by solving a set of supplementary assignments. However, solutions to the original problems have usually been posted at that point, so the supplementary assignments must be new and it is thus difficult to compare their answers to the original answers. We therefore omitted them from the study.

method so we present here only a collection of key observations.

Starting by going through the Socos tutorial in a common lab session was a good choice, since the threshold for asking questions was much lower compared to had we used e-mails or other written correspondence. A number of practical questions regarding the user interface that were initially not covered in the tutorial surfaced in this session, such as how to add intermediate points to a transition, how to layout diagrams, and how to manage program files in Socos.

Knowing *what* to do, but not *how* to do it, is a common source of frustration when working with a software tool. While the syntax used in the pen and paper part of the course was very similar to the Socos syntax, minor discrepancies were still present, e.g. in the ways arrays were declared and used. Also, procedures and formal parameters, while familiar concepts from introductory programming courses, were not used in the pen and paper part of the course. Students were initially instructed to look at and redo Socos implementations of programs they had already solved using pen and paper. It was felt that solving such familiar problems smoothed out learning curve bumps from syntactic and methodological idiosyncrasies, and allowed students to quickly assimilate "the Socos way" of building the same program.

Students did not have difficulty with the Socos workflow. They learned quickly to apply the tool in the intended way and to benefit from the dialog with the automatic prover. The direct feedback allowed errors and omissions to be detected immediately, rather than being lost in complicated derivations. This allowed students to put more effort into the formulation of the invariants. As a result, we felt that their understanding of the programs being verified deepened.

We consider separately two classes of usability issues, *intrinsic* and *accidental*. Intrinsic issues are specific to the method and are directly related to the intended workflow and the decisions made when designing the tool. They should be considered the graver category of usability issues. Accidental issues are caused by factors not essential to the method. Most of these are related to the prototype nature of the Socos tool and avoidable given sufficient development resources for the tool. However, we think that the accidental issues are still interesting as they highlight problems specific to tool integration into CS courses, where most users are novices.

Intrinsic issues

I1) The unproven VCs reported by Socos were perhaps not sufficiently emphasized in the user interface. They were displayed in a pane at the bottom of the Socos window, which we felt could have downplayed their significance.

I2) Every Socos program must be encapsulated into a procedure. Reasoning with procedures and parameter types was not treated in the theory part, and the semantics of the various parameter kinds was also new to the students.

I3) Management of diagrams in the diagram editor was perceived as tricky at

times. The most problematic tasks were resizing a situation to make room for nested situations and arranging transition labels neatly.

Accidental issues

A1) Syntax change. For instance, the Socos integer type is called Int, while the IBP theory segment used integer. Also, Socos does not support compact, mathematics style inequality syntax such as $0 \leq x < N$ (this must be expressed instead as $0 \leq x \wedge x < N$). Additionally the array update syntax was slightly different.

A2) Some students found the mechanism for typing operators and special symbols difficult. The Socos prototype uses the Mathedit [18] input method, which involves either picking the right symbol from a long menu, or learning a three letter key combination. The first option is inefficient for frequent symbols, and while the key combinations were mnemonic, it may not be easy to remember a large set of them.

A3) In Socos_1, defining a function or predicate involves creating a Mathedit type definition, giving the type and arity of the function, and a Mathedit rule, giving the body of the function. This proved to be difficult due to the scaffolding syntax required.

A4) Some syntax and type error messages were cryptic, since the Socos parser has not been fine-tuned to produce highly informative error messages; e.g., the type checker reports only that a term is not of an expected type, not the actual inferred type of the term. In a few cases, resolving such errors was unnecessarily time-consuming.

A5) Generation of VCs in the Socos prototype is rather slow, mainly due to inefficiencies in the MathEdit rule processing engine. Generating the VCs for a small program can take up to a few seconds.

A6) One student tried to install Socos at home, but experienced problems installing the correct support libraries and compiling the code.

We propose some remedies to these difficulties in Section 9.5.

9.4.4 Questionnaire result

Table 9.2 lists the questions asked in the post-course questionnaire and the arithmetic averages of the answers based on 10 respondents.

The results indicate that the students found Socos to be useful, and also that they welcome the use of tools in theoretical courses. However, they also reported some difficulties associated with the use of the tool. Four students answered

	Statement	*Average*
Q1.	The Socos tool was easy to use	**3.7**
Q2.	Automatic verification with Simplify was useful for proving programs correct	**2.1**
Q3.	I was more confident about the correctness of the programs constructed with the tool compared to the ones constructed using pen and paper	**2.1**
Q4.	Given adequate tool support, invariant-based programming could be used for programs of larger scales	**2.8**
Q5.	It is generally a good idea to include tool support in theoretical courses	**1.8**

Table 9.2: Post-course questionnaire results
(scale 1-5, 1 = completely agree, 3 = neutral, 5 = completely disagree)

the open ended questions; their answers supported these findings. All usability issues reported were related to syntax and the lack of a language reference manual. Several students wished that more time had been allotted to the Socos segment of the course.

9.5 Discussion

We will now revisit the research questions from Section 9.4 in light of the collected data.

1. What educational value does Socos provide? The course participants performed well in graded homework assignments, with two students scoring the highest 20 points, and with an average score of 13.86. While we acknowledge that success in assignments is not a guarantee for learning, the observation that students at the end of the course were able to independently construct and verify with full mathematical rigor non-trivial programs (in particular, GP3) indicates that tool-supported IBP gave students a good insight into program verification. Also, as pointed out in [19], it should be kept in mind when interpreting the results that the participating students were complete novices; many of them had prior to the course no notion of concepts such as "precondition" and "invariant".

Writing calculational proofs was considered by students to be the most difficult and time-consuming step in hand verification of invariant-based programs [19]. In the Socos segment of the course students had to write only a minimal amount

of proofs and more time was available for identifying the invariants, something we felt deepened the understanding of the problems. We noticed that the students gained some important insights this way. For example, a typical novice mistake in the loop invariant for problem UP2 (reversing an array) is shown in Figure 9.1.

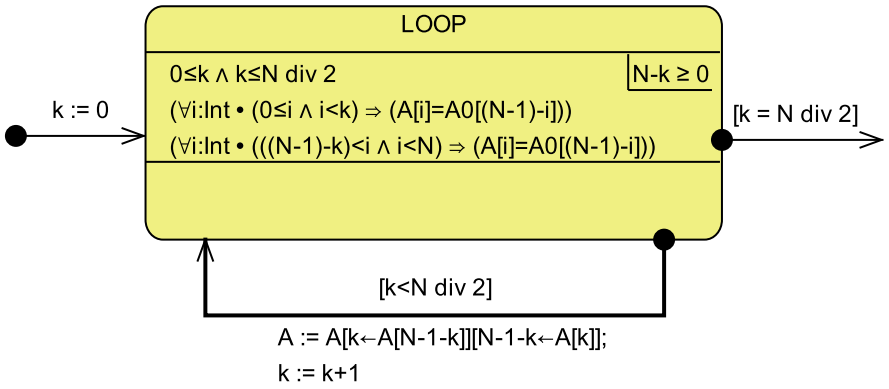


Figure 9.1: Fragment of incorrect solution to UP2. The loop invariant is too weak.

The LOOP situation relates A to the original A0 in the already processed portions $[0\,..\,k)$ and $(N-1-k\,..\,N)$. This invariant is, however, too weak since nothing is stated about the intermediate portion of the array, $[k\,..\,N-1-k]$, and the loop transition consequently cannot be proved consistent. We observed that several students who at first were confident that the above loop invariant was sufficient and that the program was provable, when asked to examine the unproven VCs without regard to the operational structure of the program quickly realized that the given assumptions were too weak. This requirement to state not only what changes, but also what does *not* change, is an important element of invariant formulation that is not obvious to novices (especially not to those used to an operational way of thinking). After the invariant was strengthened in the proper way, Simplify proved the transition automatically. We observed that this immediate confirmation from the tool contributed to the satisfaction felt by the student when the error was corrected. While the above error would be detected in a meticulously written hand proof as well, the lack of machine checking and immediate feedback induces a risk of the student using an unfounded assumption to "finish" the proof, or becoming frustrated with a derivation leading nowhere and giving up without suspecting the error being in the invariant formulation.

We were aware that introducing proof automation in a basic course carries with it the risk of students misunderstanding and misjudging the capabilities of the automatic prover. While students are not gullible, they may have unreasonably high expectations and start applying the tool in a futile guess–check–modify cycle in place of trying to understand the programming problem properly. However, this risk did not manifest itself. We believe that the theoretical part of the course had

given the students sufficient insight into the difficulty of the verification problem to understand that such random testing would not converge into a correct program. We felt that the practical use of Socos rather quickly imparted students with a good basic understanding of the strengths and limitations of the automatic theorem prover. Also, by first being subjected to manual verification, students may have learned to better appreciate the automation. In summary, we think that the course worked well in familiarizing students with computer aided verification and provided a good starting point for more advanced theorem proving courses.

2. If students have knowledge of IBP but no prior tool experience, are they able to quickly start using Socos? As the tool segment of the course had a very tight schedule (two lectures + two problem sessions at the end of the course) this question was put to a heavy test. We felt that the student quickly grasped the general idea of how to use Socos to develop programs. One student commented that *"[...] it was rather straightforward to understand the idea of the Socos tool and how to apply it."* This came as no surprise, since the manual IBP workflow taught in the course and the Socos workflow are essentially the same.

We noticed that reiterating a known problem from the theory segment in the beginning of the Socos segment turned out to be a good way of highlighting the syntactical differences between the unparsed "pseudo-code" used in the former and the machine-checked code used in the latter. The palindrome program allowed the students to identify and absorb these idiosyncrasies in a known context, reducing frustration later.

Based on the feedback we received having to learn basic reasoning with procedures (I2) was not perceived as a significant hurdle by the students. We did, however, feel that it made the transition from theory to tool somewhat less smooth than it could have been. Possible solutions are to either introduce procedures already in the theory segment, or to support procedureless programming using only global variables in Socos.

3. Given experience of pen and paper verification, are students able to effectively tackle bigger problems using the tool? As we discussed in Section 9.3.2, introductory CS courses at Åbo Akademi University limits the student workload for a single problem set to a few hours. In the final problem before the tool segment of the course students were asked to construct and verify a palindrome program. This was considered to be the limit for pen and paper proofs. All problems given in the tool segment (Section 9.4.2) were either on par with the palindrome program in difficulty (UP1, GP1) or considerably more difficult (UP2–3, GP2–3). The students who picked the more difficult sorting algorithms also performed well. Overall we think that the quality of the student's solutions and the results in the graded problems support this statement.

4. Does the tool add perceived value to the course from the point of view of the students? Our general feeling was that the students clearly showed interest in Socos and found it a nice addition to the course. We were also impressed by their motivation to learn the tool and their diligence in completing the assignments, and it was satisfying to see how their programming and verification competence increased. One student gave the following feedback on the course:

> "*Socos definitely added value to the course. It was interesting to see what a mathematical tool looks like. I've never seen or used one before. But it was good to start with only pen and paper.*"

Results from the questionnaire indicate that students were predominantly positive about the use of automatic theorem proving (Q2), that they found that Socos increased their confidence about the correctness of programs (Q3), and that they welcomed the use of tools in theoretical courses (Q5). On the question whether IBP could be realistically used to construct larger programs (Q4), answers were more negative with the average tending towards neutral. In summary, we conclude that Socos was well received and that the students clearly appreciated an automatic prover, and that they also understood the necessity of having a clear understanding of the underlying theory.

