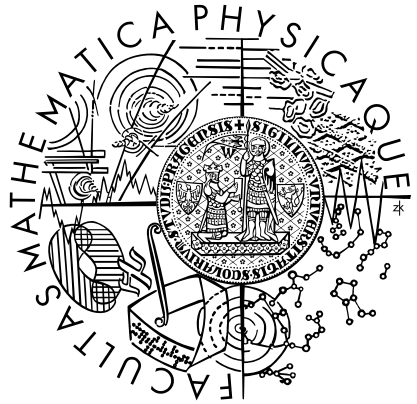
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Mikoláš Janota

Automated Theorem Proving and Program Verification

Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Prof. RNDr. Petr Štěpánek, DrSc.

Study Program: Computer Science

Prague 2005

First of all, I would like to thank my supervisor, Petr Štěpánek, for showing me the fascinating world of mathematical logic and for his advice and many helpful comments on earlier drafts of this Thesis.

Furthermore, I would like to thank Petr Horský for numerous discussions on software verification and on this Thesis.

Last but not least, I would like to thank Josef Urban for his advice he gave me during my research for this Thesis.

I declare that I have written this Master Thesis on my own and listed all used sources. I agree with lending of the Thesis.

Prague, August 11, 2005

Mikoláš Janota

Contents

Chapter 1

Introduction

Today's world is full of various devices and machines. Many are controlled by software. These are not just personal computers but also washing machines, cellular phones, air-planes, etc. In many cases our work, property or even lives depend on these machines. So on one hand, their proper functioning is of very high importance. On the other hand, these machines are constantly getting more complex and powerful. Naturally, the same holds for the controlling software.

Throughout the years it has turned out that developing SW is extremely difficult. Development of many projects was accompanied by serious problems. For example in 1981 the Federal Aviation Administration (FAA) started a program to modernize its National Airspace System (NAS). The modernization was both in SW and HW. The Advance Automation System was the main SW component of the system. The project took 16 years and as a whole it was a failure[1]; to quote from a report by United States General Accounting Office [17]:

> Over the past 16 years, FAA's modernization project have experienced substantial cost overruns, lengthy schedule delays, and significant performance shortfalls. To illustrate the centerpiece of that modernization program – the Advanced Automation System (AAS) – was restructured in 1994 after estimated costs to develop the system tripled from $2.25 billion to $7.6 billion and delays inputting significantly less-than-promised system capabilities into operation were expected to run 8 years or more over original estimates.

Other notorious examples of SW failure is the failure of Ariane 5 flight 501 or FDIV design fault in the Pentium processor (see e.g. [8]). Many, less spec-

[1] Some pieces were salvaged through follow-up programs.

tacular, SW failures are experienced daily by countless amount of computer users when their operating system stops responding or their word processor crashes etc.

These facts are somewhat surprising. When computers are human-built machines, why do they behave so unexpectedly? The answer is not straightforward but basically we are dealing with problems of two kinds.

- As it has been said above, today's SW systems are extremely complex. They grow rapidly with new capabilities of HW. In a larger system it is not feasible for one person to imagine every possible situation that might occur. Indeed, even the author of a specific piece of code very quickly forgets how exactly the code works. Misusing old code is a common cause for error.

- Secondly, there's always a gap between the original assignment the system has and its implementation. Usually, the task comes from the real world: "We need a word processor." Then, during the system is being developed, thousands of things occur which are not clear how they should be implemented.

To conclude, software projects are notoriously behind schedule (sometimes even fail), over budget and contain errors. The phenomenon was coined as *software crisis*.

In this Thesis we will present and compare nowadays techniques and tools for SW verification. The Thesis is organized as follows:

- the second chapter puts SW verification the context of software engineering

- the third chapter talks about automated and semi-automated tools that support mathematical logic

- the fourth chapter introduced Hoare style reasoning about programs

- the fifth chapter introduces how programs can be reasoned about via abstract interpretation and defines predicate abstraction

- the sixth chapter discusses benefits and drawbacks of the techniques introduce in two previous chapters

- the seventh chapter presents current systems for software verification, their weaknesses and capabilities being illustrated on a set of examples

- the eight chapter presents suggestions for current SW verification systems
- the last chapter summarizes the observations made in the Thesis

The reader is expected to be familiar with basics of mathematical logics and at least one procedural language (C/C++, Java, etc.).

Chapter 2

Software Engineering

Proper behavior of software has become so important in our society that a new discipline emerged: *software engineering*. Software engineering is comprised of methodologies for developing and maintaining software applications. These methodologies combine technologies and practices from many fields, such as computer science, project management, and others. Software application is a product and as for any other product, efficiency of development and quality and reliability of the final outcome is of interest.

Software engineering combines two worlds: *formal* and *pragmatic* [27]. In the formal world we are using mathematical tools, in particular mathematical logic and algebra. Such methods are referred to as *formal methods*.

The pragmatic world arises from experiences with the design of large software systems. Precise notation and firm basis are not in the main scope of interest; the methods are focused on economical and managerial side of the project.

As usual, the line between these two worlds might sometimes be blurred and ideally the two worlds meet during the development. With respect to the theme of this Thesis, it should be noted that mathematical logic can serve as a mean of communication.

2.1 Evolution of Programming Languages

As programming in the machine code is a difficult task, a growing set of programming languages has been developed to overcome the peculiarities to express more complex tasks by more general (and sometimes) more understandable constructs.

The issue of programming languages is slightly outside the scope of software engineering. Nevertheless, the right choice of the appropriate language is very

important and may significantly influence the efficiency of the development process.

Constructs of programming languages are in general driven by two main objectives:

- to increase their expressiveness
- to make it more difficult for the programmer to make mistakes

The history is very rich. Almost every year a bunch of new concepts and languages emerges. To name just a few: procedural programming languages (e.g. ALGOL), object oriented programming (e.g. C++, Java), garbage collector (e.g. Java, C#), etc.

Mathematical approach sets basis for functional (e.g. ML, Haskell) and logic languages (e.g. Prolog). These languages have proven that they are suitable for certain types of problems, mainly artificial intelligence. Indeed, one of the reasons why we have so many languages is that different ones are suitable for different problems.

One of the most powerful tools is *data typing*. Data types serve as a protection against inadvertent assignments and also are making the languages more suitable to model a given problem. By this way data types are helpful in static analysis of programs.

The evolution of programming languages has proven to be extremely important. It is unquestionable that it wouldn't be possible to develop contemporary SW systems without higher level languages. On the other hand, it is not the primary aim of programming languages constructs to **guarantee** programs' correctness.

We should note that despite the plethora of programming languages, in some applications, the choice of the programming language is limited. Often because of the extra requirements on HW imposed by the higher level languages. We should also note that compiler might play an important role too. For simpler language it is easier to verify correctness of the compiler. That is why we should still pay attention to the languages like for example ANSI C.

2.2 Testing

Usually, a SW application is tested before it is deployed. There are various types of tests. Basically, a SW application can be tested by automated tests, that are part of the source code, or by manual testing, i.e. the tester is working with the application and simultaneously reporting any non-standard behavior of the application. Automated tests have become very important in nowadays

SW development. Indeed, one of the approaches of to software development, *extreme programming* [3], relies on automated tests.

Thorough manual testing requires big amount of human resources and therefore it is very expensive. Naturally, writing automated tests also requires significant amount of human resource and ingenuity. Nevertheless, advocates of extreme programming claim that it is worth it to write automated tests, i.e. the amount of resources saved on debugging (due to automated tests) is greater than the resources invested in writing the automated tests. However, even a very extensive testing cannot guarantee that the application doesn't contain errors. It would be naïve to suppose that the verification methods will eventually entirely remove the need for testing. On the other hand, the verification methods can be in many cases more efficient with respect to the human resources and the verification methods have the capabilities of proving the **absence** of errors.

When erroneous behavior is discovered during testing, it is not always easy to find the primary cause of the problem in the source code. In fact, the problem is undecidable in general. This is due to the fact that undesired behavior of a piece of code can cause undesired behavior of another piece of code. This effect can grow into a chain of arbitrary length and therefore the undesired behavior may be observed by the tests far from the primary cause or the error. The process of finding the primary cause of the error in the source code after it has been observed is called *post-mortem* analysis.

To help capturing the undesired behavior as close to its primary cause as possible, two important concepts have been developed in modern programming languages, *assertions* (see e.g. [28]) and *exceptions* (see e.g. [18]). By the assertion construct the programmer expresses statements that he or she assumes to be valid at certain points of execution.

Example 2.2.1 *A C code with an assertion:*

```
void someLoop(int n, int k)
{
  assert(k > 0 && n > 0);
  i = 0;
  while (i < n)
  {
     i = i + k;
  }
}
```

In the example 2.2.1 the author of the code expects that the values of k and n are positive[1]. If the property declared by the assertion is broken, the program halts and reports the error. Assertions are usually not included in the final product and therefore they serve only as an aid for the programmers during the development phase.