5. Does the tool add perceived value to the course from the point of view of the teacher? We found that using Socos simplified the grading process of homework assignments in two ways. Firstly, offloading the syntax checking to the tool allowed the teacher to focus on the semantic errors. While this is unsurprising, we still think it is worth pointing out, as a quantitative analysis of errors in pen and paper solutions by Erikson [72, p.43] in 2008 reports that 39 per cent of errors were syntactic, and recent analysis of student difficulties with IBP by Mannila [112] also indicate that a significant share of the errors related to the IBP and logic notation could easily be prevented with a syntax checker. Grading exercises with a large number of syntax errors frequently requires the teacher to guess what the student intended, resulting in wasted time and the risk of missing more serious semantic errors.

Secondly, Socos provided a semi-automatic procedure for assigning scores: each submitted solution was checked with Socos and a base score was calculated based on the proportion of VCs automatically proved; the solution was then inspected by hand and the base score adjusted with additional points for manually proven VCs, as well deductions for incorrect pre- or postconditions as described in Section 9.4.2. Since this procedure considers only the number of proven VCs and not their difficulty it must not be applied blindly—it is easy to contrive a program with many trivial transitions to achieve a high ratio of proven VCs. However, we found it to be a practical way of establishing a baseline when grading assignments.

6. Is the Socos tool considered easy or difficult to use? How can we improve it? While students did not seem to have any problems understanding how to build and verify programs in Socos and were able to successfully complete the assignments, the survey indicates that they did find the practical use of the tool somewhat difficult (Q4). Most of the issues reported were related to not knowing how to express things properly (A1–A3). Due to lack of a language reference, the Socos syntax and semantics were taught on the whiteboard and by examples. The only documentation available for the students was the Socos introductory tutorial and a list of key combinations for typing mathematical symbols. As one student commented in the open ended feedback:

> *"The Socos user manual was insufficient (e.g. to define own functions) and the system was difficult to use due to not knowing the syntax."*

We experienced that almost all of the questions and issues raised during the first problem session could have been resolved with a standard reference manual. The usability issues A4–A6 were understandable and expected as the tool is a prototype. All of the syntax issues could be resolved quickly in the classroom by the teacher. While we do not consider these issues to indicate any fundamental flaw in the workflow of Socos, they do call for further refinement of the tool.

7. How can we streamline Socos integration into teaching? It must be admitted that we did not allocate enough time for the Socos segment of the course, and that a main difficulty in this study was lack of time. Even if the students knew the background theory well, introduction of a tool inevitably brings with it a considerable amount of new material to learn. As one student commented:

> *"Difficult to get started* [with Socos]*, and when I finally did, the course was over. Socos was difficult because of the lack of time."*

We think that adjusting the proportion of the Socos segment of the course upwards, together with improving the documentation, should be sufficient to handle most of the issues encountered by students.

We are currently also working on improving more fundamental aspects of the tool's usability in light of this study. One thing we noticed was the verification report (the list of unproven conditions) deserves a more prominent place in the Socos user interface (I1). In the re-designed diagram editor, verification results will be indicated directly in the diagram by color-coding invariants based on the verification status. The new editor has improved layout capabilities to address issue I3.

9.6 Related work

Walther and Schweitzer [152] have developed the Verifun system and evaluated it in the context of a practical course for advanced students and a second year

undergraduate course; the latter was much larger in scope (more than 400 students) than the Socos course. Verifun focuses on the verification of functional programs in an easy to use theorem prover with automation; program specific lemmas can be formulated to help the automatic tactic. While Verifun is mainly geared towards building correctness proofs of existing programs rather than correct-by-construction development, it is designed for classroom use and has similar design goals as Socos. Exercises given in their undergraduate course were similar in difficulty to the Socos problem sessions (the most difficult problem was merge sort), and the reported findings are in-line with ours: the authors feel that a verification system can be used successfully at the undergraduate level, given that the students have a good grasp of the programming notation and understand the role of logic and proofs from the outset. Their study also highlighted the importance of an introductory tutorial.

9.7 Conclusions

We have in this chapter described a case study to evaluate the use of Socos in an introductory program verification course. We have considered the educational aspects of Socos, the feasibility of introducing it to novices, its perceived value by students as well as teachers, and potential usability issues. We have collected data using classroom observation, analysis of homework assignments and anonymous questionnaires.

While the aim of the course was to teach the IBP method rather than a verification tool, we wanted to include the tool in the course both as a learning support and to let students advance beyond trivial examples. However, we consider it essential that the students were introduced to the IBP methodology through pen and paper exercises before being allowed to use Socos. In this way the tool was introduced to support, and not to replace, the already learned workflow. There was no requirement that given a problem had to be proved by the tool, even if it would have been possible; students were still allowed to write manual proofs, which were not machine checked. Thus the students could use the automatic prover as a program construction support rather than merely as a verification oracle, and they found its use valuable even without complete understanding of the underlying theorem proving technology. The feedback cycle allowed them to identify semantical errors that had slipped through in hand proofs, and we felt that their understanding of programming deepened as a result of this.

As Socos was introduced near the end of the course and the students had already learned the IBP methodology, we did not have any fundamental problems with introducing and familiarizing it. Most of the usability issues were due to slight differences in notation and, to some extent, lack of documentation. It did, however, become clear that two weeks was not really enough time to get the students competent with the tool. However, we still think it is a valuable goal in

itself to familiarize a tool in an introductory course, a sentiment also shared by the students who answered the questionnaire. The course also provides a natural bridge to a course in computer aided verification and theorem proving.

Our conclusion from this study is that Socos is a useful teaching and learning support for IBP. The promising results from this pilot course resulted in a re-iteration of the course in 2008, and it has since then become a standard course in the CS curriculum at Åbo Akademi University. We admit that the scope of the pilot study is so small that we cannot draw any hard conclusions based on these results. Also, comparisons with other methods need to be made. However, since IBP is a new method and Socos is an experimental tool, we still think that the study was valuable in that it did not disprove the feasibility of using either one in undergraduate education.

The fact that one of the teachers was also a developer of Socos may have impacted the students' answers to the questionnaire, which could be considered a challenge to the veracity of the results. However, since the questionnaire was post-course and anonymous the students were assured that their answers would not affect their grading, and hence we do not think the candidness of their answers is in question.

Chapter 10

The Socos Project

This chapter describes the development history of Socos. We present the settings in which the tool was developed. We describe the architecture of Socos_2 and its current status of implementation, and we also list the most important ongoing and future work related to tool implementation.

10.1 Introduction

The tool presented in this thesis is a product of the *Socos Project*, a research software development project started in 2005. The project is directed by Ralph-Johan Back and carried out in the context of the Software Construction Laboratory at the Department of Information Technologies, Åbo Akademi University. The Socos Project has evolved symbiotically with three other research projects at the department: TORES II, IMPEd, and Gaudí. Figure 10.1 shows the relationships between Socos and these projects.

The aim of the *TORES II* Project (Tools for Reliable Software Construction, 2005–2008) has been to research and implement an advanced environment to support the construction of reliable software. TORES II has funded Socos tool development and provided a collaborative environment for tool development. Socos_1 and *Coral*, a general purpose modeling framework [5], are the two main software products to come out of the TORES II Project. The relationship between these two products is described in the next section.

The *IMPEd Resource Center* is a joint initiative between the IT departments at Åbo Akademi University and the University of Turku. It focuses on improving the basic mathematics and programming education in high schools and universities through the introduction of new ideas and methods [49]. It is directed by Ralph-Johan Back and provides a resource center for teachers, faculty, researchers, and other interested parties. A particular branch of IMPEd focuses on *practical formal methods* and is dedicated to promoting IBP in programming education. IMPEd has actualized the need for an IBP tool and provided the opportunity to apply and

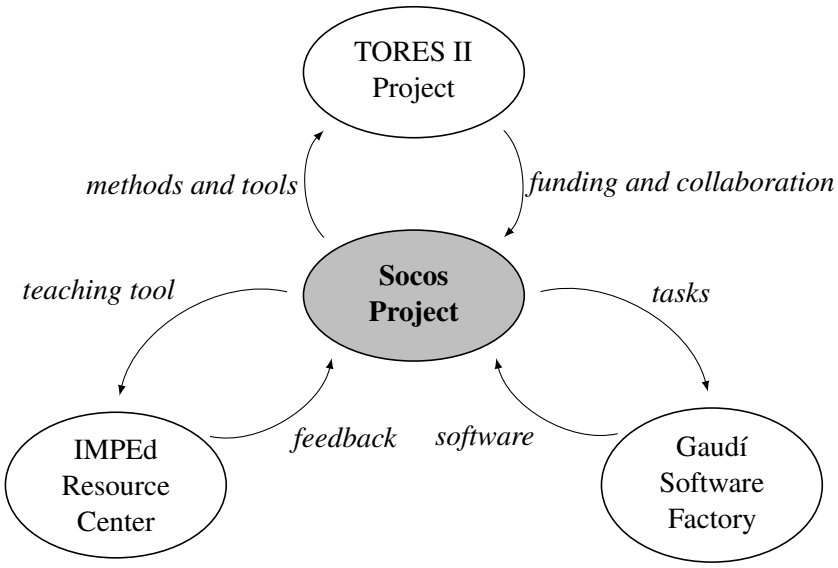


Figure 10.1: Interaction between Socos and research projects

evaluate Socos in teaching.

The *Gaudí Software Factory* is a software production unit in an academic setting [26]. The factory employs programmers from the undergraduate-level CS student body at Åbo Akademi to develop, among other products, research software for the Software Construction Laboratory. Employment is full time for three months (usually in the summer), in which a group of two to four students complete a well-defined subset of a software project. The students are hired as software developers and do not earn academic credit points for their work, and they receive a full salary following the university regulations for research assistants. Software projects in Gaudí follow the *Gaudí Process*, an agile like software process that has been extensively adapted to academic settings. The process includes intensive customer involvement: for research software products, the customer is a research group (internal or external) or an individual researcher or project leader. In addition to providing the infrastructure for software development, the factory also functions as a platform for empirical software process research: researchers at the laboratory closely monitor the projects in Gaudí and use the factory as a sandbox for experimental software process evaluation and improvement [27, 117]. In this way Socos has also contributed to the corpus of software available for empirical evaluation within Gaudí.

Development of Socos has comprised three main stages: 1) the prototype tool Socos_1 was built in 2005 and 2006; 2) Socos_1 was evaluated in case studies (including the 2007 undergraduate course described in Chapter 9); and 3) the redesign and reimplementation project Socos_2 was started in 2007 to improve the

software architecture and to address issues raised in the evaluation. A large portion of the implementation work on both Socos_1 and Socos_2 has been carried out by student programmers employed in the Gaudí software factory. Socos_1 is no longer developed. Socos_2 at the present consists of a command line tool, which compiles programs into PVS theories containing verification conditions and sends them to the Yices SMT solver. It is implemented in Python [151], and relies on no other external components outside of PVS and Yices. We are currently developing a graphical front-end as an Eclipse plug-in, based on the editing framework GEF [70].

The remainder of the chapter describe the software architectures of Socos_1 and Socos_2 as well as the current status of implementation of Socos_2.