The exception construct is for the cases when it is not clear, in the current context, how a specific situation should be handled. A typical use of the exception construct is illustrated by the following example.

Example 2.2.2 *A Java code throwing an exception:*

```
int[] createOnes(int n)
{
  if (n <= 0)
    throw new Exception("Illegal argument");

  int[] retv = new int[n];
  for (int i = 0; i < n; i++)
     retv[i] = 1;

  return retv;
}
```

This Java function in the example 2.2.2 allocates and returns an array of length n with all its elements set to 1. Again, the programmer didn't know what was supposed to happen when the argument of the function n is not positive. When the value of n is not positive, the code evokes (*throws*) illegal argument exception. Then, the exception is handled (*caught*) by some other part of the program. Exceptions are usually included in the final product.

Assertions and exceptions are worth noticing with respect to program verification. Instead of testing the code we can try to prove that assertions always hold and exceptions never occur.

2.3 Mathematical logic in Computer science

Mathematical logic is a powerful tool. It can be used to express things of various nature. In computer science, mathematical logic is mostly used to model behavior of various systems and then to reason about such a model. Here

[1]For the loop to terminate it is sufficient to assume $k > 0 \vee n \leq 0$, therefore the assertion that n should be positive provides additional semantic information about n.

are some examples of areas where mathematical logic and computer science meet.

- Reasoning about HW
- Reasoning about communication protocols (security)
- Reasoning about specifications
- Reasoning about the correspondence between a program and its specification
- Reasoning about programs' properties
- Automated code synthesis from specification

This list illustrates how broad the field is. It is out of the scope of this Thesis to give a detailed view of all these subjects. Instead, we give overview of nowadays principles and techniques, involving mathematical logic, that help us to write correct software.

2.4 Specifications

Before a piece of program is written, it should be clear what its purpose and function is. In other words, its function should be *specified*. During development, specifications appear in various forms and roles.

Often, the first specification is the initial motivation for the SW coming from the needs of the eventual users. Usually, this idea is proposed by someone who is not a programmer and so it may be vague and too general. Such a requirement might for example look like this: "We want a system to collect feedback from our customers."

Successful development requires good planning. Many questions have to be answered: "In what order parts of the system will be developed?", "How much time/people/resources do we need?", etc. To be able to conduct the project, the original demand has to be refined. Unclear spots have to be clarified: "Do you want to do that via the Internet or mail?" The task has to be decomposed into smaller ones: "We need a web interface, email interface a database part, etc". A good specification of the system helps us to answer these questions before we start to write the code. Naturally, people with minimal knowledge of programming get involved in the process and it is often very hard to obtain precise specification of the goal.

Nevertheless, eventually there is someone who actually implements the system. A common scenario is that partial goals are distributed among the programming team. Such a partial task might look like that: "I want you to develop a web interface with such and such properties." On one hand, the programmer has to decide what exactly the system will do – programs cannot be vague. On the other hand, the programmer might have limited knowledge about the issue itself. That is, he or she might not have sufficient knowledge to decide certain things. Therefore very precise specifications are needed here.

Some SW is written by programmers for programmers, libraries typically. Again, precise specifications are of very high importance. Source code of the library might not be publicly available and the user of the library relies solely on the documentation.

Altogether, different people with different skills and knowledge are involved in SW development. As many different people, so many different ways how they describe their products and express their requirements. Typical means of communication are natural language, drawings, diagrams, etc. In this Thesis we will be discussing *formal specifications*. More precisely, specifications expressed by the means of mathematical logic.

Throughout the history of SW development it has been discovered that not only it is difficult to develop correct SW but it is also difficult to specify what the correct behavior is. Moreover, a mistake in the design of a program can have severe economical consequences when discovered too late.

This gave birth to so-called specification languages – such as Z or VDM [34]. Later in this Thesis we will present PVS language. These languages are based on higher order logic which gives them a very high level of expressiveness. Their primary modus operandi is to write down the specification in the specification language and then use automated tools to check consistency and properties of the specification.

Second major area where logic specifications are used is when we connect a specification with source code. Then the source can be verified with respect to the logic specification; we will discuss this approach in chapter 4. Conversely, the logic specification serves as a documentation of the source code.

Sometimes, we do not insist on that the logic specification completely specifies the code's behavior. Roughly speaking, it just points out some of the desired properties. In the language JML, which will be presented later, the terms light-weight and heavy-weight are used to distinguish such character of a logic specification.

Chapter 3

Inference Systems

Automated reasoning plays an important role in SW verification and in formal methods in general. In many cases the verification process requires proving or disproving some conjectures. Technically, proving and disproving could be done by hand. Obviously, that is not preferable. Often, the proofs are not very difficult but big amount of the proofs is needed. Lack of user-friendly tools for automated reasoning is considered as one of the major obstacles to wide-spread use of formal methods.

An *inference system* is in [38] defined as follows

> "... a program or computer assisted tool which is able to perform logical operations in the framework of the formal method(s) under consideration."

Then we talk about three different kinds of inference systems

- *model checkers*
- *interactive theorem provers* (ITPs for short)
- *automated theorem provers* (ATPs for short)

Model checkers are tools that show validity, invalidity respectively, of a formulae in temporal propositional logic on a finite state-transition systems (e.g. see [29]).

Interactive theorem provers, or *proof assistants*, are in principle similar to pocket calculators. A proof assistant provides the user with a set of inference rules that help the user to construct the proof. Later in this Thesis we will present the proof assistant PVS.

Automated theorem proving has been around nearly for fifty years and these fifty were marked by substantial improvement of ATPs. There were some

impressive accomplishments in the mathematical field by ATPs. Nevertheless, ATPs still haven't made its way to the wide public. It should be noted that general theorem proving implies incompleteness [36]. That means that some conjectures are neither provable nor refutable. Moreover, there is no upper bound for length of proofs (disproofs respectively) for conjectures that are provable (refutable respectively). General ATPs, based rather on the resolution method (see e.g. [31]), lack support for arithmetic[1] which is crucial for most of the proof tasks that arise during SW verification.

Some theories are decidable (e.g. Presburger arithmetic). Various algorithms [20], called *decision procedures*, DP for short, were developed to efficiently reason in these theories. Using DPs for SW verification has a long tradition. For example in [39] a theorem prover based on the Fourier-Motzkin method of linear programming was used to implement an automated array bound checker.

Obviously the most important advantage of DPs is that they are guaranteed to return yes/no answer in a finite amount of time. Despite this fact, complexity of DPs is a significant problem. Although Presburger arithmetic is a decidable theory, it was proved [21] that every decision algorithm for the theory is super-exponential. Therefore, many DPs deal with subsets of Presburger arithmetic.

Relying merely on DPs implies losing completeness property[2]. Moreover, DPs might be implemented with bugs in them which make them hard to rely upon[3].

Therefore it is desirable to combine classical resolution-based ATPs with DPs. Such a fusion is a subject of research.

Another way how to help the general ATPs is to pre-process the proof tasks; simplify the proof tasks and add axioms that will most likely be useful (e.g. see [13]).

The techniques presented in this Thesis all rely on proving. Big amount of proof tasks have to be discharged and therefore it would not be feasible to perform all the proofs by hand. Improvements of ATPs have a direct positive impact on the range of use of these techniques. For more details on theorem proving in software engineering see [38].

[1] Conjectures in arithmetic have to be proven directly from the arithmetic axioms, which is in general very inefficient.

[2] A calculus is *complete* if every valid conjecture (i.e. holds in every model of the theory) can be proven via the calculus.

[3] This can be overcome by enhancing the DP with a possibility of producing a proof. Such a proof can be checked by a different tool (a *proof checker*).

Chapter 4

Reasoning about Programs

Since programs can be seen as mathematical structures, it is quite natural to ask whether we can prove that a program complies with its specification. Unlike testing, reasoning enables to guarantee that certain properties of a program hold for arbitrary large (even infinite) sets of inputs.

From Computation Theory we know that this task is in general undecidable[1] . Even for a program whose correctness can be proven, the proof might still be extremely complicated. A notorious example is a program that halts if and only if the Last Fermat's theorem doesn't hold. Such a program is easy to write but showing that the program never halts implies providing a proof of the theorem.

Despite these, rather pessimistic, facts we should note that the term **program** is very general and in real applications, we deal only with specific types of programs. To cite Dijkstra:

> "We must not forget that it is not our business to make programs, it is our business to design classes of computations that will display a desired behaviour." [16]

Roughly speaking, programs mostly solve problems from the real world. They are written by humans. The common scenario is that the programmer has the goal in mind and is achieving it inductively. That is, the goal is decomposed into subgoals, which can be decomposed later on etc.

4.1 Linking Programs with Logic

To be able to reason about programs we need some mathematical construct to model them. Although Turing machine has shown its usefulness in Recursion

[1]For example the *halting problem* and the *Rice's Theorem* [35].

Theory and Complexity Theory it is not very suitable to model programs written in today's programming languages.

In this chapter we will show how programs can be connected with mathematical logic. In order to do that, we need a programming language, logic language and a theory. The framework presented here is very general thus we will make as little assumptions as possible; for details see [23].

In this Thesis we will be speaking about *imperative programs*. A program operates on a finite fixed number of variables. Each variable has a *data type* (*type* for short). We will not go into details of data types; here we will assume only basic types as integer, boolean, array of thereof, etc. Most of the examples will rely on the type $NaturalNumber$ which denotes $\{0, 1, 2, \ldots\}$.

We assume that the expressions in the programs do not contain constructs with side-effects. For integers and booleans we assume classical operators ($+$, $-$, OR, AND, etc.).

In the *logic language* we will assume conventional boolean connectives, $true$, $false$, quantification and standard operators ($<$, $=$, $+$, etc.). A logic formula will usually appear in context of a program. In that case we assume that the free variables that are shared by the program and the formula are used appropriately according to their data type. So for a program containing a variable b of type boolean and x of type integer the expression $x + b < 0$ has no meaning.

We will assume a *theory* T that corresponds to the semantics of expressions in the programming language. We will use $\models C$ to denote that a formula C is valid in the theory T.

It should be noted that conventional imperative languages operate on integers with bounded domain. Then we have to face problems with overflow. There are different kinds of behavior of programs when overflow is encountered. Mostly, modulo or error[2]. For modulo behavior we should use theory that captures such behavior. For error behavior it should be proved that overflow doesn't occur. Nevertheless, in practice this is often neglected and bounded-domain integers are modeled as generic integers.

We will use the operational semantics of programs. A program's execution is modeled as a sequence of the program's states. A *state* is a total function from the program's variables to their values[3].

[2]Roughly speaking, modulo behavior guarantees that results of operations are correct modulo some number (typically maximal value + 1). Error behavior produces an error if the overflow occurs.

[3]If convenient, we can use tuples to represent states.

Example 4.1.1 *For the following program:*

x, y: NaturalNumber;
if $(x < 0) \wedge (y < 4)$ ***then***
  $x \leftarrow 0$
else
  $y \leftarrow 0$
end if

An example of a state of this program is $\{(x, 10), (y, 2)\}$.

We will use the term *predicate* to denote a well formed formula where the set of free variables is a subset of the program's variables. Thus, the following are predicates:

$$(x < 0) \wedge (y < 4)$$
$$(x = 0)$$

Note that the predicate $(x = 0) \vee (y = 0)$ holds at the end of any execution of the program from example 4.1.1.

In a program's context, a predicate represents a subset of program's possible states. This subset is formed by those states in which the predicate holds.

Example 4.1.2 *Here are some examples of predicates where* x *and* y *are both natural numbers:*

- $x = y$ *represents* $\{\{(x, 0), (y, 0)\}, \{(x, 1), (y, 1)\}, \{(x, 2), (y, 2)\}, \ldots\}$
- $x = 3$ *represents* $\{\{(x, 3), (y, 0)\}, \{(x, 3), (y, 1)\}, \{(x, 3), (y, 2)\}, \ldots\}$
- $false$ *represents the empty set of states*
- *true represents all possible states:*

$$\{\{(x, v_x), (y, v_y)\} | v_x, v_y \in \{0, 1, 2, \ldots\}\}$$

Conversely, not every set is necessarily representable by some predicate. Which sets are representable depends on the strength of the logic language and theory that is used.

Definition 4.1.3 *For any predicates* P *and* Q *such that* $\models P \Rightarrow Q$.

- *We say that the predicate* Q *is* **weaker** *than the predicate* P.
- *We say that the predicate* P *is* **stronger** *than the predicate* Q.

Especially, the predicate *true* represents all possible states and is weaker than any other predicate. The predicate $false$ represents the empty set of states and is stronger than any other predicate.

4.2 Floyd-Hoare Triples

In the previous chapter we have shown how to use predicates to represent sets of programs' states. In this section we will present how to use the above described formalism to specify program's correctness. It should be stressed, that in what follows we are not concerned with **how** the program computes but with **what** it computes.

We describe the behavior of a program by two predicates. The first predicate represents the set of states in which the program can be started; we call it the *precondition* of the program. The second set represents the set of states in which the program should terminate; we call it the *postcondition* of the program. Such description does **not** say anything about executions that begin in states that do not satisfy the precondition.

Example 4.2.1 *For the function maximum that computes maximum of two integers we have*

- **variables:** *a, b and result. All three of type integer.*
- **precondition:** *all possible states, i.e. true*
- **postcondition:** *such states where the value of result equals either to the value a or the value b and the value of result is greater or equal to the value of a and the value of b, i.e.*

$$((a = result) \vee (b = result)) \wedge (a \leq result) \wedge (b \leq result)$$

Definition 4.2.2 *The triple, precondition, program and postcondition is called a Floyd-Hoare triple. The following definition introduces notation for Floyd-Hoare triples*[4] *.*

For any predicates P, Q and a program S

- **Partial correctness**

  $\{P\}\ S\ \{Q\}$ *denotes that if the program S is executed in a state satisfying the predicate P and the execution terminates then the predicate Q is satisfied by the resulting state.*

- **Total correctness**

  $[P]\ S\ [Q]$ *denotes that if the program S is executed in a state satisfying the predicate P then the execution terminates and the predicate Q is satisfied by the resulting state.*

[4]The notation we are introducing here is slightly different from the original one. In the original notation $\{P\}S\{Q\}$ was used to denote total correctness and $P\{S\}Q$ was used to denote partial correctness.

Example 4.2.3 *In the above notation the example 4.2.1 can be written as follows:*

$a, b, result : integer;$
$\{true\}$
if $a > b$ ***then***
  $result \leftarrow a$
else
  $result \leftarrow b$
end if
$\{((a = result) \vee (b = result)) \wedge (a \leq result) \wedge (b \leq result)\}$

Comment. Here the thoughtful reader might spot a flaw. The definition doesn't say anything about which variables may or may not be changed by the program. Which means that a program that sets all three variables to 0 satisfies the same condition.

Therefore we need to refer to the initial values of a and b; for this purpose we introduce special variables (sometimes called *ghost variables*).

Example 4.2.4

$a, b, result : integer;$
$\{a = A \wedge b = B\}$
if $a > b$ ***then***
  $result \leftarrow a$
else
  $result \leftarrow b$
end if
$\{((A = result) \vee (B = result)) \wedge (A \leq result) \wedge (B \leq result)\}$

Moreover, if we want to state that, for example, a shouldn't be modified by the program, we add $a = A$ to the postcondition.

In implementations a special notation is introduced that enables expressing which variables may be changed (or which may not). Often, special notation is also introduced to refer to the initial values of variables. The problem of specifying the variables that a subprogram can/cannot change is called the *frame problem*[5].

[5]This problem gets particularly difficult when we are dealing with un-bounded amount of objects (e.g. dynamically allocated structures), because then we are not able to specify variables that should remain unchanged.

4.3 Hoare Axioms and Rules

The previous section has introduced notation that enables us to describe the desired behavior of a program but still we lack the calculus that would tell us whether the program really displays such behavior. The calculus we will present in this chapter was first introduced by C. A. R. Hoare in [25] and therefore it is referred to as *Hoare logic*.

In what follows we consider a small programming language with the following commands[6]:

$$\textbf{skip} \mid x \leftarrow E \mid S_1; S_2 \mid \textbf{while } B \textbf{ do } S \textbf{ endwhile} \mid$$

$$\mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ endif}$$

These constructs form a core of every conventional imperative language. Additional features can be added and we will make some comments on the extensions in the section 4.7.

Definition 4.3.1 *The* **skip** *command is an empty command and so it does not affect the state of the program. Therefore, for any predicate Q:*

$$\overline{\{Q\}\textbf{skip}\{Q\}}$$

Definition 4.3.2 The Assignment Axiom *For a variable V, expression E and a postcondition P*

$$\{P[V/E]\}V \leftarrow E\{P\}$$

This axiom tells us that if P is the postcondition of an assignment command, then P with all occurrences of V replaced by the expression E is the precondition.

Example 4.3.3 *For integer variables X and Y the following statements hold by the assignment axiom.*

$$\{5 > 4\}X \leftarrow 5\{X > 4\}$$

$$\{X + 1 > 10\}X \leftarrow X + 1\{X > 10\}$$

$$\{2 * Y = Y\}X \leftarrow 2 * Y\{X = Y\}$$

[6]The construct **if** B **then** C **endif** can be defined as a shorthand for **if** B **then** C **else skip endif**

Definition 4.3.4 The Sequence Rule *For any predicates P, Q, R, programs C_1 and C_2:*

$$\frac{\{P\}C_1\{Q\}\ \{Q\}C_2\{R\}}{\{P\}C_1;C_2\{R\}}.$$

The sequence rule tells us how to join pieces of code. Note that the logical rules, defined bellow, can be used to generalize this rule to:

$$\frac{\{P\}C_1\{Q_1\}\ \{Q\}C_2\{R\}\ Q_1 \Rightarrow Q}{\{P\}C_1;C_2\{R\}}.$$

Definition 4.3.5 Conditional Rules *For any predicates P, Q, boolean expression S and programs C_1 and C_2:*

$$\frac{\{P \wedge S\}C_1\{Q\}\ \{P \wedge \neg S\}C_2\{Q\}}{\{P\}\textbf{if } S \textbf{ then } C_1 \textbf{ elseif } C_2 \textbf{ endif}\{Q\}}$$

This rule splits reasoning about the if-then-else command into the case when the condition holds and when it does not.