10.2 Socos_1

The Socos_1 prototype consists of a graphical invariant diagram editor and a verifier connected to an automatic theorem prover. The verifier generates conditions from the diagrams and tries to discharge as many as possible using Simplify. The unproven conditions are reported to the programmer via the editor interface. The user interface of the editor is similar in style to that of UML state chart editors. Figure 10.2 shows a screenshot of the Socos_1 environment in action.

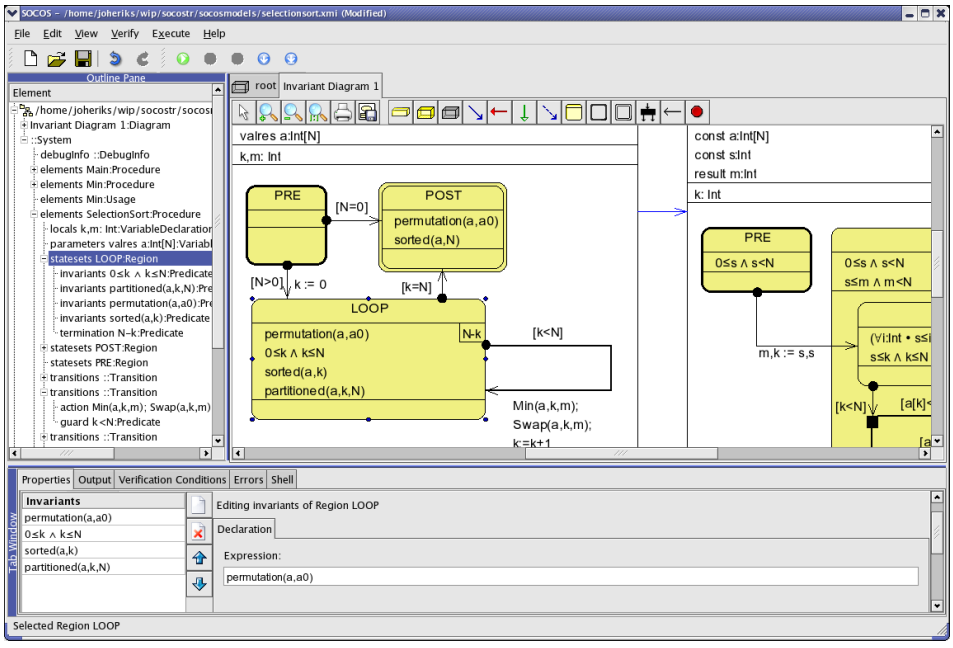


Figure 10.2: The Socos_1 environment

Socos_1 uses the Coral data repository to store invariant diagrams. Coral is a

metamodeling framework developed by Alanen and Porres [3, 5]. The diagram editor was initially implemented by Nevalainen [127] as an extension to Coral. The extension was based on DIML, a domain-specific language for defining concrete graphical syntaxes in Coral [4, 110].

Back and Myreen [28] built the first verifier for invariant-based programs as a feature layer of MathEdit, an extensible mathematics editor [18]. The verifier generates verification conditions from textual programs written in MathEdit and uses the validity checker Simplify [62] as a back-end for automatically simplifying the conditions. MathEdit was originally developed in the Gaudí factory as a case study of Back's *stepwise feature introduction* [12, 20]. Its software architecture is a layered class hierarchy, in which each layer introduces a specific feature; the program verifier contributes one layer to this hierarchy. Myreen later extended the layer with a compiler to Python. The Coral-based diagram editor and MathEdit were subsequently integrated into the Socos_1 environment [22] by the author in collaboration with Haikarainen [88].

The original intention of the Socos Project was to support refinement diagrams [13] rather than invariant diagrams. The focus of development was shifted towards IBP in 2005. Also, MathEdit was originally intended to be a tool for writing structured derivations rather than programs. While a working Socos_1 prototype at the time could be quickly built on top of these components, the fact that the underlying components were originally designed with other objectives made it difficult to maintain and develop a clean code base. Hence, the research group started development of the simplified and redesigned Socos_2 tool. Another motivation was to take advantage of the new generation of SMT solvers (Socos_1 is based on Simplify, which is no longer being developed).

10.3 Socos_2

The overall architecture of Socos_2 is shown in Figure 10.3. Components with dotted outlines are planned, but have not yet been implemented. The architecture is motivated by a desire to separate back-end and front-end components, and to add a degree of extensibility to the individual components of the system.

The verifier-compiler (IBP-VC) processes invariant-based programs into VCs and/or code. It consists of a command line program that reads invariant-based programs given in the syntax of Chapter 5. The discussion in this thesis has centered mainly around its first subcomponent, the verification condition generator (VCG), which generates the PVS theories as described in Chapter 7. We are currently evaluating the IBP-VC through case studies in early 2010, and are planning to make available a primary version of the Socos_2 system later that year. As we noted earlier, we have not yet implemented a compiler (COMP) for Socos_2. In the future work section of the next chapter we describe our current and planned work related to this component.

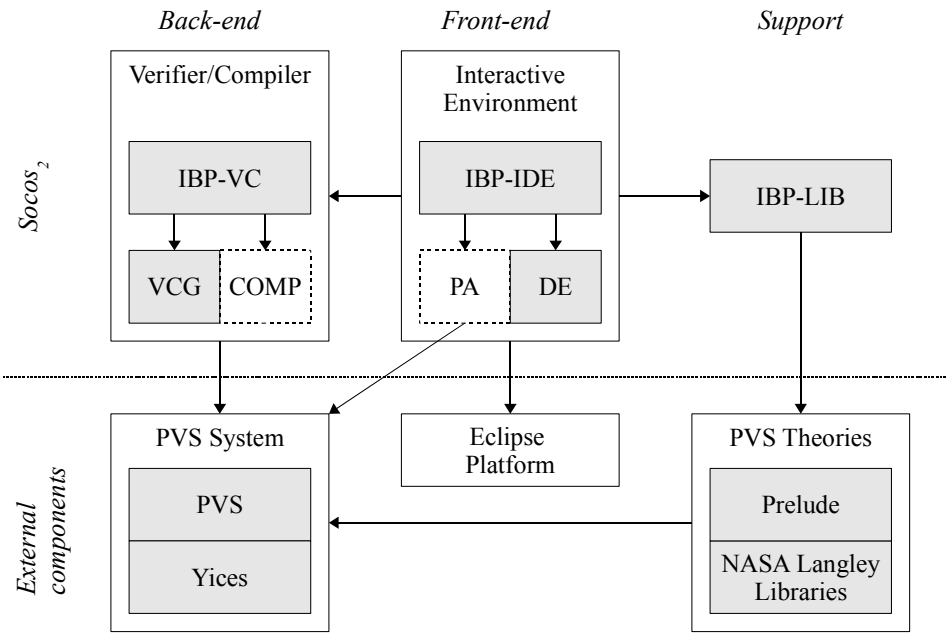


Figure 10.3: Software architecture of Socos_2 and its external components. Arrows indicate dependencies; dotted components are yet to be implemented

The front-end, IBP-IDE, consists of an Eclipse-based interactive development environment. Eclipse is an open source extensible development environment that offers a large collection of software engineering and problem solving related plug-ins, and has become one of the standard IDEs for software development. We are currently building a diagram editor (DE) in which programs can be created and checked directly as diagrams. The editor is implemented as an Eclipse plug-in; a snapshot of a prototype version of the editor in use is shown in Figure 10.4. The DE is currently being finished and integrated with the IBP-VC; when this work is completed, it will be possible to check the correctness of diagrams directly from Eclipse. The verification feedback will be presented within the editor in a context-sensitive manner. When a transition is selected, the tool will highlight the constraints of the target situation that were not proved automatically, and additionally show the actual VC in a popup window when a highlighted constraint is being hovered with the cursor.

A useful addition to the IBP-IDE whose implementation we have not yet considered is an Eclipse-based proof assistant (PA), such that PVS proofs could be carried out without leaving the Eclipse environment (the standard front-end of PVS is Emacs). Implementing such a front-end is a major undertaking, although there are currently projects in this direction underway. Proof General Eclipse [7] is an effort to build a generic prover interface for Eclipse-based on a protocol

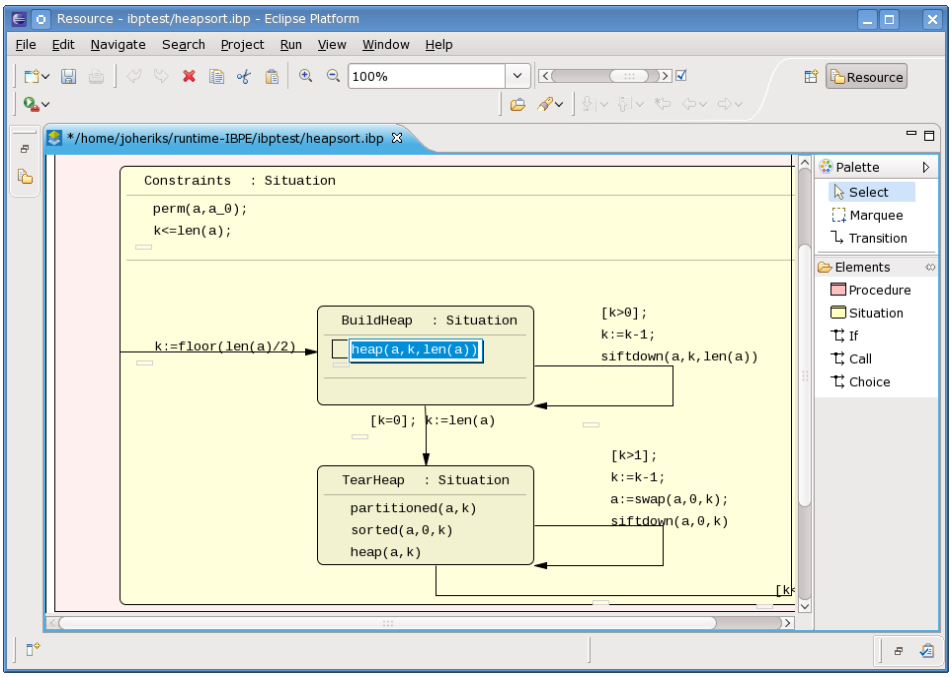


Figure 10.4: Socos diagram editor in Eclipse

called PGIP; there is support for Isabelle, but a PVS mode for Proof General Eclipse does not seem likely to appear in the near future. ProverEditor [51] is a generic lightweight theorem prover interface and is part of the Mobius Program Verification Environment [50]. The main target of ProverEditor has been to support Coq interaction in PVS, but an experimental PVS plugin is also available.

The IBP-LIB is intended to be a library of programs, background theories and strategies that can be reused. It currently consists of just a few basic elements (such as the `vectors` theory and the `endgame` strategy) but is continuously being extended as case studies are implemented in Socos. The NASA Langley PVS collection [122] contains several ready made packages in the domains of graphs, arrays, integer arithmetic, trigonometry, etc. This library of theories could be used directly from Socos to assist specification and verification of programs in these domains.

10.4 Summary

Socos has been developed in a project directed by Ralph-Johan Back at the Department of Information Technologies at Åbo Akademi University in collaboration with the TORES II Project, the IMPEd Resource Center, and the Gaudí Software Factory. Researchers functioned as customers for the project, whereas CS students

from Åbo Akademi University carried out most of the actual implementation work.