Definition 4.3.6 The While Rule *For any predicates P, I, Q, boolean expression S and a program C:*

$$\frac{P \Rightarrow I\ \ \{I \wedge S\}C\{I\}\ \ (\neg S \wedge I) \Rightarrow Q}{\{P\}\textbf{while } S \textbf{ do } \{I\}C \textbf{ end while}\{Q\}}$$

Note that the predicate I appears both as a precondition and a postcondition of the loop body. The predicate I is called the *loop invariant*. This rule is analogous to the proof by induction in mathematics. The loop invariant holds when the program enters the loop and every subsequent iteration of the loop preserves the invariant. Note that in the while rule and the conditional rule we slightly abuse the notation because S is a boolean expression in the program but also used as a predicate.

Definition 4.3.7 Logical Rules *For any predicates P, P', Q, Q' and program C:*

- *Precondition strengthening*

$$\frac{P \Rightarrow P'\ \{P'\}C\{Q\}}{\{P\}C\{Q\}}$$

- *Postcondition weakening*

$$\frac{Q \Rightarrow Q'\ \{P\}C\{Q\}}{\{P\}C\{Q'\}}$$

4.4 Weakest Precondition

The previous section introduced a set of rules and axioms that can be used to determine validity of a Floyd-Hore triple. Nevertheless, the calculus doesn't provide us with a way how the validity could be computed. Therefore, we will extend the scheme with the operator of *weakest precondition*.

Definition 4.4.1 *For any program C and postcondition Q*

- **Weakest liberal precondition** $wlp(C, Q)$ *denotes such a predicate W that if for any predicate P, $\{P\}C\{Q\}$ then $W \Rightarrow P$*
- **Weakest precondition** $wp(C, Q)$ *denotes such a predicate W that if for any predicate P, $[P]C[Q]$ then $W \Rightarrow P$*

For a fixed program, the weakest precondition operator is in the role of a predicate transformer. Roughly speaking, given the set of states in which the program should terminate, the weakest precondition operator returns the set of all possible initial states.

In the following we will use $\equiv$ to denote graphical equality.

Lemma 4.4.2 *The following statements hold*

1. [7] $wlp(X \leftarrow E, Q) \Leftrightarrow Q[X/E]$
2. $wlp(S_1;\ S_2, Q) \Leftrightarrow wlp(S_1, wlp(S_2, Q))$
3. $wlp(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ endif}) \Leftrightarrow ((B \wedge wlp(S_1, Q)) \vee (\neg B \wedge wlp(S_2, Q))$
4. $wlp(S, Q) \wedge B \Rightarrow wlp(S_1, wlp(S, Q))$
   where $S \equiv \textbf{while } B \textbf{ do } \{I\}S_1 \textbf{ end while}$
5. $wlp(S, Q) \wedge \neg B \Rightarrow Q$
   where $S \equiv \textbf{while } B \textbf{ do } \{I\}S_1 \textbf{ end while}$
6. $\{P\}S\{Q\}$ *iff* $P \Rightarrow wlp(S, Q)$

Since in our language **while** command is the only command that can cause a program not to terminate, the same properties hold for wp for programs that do not contain loops.

Lemma 4.4.3 *Let S, S_1, S_2 be programs that do not contain the* **while** *command. Then the following statements hold:*

[7] Compare with the assignment axiom (definition 4.3.2).

1. $wp(X \leftarrow E, Q) \Leftrightarrow Q[X/E]$
2. $wp(S_1;\ S_2, Q) \Leftrightarrow wlp(S_1, wlp(S_2, Q))$
3. $wp(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ endif}) \Leftrightarrow ((B \wedge wp(S_1, Q)) \vee (\neg B \wedge wp(S_2, Q))$

Note that the equivalences 1-3 in both lemmata provide a way how to back-propagate a postcondition. The weakest precondition can be computed for programs using only assignments and **if-then-else** commands.

Example 4.4.4 *We will show that after an execution of the maximum function, the result is greater or equal to the first argument. We will do this by computing the weakest precondition, using lemma 4.4.3.*
$wp($**if** $a > b$ **then** $result \leftarrow a$; **else** $result \leftarrow b$ **end if**, $result \geq a) \equiv^{(by3)}$
$(a > b \wedge wp(result \leftarrow a, result \geq a)) \vee$
$\vee (a \leq b \wedge wp(result \leftarrow b, result \geq a)) \equiv^{(by1)}$
$(a > b \wedge a \geq a) \vee (a \leq b \wedge b \geq a) \equiv^{(by\ logic)}$
$true$

This shows that the maximum function brings about the postcondition independently from the state the execution begins in.

Unfortunately, the propagation of the weakest preconditions is not possible for loops[8]. To reason about loops the while rule has to be used instead.

4.5 Total Correctness

Intuitively, a program might not terminate only due to an infinite loop. To show that a loop is finite we need to provide well-founded ordering (i.e. with no infinite decreasing chains) and an expression T that is decreased w.r.t. the ordering by each iteration of the loop.

Example 4.5.1 *For the following program, the expression* $T =^{def} 100 - i$ *decreases, w.r.t* $<$*, by* 1 *each iteration of the loop. Since the predicate* $i \leq 100$ *is an invariant of the loop,* $0 \leq T$ *holds and therefore* T *is always a natural number. The ordering* $<$ *on natural numbers is well-founded.*

$i, s : integer;$
$i \leftarrow 0;$
$s \leftarrow 0;$
while $i < 100$ ***do***

[8]This is not a very surprising fact since *while* command is the source of undecidability.

$s \leftarrow i^2 + s;$
$i \leftarrow i + 1;$
end while

Total correctness can be shown by first demonstrating the partial correctness and then showing termination.

4.6 Annotated Programs

The Hoare logic gives us a mean to reason about programs via standard mathematical logic. A natural question is, how it should be applied to programs. In this section we will introduce a concept of *annotated programs*. An annotated program is such a program where preconditions, postconditions and loop invariants are inserted directly in the source code. Note that a precondition can be joined with the postcondition of the previous command.

Example 4.6.1

1: $\{true\}$
2: $i \leftarrow 0;$
3: $\{i = 0\}$
4: $k \leftarrow i + 1;$
5: $\{k = 1 \wedge i = 0\}$

It is not required for each command to be surrounded by annotations (especially simple assignments). Therefore the previous example can be rewritten as follows:

Example 4.6.2

$\{true\}$
$i \leftarrow 0;$
$k \leftarrow i + 1;$
$\{k = 1 \wedge i = 0\}$

If it is to be verified that the annotations correspond to the program, the annotated source code is translated to set of conjectures using Hoare logic.

Example 4.6.3 *To verify that the annotations in the example 4.6.1 correspond to the source code.*

- *For lines 1-3 we need to show*

$$\{true\}i \leftarrow 0\{i = 0\}$$

which is translated (using points 6 *and* 1 *of the lemma 4.4.3) into*

$$true \Rightarrow 0 = 0$$

- *Lines 3-5 we need to show*

$$\{i = 0\}k \leftarrow i + 1\{k = 1 \wedge i = 0\}$$

which is translated (using points 6 *and* 1 *of the lemma 4.4.3) into*

$$(i = 0) \Rightarrow ((i + 1 = 1) \wedge (i = 0))$$

Such implications are called *verification conditions* and automated reasoning can be used to discharge them.

4.7 Extensions to Hoare Logic

In the previous sections we have presented a calculus for a very basic programming language. Nevertheless, the calculus may be extended to reason about more complicated constructs. In this section we will present a few examples of such extensions.

To be able to reason about function calls, we need to provide the function's precondition and postcondition.

Example 4.7.1
Requires: $a > b$
Ensures: $(a > c \Rightarrow \mathbf{result} = a) \wedge (c > b \Rightarrow \mathbf{result} = b) \wedge$
$(\mathbf{a} \geq \mathbf{c} \wedge \mathbf{c} \leq \mathbf{b} \Rightarrow \mathbf{result} = \mathbf{c})$
function *Trim(a, b, c : NaturalNumber)* **returns** *NaturalNumber;*
begin
  returnValue: NaturalNumber;
  if $a > c$ ***then***
    $returnValue \leftarrow a$
  else if $c > b$ ***then***
    $returnValue \leftarrow b$
  else
    $returnValue \leftarrow c$
  end if
  return $returnValue$
end

Here, we can see two additional constructs:

- **Requires** part, i.e. the precondition that must be established before the function is called.
- **Ensures** part[9] , i.e. the postcondition that will hold after the function terminates and the execution of the function begun in a state satisfying the **Requires** part.