The development of Socos comprised three stages in the timespan 2005–2010. We first built the prototype Socos_1 and subsequently evaluated it in case studies. This prototype was based on MathEdit, Simplify and the Coral framework. We have since implemented a VC generator for PVS as described in Chapter 7, and we are currently building a graphical diagram editor based on Eclipse. Remaining components include an Eclipse front-end to PVS and a compiler to executable code.

Chapter 11

Conclusions and Future Work

This chapter concludes the thesis. We summarize the thesis, discuss the obtained results and outline some possible future research directions.

11.1 Summary

This thesis has described the product of a research project intended to build tool support for invariant-based programming (IBP). IBP was introduced to be a hands-on, concrete method for building programs that are correct by construction. The Socos project is a step towards making IBP practicable. The project has produced two prototype tools. We have in this thesis described the design goals and the technologies used to realize them, the features and limitations of the prototype tools built, and the contexts in which they have been evaluated.

Chapters 2–4 gave the research background of the thesis. We described a vein of program verification research leading up to IBP, followed by an overview of the technical basis for tool-supported program verification: verification semantics, language embedding and theorem provers. We also described the PVS specification language and theorem prover.

Chapter 5 introduced a basic invariant-based programming language based on the PVS language. A program is a verification context and a set of mutually recursive procedures specified with parameters and pre- and postconditions. The implementation of a procedure is an invariant diagram—a graph of nested situations and transitions with unrestricted control flow. Transitions are composed of nondeterministic choice, guarded choice, multi-exit procedure calls and single-exit statements.

Chapter 6 described a semantics based on weakest preconditions for verifying that an invariant diagram is consistent, live and terminating. Consistency means that no transition aborts, and that the target situation of every transition is established. Liveness means that a diagram does not terminate in a transition or a situation that is not final. Termination means that there are no infinite loops. The basic premise

is that consistency conditions are always checked, while liveness and termination checks are optional. To ensure liveness, a final situation should be reachable from every reachable situation, and no isolated assume statements should occur. To ensure termination, the graph must be reducible to a set of terminating components, such that every cycle in a terminating component strictly decreases a variant, and does not interfere with the variants of other terminating components.

Chapter 7 described a syntax directed translation of a Socos context into a PVS theory hierarchy, based on the verification semantics of invariant-based programs. Proving all lemmas and type conditions in the generated theories ensures that the context is consistent and type correct. Each lemma in a generated theory is structurally similar to the transition tree from which it was derived, and is associated with a proof script that splits the proof into separate goals for the individual constraints in the target situations. An extensible proof strategy is then applied to each leaf; if the strategy is unsuccessful, the unproven sequent is reported to the user.

Chapter 8 described a tool session in which a verified implementation of heapsort was built. The example used a background theory to make verification involving sortedness, permutations and heaps, manageable. Once a few deep properties in the background theory are proved, almost all the VCs were discharged automatically. We showed how the output when verification fails can be used to identify errors. We described alternative ways of handling unproven conditions.

Chapter 9 was an exposition of a teaching case study, in which IBP was taught to first and second year students in a pilot course. We described the course syllabus, the case study, and the results obtained. Our main finding was that an IBP tool can provide value to the teacher as well as to the learner, but that further refinement and documentation of the tool is necessary.

Chapter 10 described the development settings, the technologies behind Socos, and the current status of implementation of Socos_2.

11.2 Conclusions

Achievement of design goals. As stated in Chapter 1, Socos was intended to be designed for 1) practicability, 2) transparency, and 3) learnability. We will now give some observations in retrospect and try to judge how well these desiderata were realized.

Desideratum (1) is concerned with whether the tool is actually fit for purpose; i.e., does it support and add value to the IBP workflow, and is it indeed usable by, e.g., students who may not have a strong background in logic? Our experience is that the answer to the first question is a definitive yes. This is in one sense unsurprising, since automation obviously adds value to any deductive method that relies on proofs. We have estimated a typical discharge rate of 80–90 per cent of the generated VCs in our case studies, although we have not carried

out a detailed quantitative analysis.[1] Our case studies have also highlighted the importance of feedback when verification fails. The features available in the tool for decomposing proof effort—separation of constraints within situations, assert statements, **decreasing** declarations and assume statements—are useful not only because they allow a verification to be divided and conquered, but also because they allow a fine-grained verification report in which goals that are automatically discharged are never shown. This makes the feedback easier to read and understand, and enables proof effort to be more precisely targeted. Our response to the second segment of the question is more cautious, but positive. Based on experience from the case studies and teaching experiments, we do think that Socos could be used effectively by anyone who is familiar with programming and has a basic knowledge of predicate logic. However, since application has been restricted to small examples within academia, there is inconclusive data to assert that Socos is accessible to, say, the average professional programmer. It is not possible to test this hypothesis empirically at the present, since Socos does not currently support programming in the large and is thus not suited for realistic scale software projects. In Section 11.3 we discuss two potential research directions in this area.

Transparency (2) comes from the sharing of syntax and semantics with the reasoning framework, PVS. Since the verification semantics is a quite shallow layer, constraints and expressions in the source program are unobfuscated by the translation process and hence the VCs appear very natural. The absence of translation artifacts maintains the impression that the program *is* the proof, in particular when all proofs are fully automated. User defined constraints and expressions occur unchanged in the VCs, and new identifiers appear only in the context of procedure parameters (for initial-value constants, and to denote the updated values of mutable actual parameters after a call). This is comparable with the integration of formulas and programs within a single language in verification systems based on dynamic logic (DL) [92], such as KeY [40]. However, in contrast to DL our system does not allow involving programs in the description of states, but makes a strict distinction between the situation language (higher-order logic) and the transition language. Another aspect of transparency is that the generated PVS theories are human readable and can be scrutinized to check that the tool produces sound VCs for a given program. This is important since the VC generator itself has not been verified.

Our main conclusion from the teaching experiment is that desideratum (3) was achieved to the extent that it indeed is feasible to introduce Socos into an IBP course, even when the participants are formal methods novices. We attribute this to the lightweight and easy to learn workflow of IBP, and the fact that automatic theorem proving could be applied in a value-adding and non-intrusive manner

[1] Rate of discharged VCs is a somewhat dubious metric. Frequently the correctness proof for a program hinges on one or a few hard lemmas, while the rest of the lemmas are routine. A metric could be used to argue for the efficiency of a tool with respect to some standard base set of problems, though.

using the tool. The latter allowed us to ease in the tool support, and we think that the student perception of formal methods tools improved as a result. In summary, the results from the study were encouraging.

Use of underlying technology. Here we summarize our experience of the formal methods tools underlying Socos.

From the user perspective, PVS seemed well suited as a front-end language for IBP. In all our case studies, the panoply of constructs and powerful type system provided by PVS enabled succinct formulation of the specifications, invariants and transitions. The large existing theory libraries, such as the prelude [133] and the NASA PVS libraries [122], proved useful in several case studies. Abstract data types, although not used in the examples in this thesis, enables modeling of lists, trees and other data structures which add to the usefulness of the PVS language as an implementation language. The PVS language also has a feel to it that seems to make it quite natural and easy to learn for programmers, a quality not shared by all mathematical or specification oriented notations.

One difficulty we encountered when embedding a subset of PVS into our own language was parsing. We currently parse an LALR encoding of the PVS grammar subsets *Expr*, *TypeExpr*, and a subset of the constant declarations. The PVS grammar is highly context-dependent, and due the powerful overloading mechanisms an abstract syntax tree must be type checked even before identifier resolution can be attempted. As implementing type checking outside of PVS is not desirable, Socos currently does no semantic analysis of terms produced by the grammar subsets shared with PVS; such terms are passed through into the generated PVS theories. Hence, semantic errors are reported for the generated PVS theories, and not for the program. A better approach is to build the theories programmatically within PVS, using existing Lisp functions for parsing and typechecking, but we have not yet explored this option in detail yet. This approach is feasible since the PVS source code is fully open and can hence be inspected. However, an issue is that the internals are not well documented.

From our experience we also feel obligated to echo the points raised by Lüttgen et al. [111] in 2000. Since the internals of PVS are not documented it is difficult to reuse in strategies existing Lisp routines such as those for identifier generation or for accessing the database of generated TCCs. The correct use of such routines must be either inferred from the code of the routine, or reverse engineered from the built in strategies. More insight into the mechanisms of the prover, in the form of well-documented access to the proof context and the proof environment, would also be useful for developing more elaborate strategies. Finally, extensibility of the syntax within PVS itself would be useful when building support for other notations. Now tool builders must write their own parser (and perhaps typechecker), which as discussed above incurs additional complexity for the implementer.

Since Yices supports dependent types well and is already integrated into PVS, it

was a natural choice for the default validity checker. A practical advantage of using Yices is that it handles the common conditions such as conditions on array bounds well. For more elaborate conditions, it is important that the programmer has a good basic understanding of the capabilities of the SMT solver, since its effectiveness as well as efficiency depend greatly on how the constraints are formulated in the program. One issue occurs when the solver does not return an answer promptly; a delay of more than a few seconds can in practice be enough to interrupt the workflow. The SMT solver's time and memory requirements are not always related to the size of the problem or the statespace in a predictable way, and paradoxically it is possible that simplifying the formulas or reducing the statespace may result in worse performance. This is, however, an instance of a more general problem and not specific to Yices.

Criticism. In the next we address some of the criticism that the work presented in this thesis may evoke.

The most difficult problem in program verification is arguably the discovery of the invariants, and hence much effort has been devoted to methods for invariant detection. IBP circumvents the problem by requiring all invariants to be given, and argues that the discovery and formulation of invariants is to be the main focus of programming. We will try to argue for this view by considering two classes of invariants: invariants that are simple enough to be inferred from code, and invariants that are hard enough not to be. For the first class, it is true that the redundancy of having to state, e.g., loop counter bounds in situations (as constraints) as well as in transitions (as guards) can seem unnecessary. However, this redundancy can also be good, since it expresses two properties at different levels of abstraction: the state description level and the control flow level, and allows inconsistencies between the two to be identified in a very local fashion (if there is an error in the loop condition, an inferred invariant will also be wrong—in the sense of not matching the programmer's *intention*). Also, it is likely to be the case the conditions associated with such invariants and transitions can be discharged automatically (it is generally easier to check a proof than to build a proof). For the second class, user specified invariants are required in both the code-first and invariants-first approaches. Here the argument is that coming up with the invariant is harder than writing the code—even in code-first development, some basic notion of the properties the program is to maintain during its execution must exist before any code can be written—so adapting the code makes the program easier to verify.

The reasoning in the previous paragraph assumes that programmer intention is better expressed and understood by way of declarative rather than operational reasoning; most programmers who use assertions in their code are likely to agree on this. Furthermore it assumes that programs are built from the ground up. If the task is to redesign an existing program into an invariant-based one, a tool that aids

in finding the invariants may indeed be valuable. However, verification *a posteriori* is outside the scope of this thesis.

We acknowledge that the set of programs we have considered exhibits only a subset of the issues that a programmer faces, and that our case studies may not be representative of industry scale software development. However, the objective of this research has not been to build a full-fledged program verification tool, but rather to investigate the feasibility of tool-supported IBP, and to evaluate the method in case studies. Since IBP is a new method, we consider this is an important stepping stone towards scaling up the technique to realistic programs.