The following example illustrates how such a specification of a function can be used when reasoning about a function call:

Example 4.7.2 *Consider the function from the example 4.7.1. Then showing validity of the triple*

$$\{true\}\ trimmed \leftarrow Trim(4, 10, 5)\ \{trimmed = 5\}$$

is translated to

- $true \Rightarrow 4 < 10$*, i.e. that the* **Requires** *part is guaranteed by the precondition.*
- $((4 > 5 \Rightarrow trimmed = 4) \wedge (5 > 10 \Rightarrow trimmed = 10) \wedge$
  $(4 \geq 5 \wedge 5 \leq 10 \Rightarrow trimmed = 5)) \Rightarrow (trimmed = 5)$,
  i.e. that the **Ensures** *part of Trim in the current context implies the desired postcondition.*

Note that the body of the function does not appear in the proof concerning the call of the function. Thus, when reasoning about the function itself we show that the function's **Requires** and **Ensures** parts are the function's valid precondition and postcondition. When reasoning about function calls, we rely merely on **Requires** and **Ensures**. For more details on function calls see [23].

Other extensions are needed when reasoning about object oriented programs. Namely, *object invariants*. Use of an object invariant is illustrated by the following example:

Example 4.7.3 *An example of a Java class with an object invariant* [10]*.*

[9] In conventional imperative programming languages (e.g. Java), a function may modify parameters passed by value. Such modifications are not visible to the caller of the function and therefore in the **Ensures** part we are always referring to the initial values of these parameters.

[10] In this example we are using JML as the annotation language, for more details on JML see section 6.1.

```
class Class1
{
   /* The object invariant */
   /* @ invariant j + i = N &
      @           N >= 0 & N < 100 &
      @           j >= 0 & i > 0
   */

   int i, j;
   int N;

   //@ requires initN < 100;
   public Class1(int initN)
   {
      N = initN;
      i = 0;
      j = N;
      /* The constructor has established the invariant */
   }

   //@ \result == j;
   public int readJ()
   {
      return j;
   }

   //@ requires j > 0;
   public void incI()
   {
      /* We know that the object invariant holds now */
      i = i + 1; /* Here the object invariant is broken */
                 /* for a while */
      j = j - 1; /* The contract specifies that this method */
                 /* cannot be called when j <= 0. */
      /* Here the object invariant holds again. */
   }
}
```

Note that if we hadn't specified the conditions on ranges of the member fields we would have run into problems with integer overflow.

Informally, the object can break its invariant but the states in which the invariant is broken shouldn't be visible to the outer world. For the above example it is sufficient to prove that the constructor establishes the object

invariant and every method of a class implicitly has the invariant as a part of its **Ensures** and **Requires**[11]. For details on object invariants see e.g. [33].

It should also be noted that we have been discussing only *sequential* programs. For extensions on concurrent programs see [1].

4.8 Design by Contract

The previous section illustrated how Hoare logic can be extended so it can be used in object oriented programming languages[12]. Annotations bring another dimensions to programs and the question is how this should be reflected by the design process. A common approach is *design by contract* [32].

In design by contract the main idea is to approach programming as a type of a business transaction. For a procedure, the implementation of the procedure is the *supplier*, the code that calls the procedure is the *client*, the logic specification of the procedure is the *contract*. Then the client is responsible for establishing the precondition given by the contract and the supplier is responsible for delivering the postcondition given by the contract. Note that the supplier does **not** have any obligations whatsoever when the precondition is not established[13].

A class implementation, as a whole, has to maintain its object invariants[14].

[11] One major problem of object invariant is the *call-back*, which is a situation when a method calls a different object that calls back the initial object.

[12] We should note that similar extensions are possible for other languages. For example Spark Ada is a subset of Ada with annotations.

[13] Such a situation is simply a bug.

[14] A specification language may enable to specify which of the object invariants are visible for other parts of code. Therefore only the visible invariants are part of the contract.

Chapter 5

Predicate Abstraction

Another technique, based rather on algebraic methods, for proving properties of programs is *abstract interpretation* [7]. Abstract interpretation has been widely used for optimizing programs. In this chapter we will show how abstract interpretation can be used for verifying programs' correctness.

First we will give a brief introduction to abstract interpretation then we will define predicate abstraction and mention some of its modifications and extensions.

5.1 Motivation for Abstracting Programs

The main idea of abstract interpretation is to group program's states according to some properties. The idea is illustrated on the following example.

Example 5.1.1 *Consider the program:*

```
1: a: NaturalNumber;
2: if odd(a) then
3:     a ← a + 1
4: end if
5: if odd(a) then
6:     ERROR
7: end if
```

How do we show that **ERROR** is never reached? One could reason as follows.

1. Initially value of a is either odd or even – we don't know.
2. If value of a was odd then 1 is added to it therefore making the value of a even. If value of a was even then nothing happens and value of a stays even.

3. At the end, value of a must be even and the expression $odd(a)$ is evaluated to $false$.

Apparently, all we needed to know about the value of a is whether it is odd or even. To formalize the idea we construct an *abstract state space*. We will refer to the original state space as the *concrete state space*. Each member of the abstract space, an *abstract state*, represents some subset of the concrete state space.

The concrete space of the program from the above example is:

$$\{0, 1, 2, \ldots\}$$

Then, consider an abstract state space $S^A \equiv^{def} \{Odd, Even\}$. Where:

- the abstract state Odd represents the set $\{1, 3, 5, \ldots\}$
- the abstract state $Even$ represents the set $\{0, 2, 4, \ldots\}$

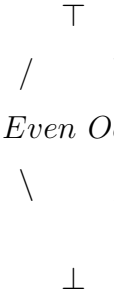
Now we can construct an abstracted versions of the operations. For the operation "+1" we can define its abstracted version $\overline{+1}$ as follows:

$$Odd\ \overline{+1} =^{def} Even$$

$$Even\ \overline{+1} =^{def} Odd$$

Such a definition is reasonable because every odd number plus 1 is even and every even number plus 1 is odd. Such an abstract state space wouldn't be sufficient to model all possible programs. For example, it wouldn't be possible to construct abstracted version of the operation "division by 2"[1]. In order to be able to do that, we require the abstract state space to form a (complete) lattice. The above introduced state space can be completed into the lattice:

Figure 5.1.2



[1]Knowing only that a number is odd/even is not sufficient to infer whether that number divided by 2 is odd/even.

Where

- the abstract state $\top$ represents the set $\{0, 1, 2, \ldots\}$
- the abstract state $\bot$ represents the empty set

Then the operation "/2" can be abstracted as follows[2]:

$$Odd\ \overline{/2} =^{def} \top$$

$$Even\ \overline{/2} =^{def} \top$$

$$\top\ \overline{/2} =^{def} \top$$

$$\bot\ \overline{/2} =^{def} \bot$$

Then an abstract version of the program can be constructed. Intuitively, the abstract program computes the same way as the original program but on the abstract state space. The abstracted program can be reasoned about more easily because the abstract state space is substantially smaller than the concrete state space. Usually lattices of abstract states of finite height are used.

For more details on abstract interpretation see [7, 6].

5.2 Transition System

Instead for some programming language we will define the abstraction on a *transition system*.

Definition 5.2.1 *A transition system is a triple* (S, I, R)*. Where*

- S *is the set of* **states**,
- $R \subseteq S \times S$ *is the* **transition relation**,
- $I \subseteq S$ *is the set of* **initial states**.

We say that if for any two states x *and* y, $R(x, y)$ *holds, that* y *is a* **successor** *of* x*. Let* M *be a natural number* [3]*, then an* **execution trace** *is a sequence of states* $x_0, x_1, \ldots, x_M$ *such that*

- $R(x_i, x_{i+1})$ *holds for every* $0 \leq i < M$ *and*

[2] Obviously such an abstraction is too coarse.

[3] The notion of trace can be extended to traces of infinite length.

- $x_0 \in I$.

Often we use just the term **trace** *for execution trace.*

We use the notation R^* for the reflexive and transitive closure of the relation R. We say that a state x is **k-reachable** *if there exists a a sequence of states $x_0, x_1, \ldots, x_k$ of length k such that $x_0 \in I$ and $x_k \equiv x$.*

We say that a state x is **reachable** *if there exists a natural number k such that x is k-reachable.*

We say that a transition system is **deterministic** *if for any states x, y, z $(R(x, y) \wedge R(x, z)) \Rightarrow (y = z)$. A system is* **non-deterministic** *if it is not deterministic.*

As in the previous chapter, we will use predicates to represent sets of states. The following definition defines some common terms used in the context of predicate abstraction.