The verification semantics of Socos is not yet formally connected to an operational (trace) semantics. An end-to-end verification methodology should establish soundness (and perhaps completeness) along the whole semantic spectrum from small step execution up to the actual proof rules; this remains to be done. While the mathematical underpinnings is an important research theme, it is also one that can be considered independent of the research into the utility of IBP. Back introduced IBP not as a new program calculus, but as a hands-on method for building correct programs. Socos should be considered an extension of this method with state of the art proof tools. For a technique that cannot be taught, learned or properly supported by tools is unlikely to have impact, even if theoretically well-founded.

We acknowledge the small scale of the teaching case study and that we have not done a comparative evaluation of IBP against other approaches means that there is not enough data to draw any hard conclusions regarding the role of an IBP tool in education. We consider our teaching experiments to the present to have been on the proof-of-concept stage, where the goals have been to evaluate the advantages and disadvantages of both IBP and Socos. The case study is not a claim that IBP and/or Socos are superior to existing approaches. The results have convinced us that this is a direction worth pursuing; further development to streamline integration of the tool into teaching and evaluations on a larger scale will be necessary to strengthen the credibility of the whole approach.

11.3 Future work

There is scope for much future work on this topic. We list here the new features of the tool that we are working on, and some future research directions.

Tool enhancement. As the IBP research has been experimental and the tools we have implemented are highly prototypical, we have certainly not yet harnessed the full potential of automation. Improving automation is an ongoing task. More comprehensive control of the application of auto-rewrites, background theory lemmas and user-defined strategies, e.g. by allowing diagrams to be annotated with prover hints or even entire proof scripts, could potentially have a significant impact on the degree of automation in practice. Another interesting prospect is

background verification, whereby VCs are generated and discharged on the fly as the diagram is being drawn. An analogous feature exists in the Spec# system [33], where assertion violations are highlighted inside the code editor as the code is being written. We are considering building a set of benchmark programs, such that endgame strategies can be tested and compared quantitatively. We are also looking forward to being able to use the new Yices2 solver, which was ranked high in the 2009 SMT-COMP [35]; Yices2 reads input in the SMT-LIB format, and hence cannot at the present be used directly from PVS.

There is currently no standard run-time environment for IBP. Socos_1 provided a basic lexical translation into Python. For Socos_2, we have done basic experimentation with the PVS ground evaluator [145], which converts PVS expressions into executable Common Lisp code. An *evaluable expression* is an expressions that is not of an uninterpreted type, and that does not contain free variables, unbounded quantification, or higher-order equality. It is straightforward to define a virtual machine for Socos programs that uses the ground evaluator to interpret expressions, or alternatively, a compiler that generates stand alone Lisp programs. Evaluation of invariants and specifications may be useful for traditional debugging and testing; in this mode the interpreter/compiler should evaluate the expressions that are in the evaluable subset. We are also planning a compilation of invariant-based programs in which the generated executable is automatically proved consistent with the verification semantics. The compilation will use a *translation validation approach*, in which instances of consistency proofs are generated together with the VCs and the executable based on a connection between the verification and operational semantics defined by Back and Preoteasa [29].

There is a number of immediate enhancements that could be implemented, including nested lexical scopes within procedures, global variables, use of any well founded set in termination proofs, and generalized termination conditions for mutually recursive procedures (discussed in Chapter 6).

Scaling up IBP and Socos. Given that the basic approach seems feasible, the next challenge is to identify realistic applications for IBP. This involves finding niches in which the method can provide direct value for its users, such that a community can be built and tools developed in interaction with an active user group. At the present, teaching may be such an application area. An interesting prospect in the long run is adapting IBP to development of software of realistic scales. Below we discuss two possible approaches to the latter.

One approach involves *extending* IBP/Socos to handle programming in the large. Firstly, an invariant-based methodology for reasoning about high-level constructs such as classes, modules, and libraries should be developed. This is in itself a major research undertaking. Support for the extended methodology should then be added to Socos. Two open research problems are 1) finding a practical method for structuring the statespace of a large program into situations, and 2)

extending the transition checking technique to higher-level constructs to ensure, e.g., that the methods of a class maintain the object invariant. Both (1) and (2) require a notion of refinement to be supported.

A second approach involves *adapting* IBP/Socos to existing programming environments, and making use of IBP mainly as a methodological tool. One possible direction is integrating IBP orthogonally into an existing "host method", such as the B-method [1], which already handles development in the large and which has robust and readily available support tools. In this way the internals of an individual component, such as a B abstract machine, could be developed in the IBP style, while the refinement and interfacing of components would be handled by the support mechanisms in the host method. This would allow IBP to be applied directly in large scale projects, since it could coexist with the other components already developed using the host method.

Teaching. The application of Socos in teaching also continues. We are currently (as of 2010) giving two courses in which the tool will be used: an introductory IBP course, as well as a special course on tool-supported verification. The first course follows the syllabus described in Chapter 9. The second course focuses on the development of correct programs using PVS and Socos, and is targeted at advanced students. We expect it to give us additional insight in the feasibility of a tool such as Socos for building and verifying non-trivial programs.

Appendix A

Listings

A.1 Symbols

Table A.1 below shows the ASCII versions of the typeset symbols used in the examples in this thesis, and the PVS theory in which each is defined.

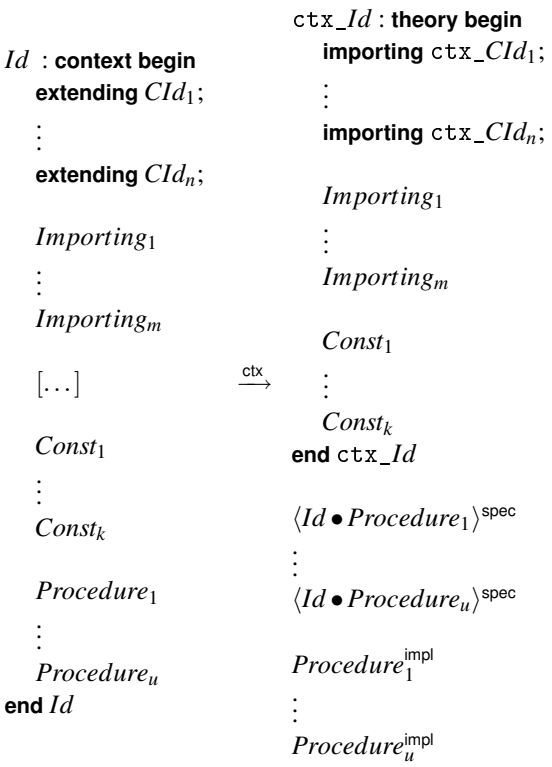
Symbol	*ASCII*	*Defined in*	*Description*
$\rightarrow$	`->`	*built-in*	function type constructor
$\exists$	`EXISTS`	–"–	existential quantifier
$\forall$	`FORALL`	–"–	universal quantifier
λ	`LAMBDA`	–"–	lambda abstraction
$\neq$	`/=`	`notequal`	not equal to
$\Rightarrow$	`=>`	`booleans`	implication
$\Leftrightarrow$	`<=>`	–"–	equivalence
$\wedge$	`AND`	–"–	binary conjunction
$\vee$	`OR`	–"–	binary disjunction
$\times$	`*`	`operator_defs`	star operator
$\leq$	`<=`	`orders`	less than or equal to
$\geq$	`>=`	`reals`	greater than or equal to
$a[i]$	`access(`a,i`)`	`vector`	array access
$a[i \leftarrow x]$	`update(`a,i,x`)`	–"–	array update

Table A.1: Typeset symbols and corresponding ASCII sequences

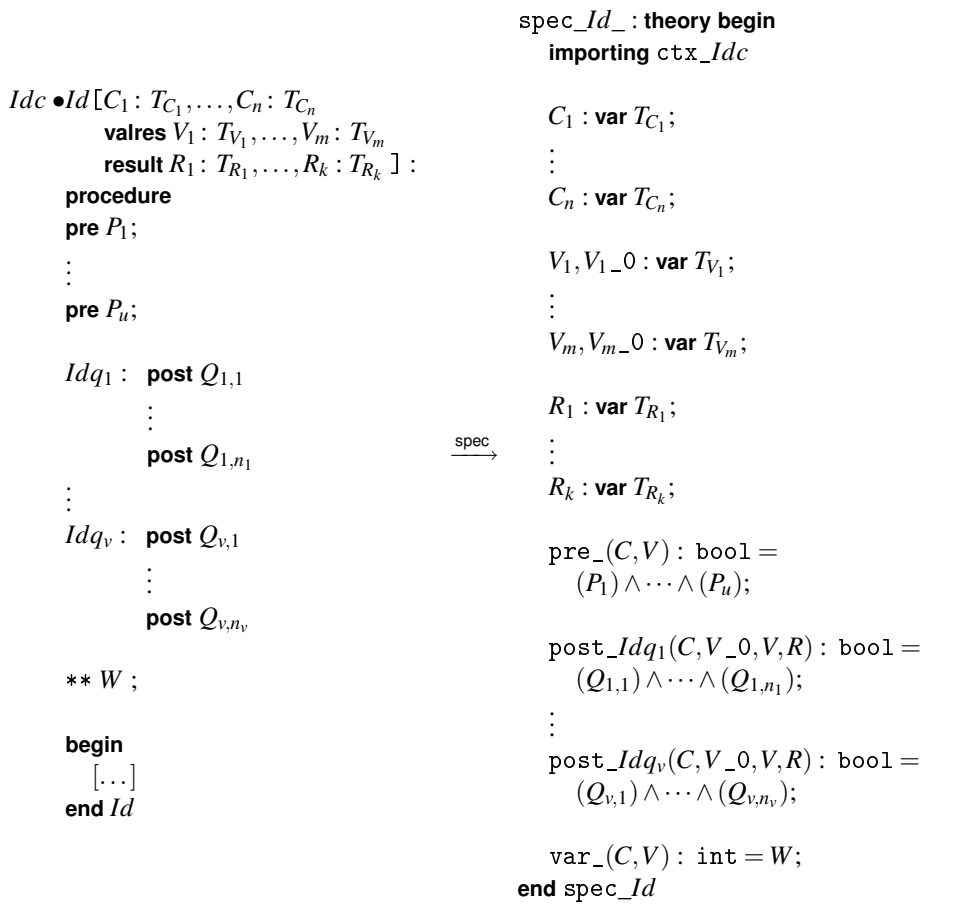
The PVS theory `vector` is listed on page 146.

A.2 Translation of Socos programs into PVS theories

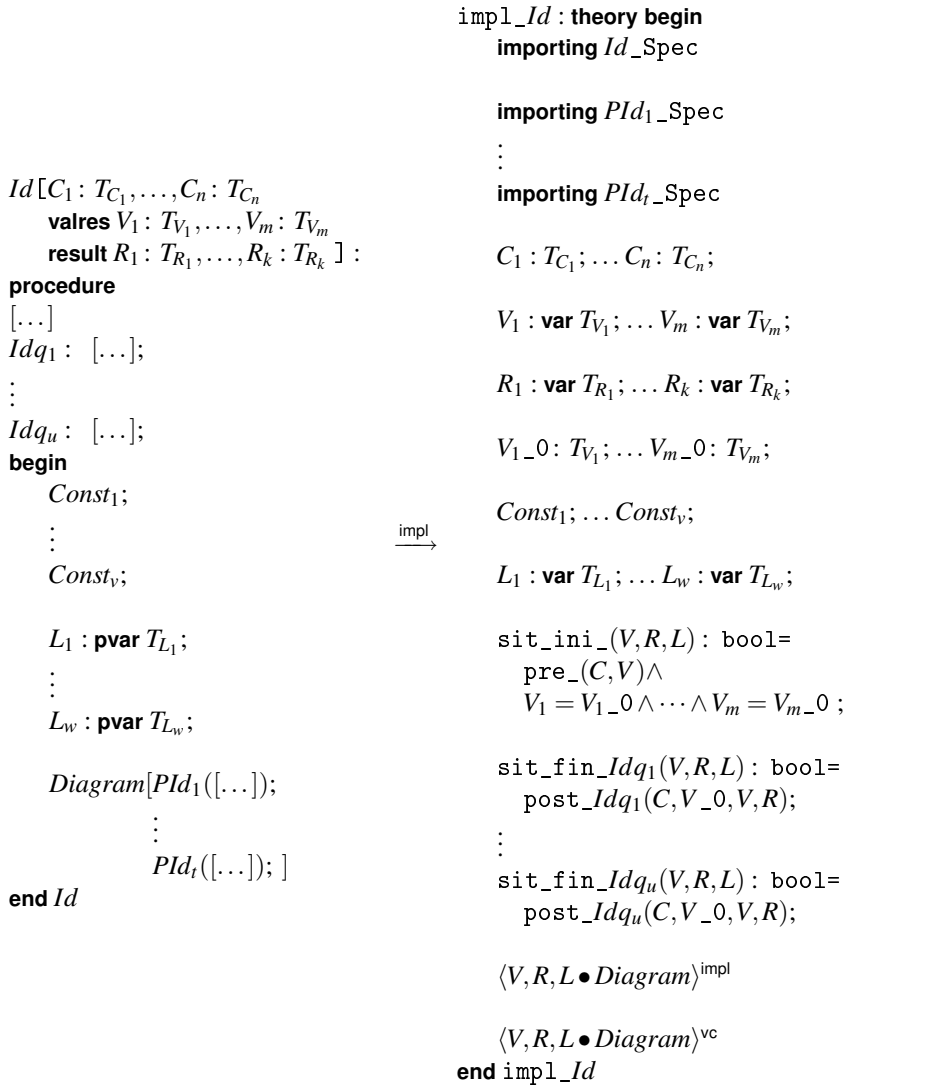
We give below syntactic rewrite rules for a transformation ctx, which translates a Socos context into a collection of PVS theories. The translation was discussed in Chapter 7.



N.B. The transformations spec and impl are given in the remainder of this section.



N.B. In the right hand side above, V_0 is an abbreviation for the sequence $V_1\_0, \ldots, V_m\_0$.



N.B. Above $PId_1, \ldots, PId_t$ is the set of called procedures. On the right hand side, V_0 is an abbreviation for the sequence V_1_0, $\ldots, V_m$_0. .

$\sigma\bullet$ *Situation*$_1$
$\vdots$
Situation$_n$
$[\ldots]$

$\xrightarrow{\text{impl}}$

$\langle\sigma\bullet$ *Situation*$_1\rangle^{\text{impl}}$
$\vdots$
$\langle\sigma\bullet$ *Situation*$_n\rangle^{\text{impl}}$

$\sigma\bullet$ *Id* : **situation begin**
* I_1;
$\vdots$
* I_n;
$[\ldots]$
Situation$_1$
$\vdots$
Situation$_m$
$[\ldots]$
end *Id*

$\xrightarrow{\text{impl}}$

`sit_`*Id*(σ) : `bool` =
$(I_1)\wedge\cdots\wedge(I_n)$;

$\langle\sigma,$ *Id* $\bullet$ *Situation*$_1\rangle^{\text{impl}}$
$\vdots$
$\langle\sigma,$ *Id* $\bullet$ *Situation*$_m\rangle^{\text{impl}}$

$\sigma,$ *Idp*$\bullet$ *Id* : **situation begin**
* I_1;
$\vdots$
* I_n;
$[\ldots]$
Situation$_1$
$\vdots$
Situation$_m$
$[\ldots]$
end *Id*

$\xrightarrow{\text{impl}}$

`sit_`*Id*(σ) : `bool` =
`sit_`*Idp*$(\sigma)\wedge$
$(I_1)\wedge\cdots\wedge(I_n)$;

$\langle\sigma,$ *Id* $\bullet$ *Situation*$_1\rangle^{\text{impl}}$
$\vdots$
$\langle\sigma,$ *Id* $\bullet$ *Situation*$_m\rangle^{\text{impl}}$

$\sigma\bullet$ *Situation*$_1$
$\vdots$
Situation$_n$
Trs

$\xrightarrow{\text{vc}}$

$\langle\sigma\bullet$ *Situation*$_1\rangle^{\text{vc}}$
$\vdots$
$\langle\sigma\bullet$ *Situation*$_n\rangle^{\text{vc}}$

$\langle$`ini_`$,\sigma\bullet$ *Trs*$\rangle^{\text{vc}}$;

$\sigma\bullet$ *Id* : **situation begin**
$[\ldots]$
Situation$_1$
$\vdots$
Situation$_n$
Trs
end *Id*

$\xrightarrow{\text{vc}}$

$\langle\sigma\bullet$ *Situation*$_1\rangle^{\text{vc}}$
$\vdots$
$\langle\sigma\bullet$ *Situation*$_n\rangle^{\text{vc}}$

$\langle$*Id*$,\sigma\bullet$ *Trs*$\rangle^{\text{vc}}$;

N.B. We described the transformation vc in Section 7.3.

A.3 Background theories

```
vector[T:type]: theory
begin
    vector:type+ = [# len:nat, elem:[below(len)->T] #]

    index(a:vector):type = below[len(a)]

    access(v:vector,i:below(len(v))):T = v`elem(i)

    acc(v:vector)(i:below(len(v))):macro T = access(v,i)
    conversion+ acc

    update(v:vector,j:index(v),x:T):{w:vector|len(w)=len(v)}=
      (# len:=len(v),
         elem:=elem(v) with [ j:=x ] #)

    update_prop_1: lemma
      forall (v:vector,i:index(v),x:T):
        access(update(v,i,x),i) = x

    update_prop_2: lemma
      forall (v:vector,i:index(v),j:index(v),x:T):
        i=j or access(update(v,i,x),j) = access(v,j)

    auto_rewrite- access,update

    eql(a:vector,b:vector,l,r:nat): bool =
      forall (i:nat): l<=i and i<r and i<len(a) and i<len(b)
                      => access(a,i)=access(b,i)

end vector
```

```
sorting: theory
begin
    importing vector[int]

    a,b,c: var vector

    sorted(a,(k:upto(len(a)))):bool =
      forall (i,j:nat): k<=i and i<j and j<len(a) =>
                        access(a,i)<=access(a,j)

    sorted(a):bool = sorted(a,len(a))
```

```
    partitioned(a,(k:upto(len(a)))):bool =
      forall (i,j:nat): i<k and k<=j and j<len(a) =>
                        access(a,i)<=access(a,j)

    perm(a,b) : bool =
      exists (f:(bijective?[index(b),index(a)])):
        forall (i:index(b)): access(a,f(i)) = access(b,i)

    perm_len: lemma perm(a,b) => len(a)=len(b)
    perm_ref: lemma perm(a,a)
    perm_sym: lemma perm(a,b) => perm(b,a)
    perm_trs: lemma perm(a,b) and perm(b,c) => perm(a,c)

    swap(a,(i,j:index(a))):{b|len(b)=len(a)} =
      update(update(a,i,access(a,j)),j,access(a,i))

    swap_acc: lemma forall (a,(i,j,k:index(a))):
      access(swap(a,i,j),k) = access(a,if k=i then j
                                       elsif k=j then i
                                       else k endif)

    swap_perm: lemma forall (a,(i,j:index(a))): perm(swap(a,i,j),a)

    auto_rewrite- perm,swap

    l(i:nat):nat = 2*i+1
    r(i:nat):nat = 2*i+2

    heap(a,(m,n:nat)): bool =
      m<=n and n<=len(a) and
      forall (i:nat):
        m<=i =>
          (l(i)<n => access(a,i)>=access(a,l(i))) and
          (r(i)<n => access(a,i)>=access(a,r(i)))

    heap_max: lemma
      forall (a,(k:nat)):
        heap(a,0,k) => forall (i:nat): 0<=i and i<k =>
                                       access(a,i)<=access(a,0)

    perm_partitioned: lemma
      forall (a,b,(k:upto(len(a)))):
        perm(a,b) and partitioned(a,k) and eql(a,b,k,len(a))
        => partitioned(b,k)

end sorting
```

A.4 The `heapsort` context

```
heapsort: context
begin
   importing sorting;
   using "(endgame :lemmas (perm_len perm_ref perm_sym
                            perm_trs swap_acc swap_perm))";

   heapsort[ valres a:vector[int] ]: procedure
   post sorted(a);
   post perm(a,a_0);
   begin
      k: pvar nat;

      Constraints: situation
      begin
         * perm(a,a_0);
         * k<=len(a);

         BuildHeap: situation
         begin
            * heap(a,k,len(a));
            ** k;
            if [k>0]; k:=k-1;
                      siftdown(k,len(a),a);
                      decreasing goto BuildHeap
               [k=0]; k:=len(a);
                      goto TearHeap
            endif
         end BuildHeap

         TearHeap: situation
         begin
            * partitioned(a,k);
            * sorted(a,k);
            * heap(a,0,k);
            ** k;
            if [k>1]; k:=k-1;
                      a:=swap(a,0,k);
                      siftdown(0,k,a);
                      decreasing goto TearHeap
               [k<=1]; exit
            endif
         end TearHeap
      end Constraints

      k:=floor(len(a)/2); goto BuildHeap
   end heapsort
```

```
   siftdown[ m,n:nat, valres a:vector[int] ]: procedure
   pre m<=n and n<=len(a);
   pre heap(a,m+1,n);
   post heap(a,m,n);
   post perm(a,a_0);
   post eql(a,a_0,0,m);
   post eql(a,a_0,n,len(a));
   begin
      k: pvar nat;

      Sift: situation
      begin
         * perm(a,a_0);
         * m<=k and k<=n and n<=len(a);
         * eql(a,a_0,0,m);
         * eql(a,a_0,n,len(a));
         * forall (i:nat): m<=i =>
            (i/=k =>
              (l(i)<n => a(l(i))<=a(i)) and
              (r(i)<n => a(r(i))<=a(i))) and
            ((l(i)=k or r(i)=k) =>
              (l(k)<n => a(l(k))<=a(i)) and
              (r(k)<n => a(r(k))<=a(i))) ;
         ** n-k;

         if [n<=r(k) or (a(l(k))<=a(k) and a(r(k))<=a(k))];
               if [r(k)=n];
                     if [a(l(k))<=a(k)]; exit
                        [a(k)<a(l(k))]; a:=swap(a,k,l(k)); exit
                     endif
                  [r(k)/=n]; exit
               endif
            [r(k)<n and (a(k)<a(l(k)) or a(k)<a(r(k)))];
               if [a(r(k))<=a(l(k))];
                     a:=swap(a,k,l(k)); k:=l(k);
                     decreasing goto Sift
                  [a(l(k))<=a(r(k))];
                     a:=swap(a,k,r(k)); k:=r(k);
                     decreasing goto Sift
               endif
         endif
      end Sift

      k:=m; goto Sift
   end siftdown

end heapsort
```

Bibliography

[1] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[2] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In *Proc. of the 8th International Conference on Formal Engineering Methods (ICFEM 2006)*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer, 2006.