Definition 5.2.2 *A predicate P is called the* **safety property** *if we want to show that P holds in all reachable states. We say that a state x* **violates** *the safety property P if $\neg P(x)$ holds. The states that violate the property P are called* **error states**. *A* **counter-example trace** *is such a trace $x_0, x_1, \ldots, x_M$ that $\neg P(x_M)$ holds.*

5.3 Transition Systems and Programs

The concept of a transition system is more general than the concept of a program defined by source code. Any program can be translated into a transition system. Here, we will briefly show how this is done.

Consider a program that contains a finite number of variables:

$$x_1 : D_1, x_2 : D_2, \ldots x_n : D_n$$

Where D_i is a domain for x_i. The domains can be infinite in general. Then we introduce one special variable pc for the program counter that has a finite domain D_{pc}. Then the state space is:

$$D_1 \times D_2 \times \ldots \times D_n \times D_{pc}$$

The construction of the transition relation will be illustrated on the example 5.1.1 from the previous section.

Example 5.3.1

- *The state space is formed by pairs* $< pc, a >$ *where* $pc \in \{1 \dots 7\}$ *and* $a \in \{0, 1, 2, \dots\}$
- *Initial states are represented by the predicate* $pc = 2$
- *The transitions relation* $R(< pc, a >, < pc', a' >)$ *holds iff any of the following predicates hold:*

$$pc = 2 \wedge odd(a) \wedge pc' = 3 \wedge a' = a$$

$$pc = 2 \wedge \neg odd(a) \wedge pc' = 5 \wedge a' = a$$

$$pc = 3 \wedge pc' = 5 \wedge a' = a + 1$$

$$pc = 5 \wedge odd(a) \wedge a' = a \wedge pc' = 6$$

$$pc = 5 \wedge \neg odd(a) \wedge a' = a \wedge pc' = 7$$

- *The safety property is* $\neg(pc = 6)$*. In other words, the predicate* $pc = 6$ *represents the set of error states.*

5.4 Predicate Transformers

Similarly as in the Hoare-like approach we can define predicate transformers.

Definition 5.4.1 *For a transition relation* R *on a set of states* S *and for a set* $M \subseteq S$ *represented by predicate* P*.*

- $post[R](P) = \exists q'(R(q', q) \wedge P(q'))$
- $\widetilde{pre}[R](P) = \forall q'(R(q, q') \Rightarrow P(q')$

The predicate $post[R](P)$ represents the set of successors of states represented by P. The predicate $\widetilde{pre}[R](P)$ is the weakest precondition for P. Note that the universal quantifier in the definition of $\widetilde{pre}$ is necessary because the transition system need not to be deterministic.

5.5 Abstraction of a Transition System

For a transition system we want to build an *abstract transition system*. We will use the adjective "concrete" to refer to the system that is being abstracted. The following definition introduces a pair of functions that establishes the correspondence between an abstract and a concrete state space.

Definition 5.5.1 *Consider two distinct sets of states S and S^A. Let $\mathcal{PS}$ denote the set of predicates on S. Two functions $\alpha : \mathcal{PS} \to S^A$ and $\gamma : S^A \to \mathcal{PS}$ form a* **Galois connection** *if*

- $\alpha \circ \gamma$ *is the identity*
- *for any $P \in \mathcal{PS}$ representing some subset of S*

$$\models P \Rightarrow \gamma(\alpha(\phi))$$

The function α is called the *abstraction function* and it associates any predicate on concrete states with an abstract state. The function γ is called the *concretization function* or *meaning function*; it associates any abstract state with a predicate that represents the corresponding set of concrete states[4]. The function α loses information but in a safe way; when a set of concrete states is abstracted and concretized again, the original set is contained in the result. Note that the function γ doesn't lose information.

The following definition defines when a transitions system is an abstraction of another transition system.

Definition 5.5.2 *Consider a transition system $T = (S, I, R)$. Then a system $T^A = (S^A, I^A, R^A)$ is a* **conservative abstraction** *of T iff*

- *For any $s \in I$ there exists $s^a \in I^A$ such that $\gamma(s^a)(s)$*
- *For any abstract state, s_1^a, and concrete states s_1 and s_2,*

$$s_1 \in \gamma(s_1^a) \wedge R(s_1, s_2) \Rightarrow (\exists s_2^a \in S^A)(R^A(s_1^a, s_2^a) \wedge \gamma(s_2^a)(s_2))$$

The first part of the above definition expresses that the set of initial states of the abstract system covers all the concrete initial states. The second part of the definition expresses that the abstract transition relation covers the concrete transition relation. Altogether this means that every execution trace of the concrete system is represented by at least one execution trace of the abstract system.

The second part of the definition is illustrated by the following diagram.

[4]For an abstract state s^a, $\gamma(s^a)$ returns a predicate P that represents some set $M \subseteq S$. Therefore, for any concrete state $s \in S$, $\gamma(s^a)(s)$ iff $s \in M$.

Figure 5.5.3

$$\begin{array}{ccc} s_1^a v & \rightarrow^{R^A} & s_2^a \\ \downarrow \gamma & & \downarrow \gamma \\ P_1 & & P_2 \\ \uparrow \in & & \uparrow \in \\ s_1 & \rightarrow^{R} & s_2 \end{array}$$

In the following, we will be discussing only conservative abstractions thus we will use the word abstraction to denote conservative abstraction.

Definition 5.5.4 *An* **abstract execution** *is a sequence of abstract states* $s_0^a, s_1^a, \ldots, s_M^a$*, such that* $s_0^a \in I^A$ *and* $R^A(s_i^a, s_{i+1}^a)$ *holds for* $0 \leq i < M$*. An* **abstract counter-example trace** *is an abstract execution* $s_0^a, s_1^a, \ldots, s_M^a$ *such that there exists a concrete state* $s \in S$*, such that* $\gamma(s_M^a)(s) \wedge \neg P(s)$*.*

Definition 5.5.5 *We say that a concrete counter-example trace* $s_0, s_1, \ldots, s_M$*,* **corresponds** *to the abstract counter-example trace* $s_0^a, s_1^a, \ldots, s_M^a$*, if these conditions are satisfied:*

- $s_i \in \gamma(s_i^a)$ *for* $i = 0 \ldots M$
- $s_0 \in I$ *and* $\neg P(s_M)$
- $R(s_i, s_{i+1})$ *holds for* $0 \leq i < M$

Conservative abstraction guarantees that all concrete traces are covered by the abstraction traces. Conversely this is not true in general. Thus there might be an abstraction trace with no concrete trace corresponding to it. To distinguish these two cases we introduce the following definition.

Definition 5.5.6 *An abstract trace is called* **real trace** *if there exists a concrete trace corresponding to it. Conversely, if there is no concrete trace corresponding to the abstract trace then the abstract trace is called* **spurious trace***.*

5.6 Predicate Abstraction

Here we are going to present a special case of abstract interpretation called *predicate abstraction*. Predicate abstraction was first presented by Graf and Saïdi in 1997 [37].

The predicate abstraction is induced by a finite set of predicates $\mathcal{P} \equiv \{p_1, \ldots, p_n\}$ on a concrete set of states S.

First we note that the set of predicates $\mathcal{P}$ induces an equivalence relation on the set of concrete states.

Definition 5.6.1 *Let $\mathcal{P} \equiv \{p_1, \ldots, p_n\}$ be a set of predicates on a set of states S. Then we define relation $\simeq_{\mathcal{P}}$ as follows. For any two states $x, y \in S$, $x \simeq_{\mathcal{P}} y$ iff*

$$p_i(x) \Leftrightarrow p_i(y) \text{ for all } i = 1 \ldots n$$

Roughly speaking, two states are equivalent w.r.t. $\simeq_{\mathcal{P}}$ if and only if they satisfy the same subset of predicates of $\mathcal{P}$. Note that the equivalence relation $\simeq_{\mathcal{P}}$ defines a partitioning of the state space S, i.e. each member of the state space belongs exactly to one equivalence class of the relation $\simeq_{\mathcal{P}}$.

Intuitively, equivalence classes of the $\simeq_{\mathcal{P}}$ define the best "resolution" which the set of predicates $\mathcal{P}$ enables.

Definition 5.6.2 *Let $\mathcal{P} \equiv \{p_1, \ldots, p_n\}$ be a set of predicates on a set of states S. Let $B \equiv \{B_1, \ldots B_n\}$ be a set of boolean variables. Then the* **abstract state space** *S^A induced by the set $\mathcal{P}$ is the set of normalized [5] boolean expressions on the variables $B_1, \ldots, B_n$. An* **abstract state** *will be denoted by $expr^A(B1, \ldots, B_n)$.*

Now, when we have defined the abstract state space, we need to define the correspondence between the abstract state space and the concrete state space. Following the approach of abstract interpretation we will define a Galois connection.