[3] M. Alanen. *A Metamodeling Framework for Software Engineering*. PhD thesis, Turku Centre of Computer Science, Finland, 2007.

[4] M. Alanen, T. Lundkvist, and I. Porres. Creating and reconciling diagrams after executing model transformations. *Science of Computer Programming*, 68(3):128–151, 2007.

[5] M. Alanen and I. Porres. The Coral Modelling Framework. In K. Koskimies, L. Kuzniarz, J. Lilius, and I. Porres, editors, *Proc. of the 2nd Nordic Workshop on the Unified Modeling Language NWUML'2004*, number 35 in General Publications. Turku Centre for Computer Science, July 2004.

[6] V. L. Almstrum, C. N. Dean, D. Goelman, T. B. Hilburn, and J. Smith. Support for teaching formal methods. *SIGCSE Bull.*, 33(2):71–88, 2001.

[7] D. Aspinall, D. Winterstein, C. Lüth, and A. Fayyaz. Proof General in Eclipse: System and Architecture Overview. In *Eclipse '06: Proc. of the 2006 OOPSLA workshop on Eclipse technology eXchange*, pages 45–49, New York, NY, USA, 2006. ACM.

[8] R.-J. Back. *On the correctness of refinement in program development*. Ph.d. thesis, Department of Computer Science, University of Helsinki, Finland, 1978.

[9] R.-J. Back. Program construction by situation analysis. Research Report 6, Computing Centre, University of Helsinki, Helsinki, Finland, 1978.

[10] R.-J. Back. Exception handling with multi-exit statements. In H. J. Hoffmann, editor, *6th Fachtagung Programmiersprachen und Programmentwicklungen*, volume 25 of *Informatik Fachberichte*, pages 71–82, Darmstadt, 1980. Springer-Verlag.

[11] R.-J. Back. Invariant based programs and their correctness. In W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 223–242. MacMillan Publishing Company, 1983.

[12] R.-J. Back. Software construction by stepwise feature introduction. In *ZB '02: Proc. of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 162–183. Springer, 2002.

[13] R.-J. Back. Incremental software construction with refinement diagrams. Technical Report 660, Turku Centre for Computer Science, Turku, Finland, Jan. 2005.

[14] R.-J. Back. Invariant based programming revisited. Technical Report 661, Turku Centre for Computer Science, Turku, Finland, 2005.

[15] R.-J. Back. Invariant based programming. In S. Donatelli and P. S. Thiagarajan, editors, *ICATPN*, volume 4024 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2006.

[16] R.-J. Back. Invariant based programming: Basic approach and teaching experiences. *Formal Aspects of Computing*, 21(3):227–244, 2009.

[17] R.-J. Back. Structured derivations: a unified proof style for teaching mathematics. *Formal Aspects of Computing*, 2009. Online at `http://www.springerlink.com/index/a75tmu1110kku422.pdf`.

[18] R.-J. Back, V. Bos, and J. Eriksson. MathEdit: Tool Support for Structured Derivations. Technical Report 854, Turku Centre for Computer Science, Turku, Finland, Dec. 2007.

[19] R.-J. Back, J. Eriksson, and L. Mannila. Teaching the construction of correct programs using invariant based programming. In *Proc. of 3rd South-East European Workshop on Formal Methods (SEEFM07)*. South-East European Research Centre (SEERC), 2007.

[20] R.-J. Back, J. Eriksson, and L. Milovanov. Using stepwise feature introduction in practice: an experience report. In *Proc. of the 2nd International Workshop on Rapid Integration of Software Engineering Techniques (RISE'2005)*, volume 3943 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2005.

[21] R.-J. Back, J. Eriksson, and M. Myreen. Testing and verifying invariant based programs in the SOCOS environment. Technical Report 797, Turku Centre for Computer Science, Turku, Finland, Dec. 2006.

[22] R.-J. Back, J. Eriksson, and M. Myreen. Testing and verifying invariant based programs in the SOCOS environment. In *Proc. of the International Conference on Tests And Proofs (TAP)*, volume 4454 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 2007.

[23] R.-J. Back, J. Grundy, and J. von Wright. Structured calculational proof. *Formal Aspects of Computing*, 9(5–6):469–483, 1997.

[24] R.-J. Back and M. Karttunen. A predicate transformer semantics for statements with multiple exits. University of Helsinki, unpublished manuscript, 1983.

[25] R.-J. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3(2):73–87, 1989.

[26] R.-J. Back, L. Milovanov, and I. Porres. Software development and experimentation in an academic environment: The Gaudi experience. In *Proc. of the 6th International Conference on Product Focused Software Process Improvement - PROFES 2005, Oulu, Finland*, June 2005.

[27] R.-J. Back, L. Milovanov, I. Porres, and V. Preoteasa. XP as a framework for practical software engineering experiments. In *Proc. of the Third International Conference on eXtreme Programming and Agile Processes in Software Engineering - XP2002*, May 2002.

[28] R.-J. Back and M. Myreen. Tool support for invariant based programming. In *Proc. of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 711–718. IEEE Computer Society, 2005.

[29] R.-J. Back and V. Preoteasa. Semantics and proof rules of invariant based programs. Technical Report 903, Turku Centre for Computer Science, Turku, Finland, July 2008.

[30] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[31] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, 2006.

[32] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium*

(FMCO'2005), volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[33] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[34] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability modulo theories competition. In K. Etessami and S. K. Rajamani, editors, *Proc. of the 17th International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 20–23. Springer-Verlag, July 2005. Edinburgh, Scotland.

[35] C. Barrett, M. Deters, A. Oliveras, and A. Stump. SMT-COMP'09. `http://www.smtcomp.org/2009`, visited Jun 1, 2010.

[36] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0 (draft). Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `http://www.smt-lib.org`, visited Jun 1, 2010.

[37] C. Barrett and C. Tinelli. CVC3. In *Proc. of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, July 2007.

[38] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK: a tool for validation of security and behaviour of Java applications. In *Proc. of the 5th International Symposium on Formal Methods for Components and Objects (FMCO'06)*, volume 4709 of *Lecture Notes in Computer Science*, pages 152–174. Springer, 2007.

[39] F. L. Bauer, M. Broy, R. Gnatz, W. Hesse, B. Krieg-Brückner, H. Partsch, P. Pepper, and H. Wössner. Towards a wide spectrum language to support program specification and program development. *SIGPLAN Not.*, 13(12):15–24, 1978.

[40] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[41] P. Behm, P. Benoit, A. Faivre, and J. Meynadier. Météor: A successful application of B in a large project. In *Proc. of the Wold Congress on Formal Methods in the Development of Computing Systems - Volume I*, pages 369–387. Springer, 1999.

[42] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[43] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. The Barcelogic SMT solver. In *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 294–298. Springer, 2008.

[44] R. Boulton, A. Gordon, M. Gordon, J. Herbert, and J. van Tassel. Experience with embedding hardware description languages in HOL. In *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156. North-Holland, 1992.

[45] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.

[46] R. S. Boyer, M. Kaufmann, and J. S. Moore. The Boyer-Moore theorem prover and its interactive enhancement. *Computers & Mathematics with Applications*, 29(2):27–62, 1995.

[47] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer, 2006.

[48] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.

[49] Centre of Excellence for Formal Methods in Programming at Åbo Akademi (CREST). IMPEd Resource Centre. Project website `http://crest.cs.abo.fi/imped`, visited Jun 1, 2010.

[50] J. Charles, D. Cochran, W. Dietl, F. Fairmichael, R. Grigore, M. Huisman, J. Kiniry, E. Poll, and A. Schubert. Deliverable 3.10: Final report on program verification environment and annotation generation, 2010. Available at `http://mobius.inria.fr`.

[51] J. Charles and J. Kiniry. A lightweight theorem prover interface for Eclipse. In *8th International Workshop User Interfaces for Theorem Proving (UITP'08)*, Aug 2008.

[52] G. Cleland and D. MacKenzie. Inhibiting factors, market structure and the industrial uptake of formal methods. In *WIFT '95: Proc. of the 1st Workshop on Industrial-Strength Formal Specification Techniques*, pages 46–60, Washington, DC, USA, 1995. IEEE Computer Society.

[53] D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2005.

[54] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[55] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[56] M. A. Cusumano. Who is liable for bugs and security flaws in software? *Communications of the ACM*, 47(3):25–27, 2004.

[57] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[58] 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):271–280, 1979.

[59] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, Apr. 2008.

[60] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, May 2005.

[61] D. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report SRC-RR-159, Compaq SRC, Palo Alto, CA, Dec. 1998.

[62] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[63] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, 1968.

[64] E. W. Dijkstra. *Notes on structured programming*, pages 1–82. Structured programming. Academic Press Ltd., 1972.

[65] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

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

[67] E. W. Dijkstra. On the cruelty of really teaching computer science. *Communications of the ACM*, 32(12):1398–1404, Dec. 1989.

[68] E. W. Dijkstra and C. S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag New York, Inc., 1990.

[69] B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, Aug. 2006. Available at `http://yices.csl.sri.com/tool-paper.pdf`.

[70] Eclipse Foundation. Graphical Editing Framework (GEF). Project website `http://www.eclipse.org/emf/`, visited Jun 1, 2010.

[71] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[72] J. Erikson. Nybörjarsvårigheter med invariantbaserad programmering. Master's thesis, Åbo Akademi University, Department of Information Technologies, 2008.

[73] L. Erkök and J. Matthews. Using Yices as an automated solver in Isabelle/HOL. In *Automated Formal Methods'08*, pages 3–13, Princeton, New Jersey, USA, July 2008. ACM Press.

[74] I. Feinerer and G. Salzer. Automated tools for teaching formal software verification. In *Teaching Formal Methods: Practice and Experience*, BCS Electronic Workshops in Computing (eWiC). BCS-FACS, Dec. 2006.

[75] J. H. Fetzer. Program verification: the very idea. *Communications of the ACM*, 31(9):1048–1063, 1988.

[76] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research report 1366, LRI, Université Paris Sud, Mar. 2003.

[77] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

[78] S. Fincher and M. Petre, editors. *Computer Science Education Research*. RoutledgeFalmer, 2004.

[79] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI'02)*, pages 234–245. ACM Press, 2002.

[80] R. W. Floyd. Assigning meanings to programs. *Proc. of Symposium of Applied Mathematics*, 19:19–32, 1967.

[81] H. H. Goldstine and J. von Neumann. Planning and coding of problems for an electronic computing instrument, 1947. Report on the Mathematical and Logical Aspects of an Electronic Computing Instrument, Part II, vol. I. Also in A.H. Taub, editor. John von Neumann, Collected Works, Volume V, pp. 80-151.

[82] D. I. Good. *Toward a man-machine system for proving program correctness*. PhD thesis, The University of Wisconsin - Madison, 1970.

[83] M. J. C. Gordon. Mechanizing programming logics in higher order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439. Springer, 1988.

[84] D. Gries. *The Science of Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1987.

[85] D. Gries. Teaching calculation and discrimination: a more effective curriculum. *Communications of the ACM*, 34(3):44–55, 1991.

[86] D. Gries. The mathematics of programming and why we should teach it. *Journal of Computing Sciences in Colleges*, 19(5):2–2, 2004.