Definition 5.6.3 *Let $\mathcal{P} \equiv \{p_1, \ldots, p_n\}$ be a set of predicates on a set of states S. Let S^A be the abstract state space induced by $\mathcal{P}$.*

We will use $\mathcal{PS}$ to denote the class of predicates on S. $expr^A[\overline{B}/\overline{p}]$ will denote a predicate which is the expression $expr^A$ with each occurrence of B_i replaced by p_i. Then the functions $\alpha : \mathcal{PS} \to S^A, \gamma : S^A \to \mathcal{PS}$ are defined as follows. For a boolean expression $expr^A$ on the variables $B_1, \ldots, B_n$:

[5] In general the set of boolean expressions on $B_1, \ldots, B_n$ is infinite because of redundancies (e.g. $B_1 \vee B_1 \vee B_1$). The redundancies can be avoided by considering only normal forms of the expressions (e.g. DNF).

- $\gamma(expr^A) = expr^A[\overline{B}/\overline{p}]$
- $\alpha(\phi) = \bigwedge\{expr^A(B_1, \ldots, B_n) | \phi \Rightarrow expr^A[\overline{B}/\overline{p}]\}$

The above definition is based on the idea that each boolean variable B_i corresponds to the predicate p_i.

Example 5.6.4 *Consider the set of natural numbers as the concrete state space. Let $\mathcal{P} \equiv \{p_1, p_2\}$, where $p_1 \equiv (n > 10)$ and $p_2 \equiv (n < 5)$. Then the equivalence classes of $\simeq_{\mathcal{P}}$ are:*

$$\{5, \ldots, 10\}$$
$$\{0, \ldots, 4\}$$
$$\{10, 11, \ldots\}$$
$$\{\}$$

These equivalence classes are represented by these abstract states respectively:

$$\neg B_1 \wedge \neg B_2$$
$$\neg B_1 \wedge B_2$$
$$B_1 \wedge \neg B_2$$
$$B_1 \wedge B_2$$

Note that the functions γ and α from the definition 5.6.3 form a Galois connection. Moreover, each equivalence class of the equivalence relation $\simeq_{\mathcal{P}}$ is represented by one abstract state[6]. Therefore for every concrete state $s \in S$ there exists an abstract state $s^a \in S^A$ such that $\gamma(s^a)(s)$ (recall that the equivalence classes of $\simeq_{\mathcal{P}}$ form a partitioning of the concrete state space S).

Now we have everything that is needed to define predicate abstraction of a transition system.

Definition 5.6.5 *Let $\mathcal{P} \equiv \{p_1, \ldots, p_n\}$ be a set of predicates on a set of states S. Let $T \equiv (S, I, R)$ be a transition system. Then the* **predicate abstraction** *of the system T, $T^A \equiv (S^A, I^A, R^A)$ is defined as follows:*

- *S^A is the abstract state space induced by $\mathcal{P}$*
- *$I^A =^{def} \alpha(P_I)$, where P_I is a predicate that represents I*
- *$R^A(s_1^a, s_2^a) \Leftrightarrow s_2^a = \alpha(post[R](\gamma(s_1^a))$*

Informally, the definition 5.6.5 defines the best (with respect to the set $\mathcal{P}$) conservative abstraction of the concrete transition system.

[6]Each equivalence class of the relation $\simeq_{\mathcal{P}}$ is represented by a boolean expression of the form $C_1 \wedge C_2 \wedge \ldots \wedge C_n$, where $C_i \equiv B_i$ or $C_i \equiv \neg B_i$. Such expressions are called *complete monomials*.

5.7 Counter-example Refinement

In the previous sections we have defined how the predicate abstraction can be build. The key issue is how to find a good set of predicates that is used to build the abstraction.

This section introduces a semi-algorithm that successively *refines* the abstraction. In this semi-algorithm, the abstraction is refined each time a spurious trace is found. Therefore it is called *counter-example refinement* [11]. The abstraction was defined so that it *over-approximates* the original system. That means that if the **abstract** transition system does not contain any abstract counter-example trace then the **concrete** transition system does not contain any counter-example trace, i.e. the concrete transition system is *safe*. On the other hand, the abstract transition system may contain a spurious counter-example trace (see definition 5.5.6), i.e. an abstract trace such that there is no corresponding counter-example in the concrete transition system.

Definition 5.7.1 *A* **counter example refinement loop** *is a semi-algorithm*[7]*. The input is a transition system. The result is one of the following:*

a) The loop terminates and returns that the input system is safe.

b) The loop terminates and returns a concrete counter-example trace. Such a trace shows that the input system is not safe.

c) The loop does not terminate.

The loop consists of four phases.

1. *("abstraction") An abstraction induced by a finite set of predicates $\mathcal{P}$ is built.*

2. *("verification") An abstract counter-example trace is sought in the abstracted system. If no such trace is found* **return safe**. *If an abstract counter-example trace $\mathcal{C}$ is found* **goto** *phase 3.*

3. *("real vs. spurious test") The abstract counter-example trace $\mathcal{C}$, found in the phase 2, is tested whether it is a real trace or a spurious trace. If $\mathcal{C}$ is a real trace,* **return** *the concrete counter-example trace that corresponds to the real trace $\mathcal{C}$. If $\mathcal{C}$ is a spurious trace then* **goto** *phase 4.*

4. *("refinement") Extend the set $\mathcal{P}$ with new predicates such that the new abstract transition system will not contain the spurious trace $\mathcal{C}$.*
   goto *phase 1.*

[7] The loop is not guaranteed to terminate thus it wouldn't be precise to call an algorithm.

The fourth phase (refinement) guarantees that the refined system will not contain the same spurious trace that has just been found. Nevertheless, the refinement does not guarantee that there are no **other** spurious traces. That is why the counter-example loop is just a semi-algorithm; it is not guaranteed to terminate[8].

The refinement is done by adding new predicates to the set $\mathcal{P}$. A straightforward approach is to add the predicates manually; that has been successfully applied for protocol verification (e.g. [10, 12, 37]). New predicates can also be computed automatically from the proof of the fact that the counter-example is spurious.

5.8 Syntactic Forms of Predicates

The fully automatic tools mostly rely on such a set of possible predicates that the language is decidable (e.g. [24]). Interesting extension was suggested in [22] where fresh Skolem constants are used to supply predicates with universal quantification. Such predicates are crucial for reasoning about arrays or other complex structures. On the other hand, such predicates are hard to discover automatically.

5.9 Efficiency

The abstract state space induced by a finite set of predicates is finite. Unfortunately, the size of the abstract space is exponential to the number of predicates; therefore it can get too large during the refinement process. Another source of efficiency problems are time requirements during the building of the abstract relation. In both abstract state space building and abstract relation building, calls to the prover form the most expensive part.

A common approach is to decompose the concrete transition relation into a finite set of transitions of the form

$$g_i(\overline{x}) \longrightarrow \overline{x} \leftarrow ass_i(\overline{x})$$

where $g_i(\overline{x})$ is a boolean expression (a *guard*), $\overline{x}$ is a finite tuple of variables that form the concrete state.

Example 5.9.1 *Consider the transition system from the example 5.3.1, the transition relation can be rewritten into the following set of guarded assignments:*

[8] This is inevitable because the problem is undecidable in general.

1. $pc = 2 \wedge odd(a) \longrightarrow (pc \leftarrow 3, a \leftarrow a)$
2. $pc = 2 \wedge \neg odd(a) \longrightarrow (pc \leftarrow 5, a \leftarrow a)$
3. $pc = 3 \longrightarrow (pc \leftarrow 5, a \leftarrow a + 1)$
4. $pc = 5 \wedge odd(a) \longrightarrow (a \leftarrow a, pc \leftarrow 6)$
5. $pc = 5 \wedge \neg odd(a) \longrightarrow (a \leftarrow a, pc \leftarrow 7)$

Usually, some over-approximation of the abstract relation is computed.

Example 5.9.2 *In the original paper on predicate abstraction [37] only particular elements of the abstract lattice were considered, monomials*[9] *on* $B_1, \ldots, B_n$*. Then the abstraction function was defined as follows:*

$$\alpha'(P) =^{def} \bigwedge\{B_i | P \Rightarrow p_i\}$$

The concrete transition relation was assumed to be a finite set of guarded assignments. Therefore the abstract transition relation was computed separately for each guarded assignment.

- *if* $expr^A[\overline{B}/\overline{p}] \Rightarrow \neg g_i$ *then* $\tau_i^A =^{def} false$
- *otherwise:*

$$\tau_i^A(expr^A) =^{def} \bigwedge_{j=1}^{n} \begin{cases} B_j & \text{if } post[\tau_i](expr^A[\overline{B}/\overline{p}]) \Rightarrow p_j \\ \neg B_j & \text{if } post[\tau_i](expr^A[\overline{B}/\overline{p}]) \Rightarrow \neg p_j \\ true & otherwise \end{cases}$$

Notice that in the computation of the abstract transition relation, in the example 5.9.2, dependency between predicates from $\mathcal{P}$ is neglected. This is a common over-approximation technique which is called the *cartesian abstraction* (see e.g. [2]).