[87] D. Gries and F. B. Schneider. *A logical approach to discrete math*. Springer-Verlag New York, Inc., 1993.

[88] J. Haikarainen. SOCOS support for data modules. Master's thesis, Åbo Akademi University, Department of Computer Science, 2006.

[89] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[90] A. Hall. What is the formal methods debate about? *IEEE Computer*, 29(4):22–23, 1996.

[91] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[92] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[93] P. B. Henderson. Mathematical reasoning in software engineering education. *Communications of the ACM*, 46(9):45–50, 2003.

[94] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[95] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[96] C. A. R. Hoare. Towards the verifying compiler. In O. Owe, S. Krogdahl, and T. Lyche, editors, *Essays in Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science*, pages 124–136. Springer, 2004.

[97] C. A. R. Hoare and J. Misra. Verified software: Theories, tools, experiments vision of a grand challenge project. In B. Meyer and J. Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2005.

[98] C. M. Holloway and R. W. Butler. Impediments to industrial use of formal methods. *IEEE Computer*, 29(4):25–26, 1996.

[99] P. V. Homeier. *Trustworthy tools for trustworthy programs: a mechanically verified verification condition generator for the total correctness of procedures*. PhD thesis, University of California at Los Angeles, 1995.

[100] IEEE Standards Board. IEEE Standard for Software Unit Testing: An American National Standard, ANSI/IEEE Std 1008-1987. In *IEEE Standards: Software Engineering, Volume Two: Process Standards*. American National Standards Institute, 1987.

[101] S. Igarashi, R. L. London, and D. C. Luckham. Automatic program verification I: a logical basis and its implementation. Technical report, Stanford University, 1973.

[102] C. B. Jones. The early search for tractable ways of reasoning about programs. *IEEE Annals of the History of Computing*, 25(2):26–49, 2003.

[103] C. B. Jones, P. O'Hearn, and J. Woodcock. Verified software: A grand challenge. *Computer*, 39(4):93–95, 2006.

[104] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Springer, 2000.

[105] C. Kelemen, A. Tucker, P. Henderson, O. Astrachan, and K. Bruce. Has our curriculum become math-phobic? (an American perspective). *SIGCSE Bull.*, 32(3):132–135, 2000.

[106] J. C. King. *A program verifier*. PhD thesis, Carnegie Mellon University, 1969.

[107] S. King and C. Morgan. Exits in the refinement calculus. *Formal Aspects of Computing*, 7(1):54–76, 1995.

[108] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Boston, 1999.

[109] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal Verifier user manual. Technical report, Stanford University, 1979.

[110] T. Lundkvist and I. Porres. Coordination of model transformation engines and visual editors. In J. Peltonen, editor, *Proc. of SPLST'09 and NW-MODE'09*, number 5 in Department of Software Systems reports, pages 269–283. Tampere University of Technology, 2009.

[111] G. Lüttgen, C. Munoz, R. Butler, B. Di Vito, and P. Miner. Towards a customizable PVS. Technical Report NASA CR-2000-2098 / ICASE 2000-4, ICASE and NASA Langley Research Center, Jan. 2000.

[112] L. Mannila. Invariant based programming in education - an analysis of student difficulties. *Informatics in Education*, 9(1):115–132, 2010.

[113] C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.

[114] B. Meyer. *Eiffel: The Language*. Prentice Hall, second edition, 1992.

[115] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[116] R. Milner. Logic for computable functions: description of a machine implementation. Technical Report CS-TR-72-288, Stanford, CA, USA, 1972.

[117] L. Milovanov. *Agile Software Development in an Academic Environment*. PhD thesis, Turku Centre of Computer Science, Finland, 2006.

[118] C. Morgan. *Programming from specifications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990. Available at `http://users.comlab.ox.ac.uk/carroll.morgan/PfS/`.

[119] C. Mundie, P. de Vries, P. Haynes, and M. Corwine. Trustworthy computing. Microsoft white paper, 2002. Available at `http://www.microsoft.com/mscorp/twc/default.mspx`, visited Jun 1, 2010.

[120] C. Muñoz. PBS: Support for the B-Method in PVS. Technical Report SRI-CSL-99-01, Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1999.

[121] C. Muñoz. Batch proving and proof scripting in PVS. Technical Report NASA CR-2007-214546 / NIA 2007-03, NASA Langley Research Center and National Institute of Aerospace, 2007.

[122] NASA Langley Research Center. NASA Langley PVS Libraries. Available at `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`, visited Jun 1, 2010.

[123] P. Naur. Proof of algorithms by general snapshots. *BIT Numerical Mathematics*, 6(4):310–316, 1966.

[124] G. C. Necula. Proof-Carrying code. In *Proc. of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL'97)*, pages 106–119. ACM Press, 1997.

[125] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[126] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[127] S. Nevalainen. An editor for invariant based programming. Master's thesis, Åbo Akademi University, Department of Computer Science, 2006.

[128] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[129] OMG. UML 2.2 Superstructure Specification, Feb. 2009. Document ptc/09-02-02, available at `http://www.omg.org/`.

[130] S. Owre. A brief overview of the PVS user interface. In *8th International Workshop User Interfaces for Theorem Provers (UITP'08)*, Montreal, Canada, Aug. 2008.

[131] S. Owre and N. Shankar. Abstract datatypes in PVS. Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993. Revised June 1997.

[132] S. Owre and N. Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, Aug. 1997.

[133] S. Owre and N. Shankar. The PVS Prelude Library. Technical Report SRI-CSL-03-1, Computer Science Laboratory, SRI International, Menlo Park, CA, Mar. 2003.

[134] S. Owre and N. Shankar. Writing PVS proof strategies. In M. Archer, B. D. Vito, and C. Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, number CP-2003-212448 in NASA Conference Publication, pages 1–15, Hampton, VA, Sept. 2003. NASA Langley Research Center.

[135] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.

[136] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference, Version 2.4*. Computer Science Laboratory, SRI International, Menlo Park, CA, Nov. 2001.

[137] D. L. Parnas. "Formal methods" technology transfer will fail. *Journal of Systems and Software*, 40(3):195–198, 1998.

[138] B. Plüss. A practical method for reasoning about procedures in invariant based programming. Master's thesis, National University of Rosario, 2008.

[139] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS '04: Proc. of the 19th Annual IEEE Symposium on Logic in Computer Science*, pages 32–41, Washington, DC, USA, 2004. IEEE Computer Society.

[140] J. C. Reynolds. Programming with transition diagrams. In D. Gries, editor, *Programming Methodology*, pages 153–165. Springer-Verlag, 1978.

[141] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

[142] J. Rushby, F. von Henke, and S. Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1991.

[143] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proc. of the Eighth International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer, 1996.

[144] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *Fourteenth International Symposium on Software Reliability Engineering*, pages 281–292. IEEE Computer Society, 2003.

[145] N. Shankar. Efficiently executing PVS. Project report, Computer Science Laboratory, SRI International, Menlo Park, CA, Nov. 1999.

[146] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide, Version 2.4*. Computer Science Laboratory, SRI International, Menlo Park, CA, Nov. 2001.

[147] C. Snook and R. Harrison. Practitioners' views on the use of formal methods: an industrial survey by structured interview. *Information and Software Technology*, 43(4):275–283, 2001.

[148] A. M. Turing. Checking a large routine. In *Report on a Conference on High Speed Automatic Computation, June 1949*, pages 67–69, Cambridge, UK, 1949. University Mathematical Laboratory, Cambridge University.

[149] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *Proc. of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2001.

[150] M. H. van Emden. Programming with verification conditions. *IEEE Transactions on Software Engineering*, 5(2):148–159, 1979.

[151] G. van Rossum and F. L. Drake, Jr., editors. *An introduction to Python: release 2.2.2*. Network Theory Ltd., Bristol, UK, 2003.

[152] C. Walther and S. Schweitzer. Verification in the classroom. *Journal of Automated Reasoning*, 32(1):35–73, 2004.

[153] T. Weber. SMT solvers: New oracles for the HOL theorem prover. In *Workshop on Verified Software: Theory, Tools, and Experiments (VSTTE 2009)*, Eindhoven, The Netherlands, Nov. 2009.

[154] F. Wiedijk. *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*. Springer, 2006.

[155] J. W. J. Williams. Algorithm 232 Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

[156] J. M. Wing. Weaving formal methods into the undergraduate computer science curriculum. In *Proc. of the 8th International Conference on Algebraic Methodology and Software Technology*, pages 2–9. Springer-Verlag, 2000.

[157] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[158] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4):1–36, 2009.

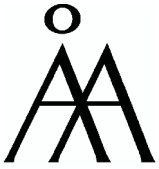
Turku Centre for Computer Science
TUCS Dissertations

92. **Arho Virkki**, The Human Respiratory System: Modelling, Analysis and Control
93. **Olli Luoma**, Efficient Methods for Storing and Querying XML Data with Relational Databases
94. **Dubravka Ilić**, Formal Reasoning about Dependability in Model-Driven Development
95. **Kim Solin**, Abstract Algebra of Program Refinement
96. **Tomi Westerlund**, Time Aware Modelling and Analysis of Systems-on-Chip
97. **Kalle Saari**, On the Frequency and Periodicity of Infinite Words
98. **Tomi Kärki**, Similarity Relations on Words: Relational Codes and Periods
99. **Markus M. Mäkelä**, Essays on Software Product Development: A Strategic Management Viewpoint
100. **Roope Vehkalahti**, Class Field Theoretic Methods in the Design of Lattice Signal Constellations
101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web: Querying and Characterizing
105. **Sampo Pyysalo**, A Dependency Parsing Approach to Biomedical Text Mining
106. **Anna Sell**, Mobile Digital Calendars in Knowledge Work
107. **Dorina Marghescu**, Evaluating Multidimensional Visualization Techniques in Data Mining Tasks
108. **Tero Säntti**, A Co-Processor Approach for Efficient Java Execution in Embedded Systems
109. **Kari Salonen**, Setup Optimization in High-Mix Surface Mount PCB Assembly
110. **Pontus Boström**, Formal Design and Verification of Systems Using Domain-Specific Languages
111. **Camilla J. Hollanti**, Order-Theoretic Mehtods for Space-Time Coding: Symmetric and Asymmetric Designs
112. **Heidi Himmanen**, On Transmission System Design for Wireless Broadcasting
113. **Sébastien Lafond**, Simulation of Embedded Systems for Energy Consumption Estimation
114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Commutation and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip
122. **Teijo Lehtonen**, On Fault Tolerance Methods for Networks-on-Chip
123. **Eeva Suvitie**, On Inner Products Involving Holomorphic Cusp Forms and Maass Forms
124. **Linda Mannila**, Teaching Mathematics and Programming – New Approaches with Empirical Evaluation
125. **Hanna Suominen**, Machine Learning and Clinical Text: Supporting Health Information Flow
126. **Tuomo Saarni**, Segmental Durations of Speech
127. **Johannes Eriksson**, Tool-Supported Invariant-Based Programming

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics



Åbo Akademi University
- Department of Information Technologies

Turku School of Economics
- Institute of Information Systems Sciences

ISBN 978-952-12-2446-1
ISSN 1239-1883

Johannes Eriksson

Tool-Supported Invariant-Based Programming

Johannes Eriksson

Tool-Supported Invariant-Based Programming

Johannes Eriksson

Tool-Supported Invariant-Based Programming