[9] A monomial on $B_1, \ldots, B_n$ is a conjunction of B_i's and $\neg B_i$'s where each B_i appears at most once. Furthermore, we consider $false$ and $true$ as monomials.

Chapter 6

A Review of Existing Systems for Verification

6.1 Java Modeling Language (JML)

JML is a language for annotating Java programs. It follows the scheme of design by contract. The language has been designed to be as close to the standard Java code as possible. For example, operators as ==, &&, etc. are common for both JML and Java.

Example 6.1.1 *An example of a Java source annotated with JML:*

```
/*@ requires true;
  @ ensures (\return == a || \return == b) && \return >= a
  @          && \return >= b;
  @*/
int max(int a, int b)
{
 if (a > b)
   return a;
 else
   return b;
}
```

Since "/* ... */" is a Java comment, a standard Java parser will simply skip the first four lines. On the contrary, for a JML parser the "@" sign signifies that the commentary is actually a JML annotation. The keyword "requires" expresses precondition and "ensures" postcondition[1] of the function. Note the

[1] As we have mentioned earlier, changes of values of parameters made inside the function are not visible to the caller, therefore the formal parameters in the ensures part implicitly refer to the initial values of parameters.

keyword "\result" which is used to denote the result of a function.

On top of the Hoare-style annotations special syntax is added to describe OOP specific properties (e.g. object invariants, inheritance). The frame problem is addressed by the "assignable" pragma that enables specifying variables (visible to the caller) that can be changed in a method.

Example 6.1.2 *In the "assignable" pragma a special keyword "\nothing" can be used to express that nothing can be changed inside the method. The following is a specification of a function that returns a maximal element of a given integer array.*

```
/*@ assignable \nothing;
  @ requires a != null && a.length > 0;
  @ ensures (\forall int i1; i1 >= 0 && i1 < a.length;
  @                          a[i1] <= \result);
  @ ensures (\exists int i1; i1 >= 0 && i1 < a.length;
  @                          ai[i1] == \result);
  @*/
int max(int[] a)
....
```

There is a wide tool support for JML [4]. Just to name a few. JML Compiler translates annotations into Java code (and therefore the annotations are tested during the run-time). JMLUnit generates tests that test whether annotations are not broken. We will mention some other tools in the following sections. For more information on JML see the JML web-site [http://www.cs.iastate.edu/ leavens/JML/].

6.2 PVS (SRI)

PVS is a powerful proof assistant, developed by SRI widely used by NASA. Its primary purpose is to to reason about specifications. It is freely available for research purposes and therefore it has been used in many other applications. For example the tool JACK[2] can generate proof obligations in PVS language from Java programs annotated by JML.

The PVS specification language is based on higher-order logic. The language is illustrated by the following example.

Example 6.2.1 *An example from a tutorial on PVS [9]. The example consists of a theory describing a phone book. It also contains two conjectures. Conjectures are used to express properties that the author believes should hold.*

[2]See `http://www-sop.inria.fr/everest/soft/Jack/core.html`

One of the conjectures in this example is not valid which exposes deficiency in the specification.

```
phone_1: THEORY
BEGIN
N: TYPE            % names
P: TYPE            % phone numbers
B: TYPE = [N -> P] % phone books

% a constant of type phone number denoting the "empty" phone number
n0: P

% a constant of type phone book denoting the empty phone book
emptybook: B

% an axiom that expresses the fact that everyone
% in the emptybook has the empty phone number
emtyax: AXIOM FORALL (nm: N) : emptybook(nm) = n0

% axiom for a look-up by name function
FindPhone: [B, N -> P]
FindAx: AXIOM FORALL (bk: B), (nm: N) : FindPhone(bk, nm) = bk(nm)

% declaring type of a function AddPhone as a function from
% phone book, name and a phone number to a phone book
AddPhone: [B, N, P -> B]
% axiom for function that assign phone number to a name
Addax: AXIOM FORALL (bk: B), (nm: N), (pn: P):
                      AddPhone(bk, nm, pn) = bk WITH [(nm) := pn]

% a conjecture that we belive should hold
% (and it can be proven that it does hold)
FindAdd: CONJECTURE FORALL (bk: B), (nm: N), (pn: P):
  FindPhone(AddPhone(bk, nm, pn), nm) = pn

% an axiom that describes a delete by name function
DelPhone: [B, N -> B]
Delax: AXIOM FORALL (bk: B), (nm: N):
                         DelPhone(bk, nm) = bk WITH [ (nm) := n0 ]

% a conjecture that turns out to be false
DelAdd: CONJECTURE FORALL (bk: B), (nm: N), (pn: P):
  bk(nm) = n0 => DelPhone(AddPhone(bk, nm, pn), nm) = bk
END phone_1
```

Note that the phone book in the above example is represented as a function from names to phone numbers. Advantage of such a representation is that the representation does not put any demands on the implementation of the phone book (structures like hash map, array, linked list, etc. can be used). The PVS language is typed; all functions must be total and partial functions are supplemented by restriction on the domain type.

Example 6.2.2 *The example 6.2.1 can be extended with a subtype of the type phone number (denoted as P).*

```
ValidNumber: TYPE = {pn: P | pn /= n0}
```

The type $ValidNumber$ is formed by those phone numbers that are not equal to the empty number $n0$. Then, this type can be used as a domain of a function:

```
ReverseLookup : [ ValidNumber -> N ]
```

The tool is interactive and the user guides the proof by a set of commands. The user can extend the standard set of commands by so-called strategies. Briefly, a strategy describes some typical sequence of commands.

A proof is a tree of *sequents* and the proof is done when all sequents are proven.

Example 6.2.3 *An example of a sequent. The formula above the division line (the antecedent) states that we assume that $X = f(x)$. The formula bellow the division line (the consequent) is what we want to prove from the antecedent:*

```
{-1}  X = f(X)
  |-------
{1}   f(f(f(X))) = X
```

Mostly, core of the proofs are decision procedures. The above example can be proven by the command "(assert)", which is a command that select the appropriate decision procedure(s) and simplifies and/or proves the current sequent.

6.3 ESCJava

ESCJava [14] is a project started at the Compaq Systems Research Center[3] and is continuing as ESCJava/2 at KindSoftware[4].

[3]See `http://research.compaq.com/SRC/esc/`.

[4]See `http://www.kindsoftware.com/products/opensource/ESCJava2/`.

The aim of this project is to **automatically**, at compile time, confront Java programs with JML annotations. In order to maintain full automaticity the tool is neither sound nor complete. That is, the tool may report an error where there isn't one and it might not find an error where there is one.

Reasoning about loops can be done in two different ways. Either the loop is unrolled n times (where n is a parameter of the tool) or the loop can be treated according to the while rule (see definition 4.3.6). When the while rule is used, loops have to be annotated by loop invariants.

The underlaying prover is Simplify [15]. Integers are modeled as generic integers, which is one of the sources of unsoundness.

6.4 BLAST (Berkley)

BLAST is a tool based on the predicate abstraction with counter-example refinement. The input is an ANSI-C program where the safety property is that the label "ERROR" is not reachable.

The efficiency issues are targeted by a technique called *lazy abstraction* [24]. In lazy abstraction the phases of the counter-example refinement loop are integrated and the abstraction is build "on demand". The tool is fully automatic[5]. The abstract transition relation is over-approximated with cartesian abstraction. The Simplify prover is used to build the transition abstraction. Again, integers are modeled as generic integers and the tool is not sound.

6.5 SATABS (Carneggie Mellon)

SATABS is another tool based on the predicate abstraction with counter-example refinement. Again, the input is an ANSI-C program. The tool is trying to prove/disprove that assertion hold, null pointers are not dereferenced and array bounds are not exceeded. A boolean program[6] is build which is then model checked. To build the abstracted relation, instead of proving, a SAT solver is used [5]. Integers are modeled as bitvectors[7].

[5]Nevertheless, users can add predicates if needed.

[6]Informally, a boolean program is such a program where variables are only of type boolean. Moreover, recursion is not allowed. Therefore a boolean program is always finite and can be model checked.

[7]It can be chosen between 32-bit or 16-bit representation of integer.

6.6 UNO (Bell Labs)

UNO is a small and fast tool that is meant to intercept the most common types of errors in ANSI-C programs. These are:

1. uninitialized variables,
2. nil-pointer dereferencing and
3. out-of-bound array indexing

The tool is capable of intercepting errors of static nature.

Example 6.6.1 *On the following program UNO report out-of-boud array access:*

```
main ()
{
  int i;
  int a[10];

  for (i = 0; i < 12; i++)
  {
    a[i] = 5;
  }
```

Nevertheless, many errors remain undetected.

Example 6.6.2 *After slight modification of the program from the example 6.6.1, UNO fails to report the error:*

```
main ()
{
  int i;
  int a[10];

  int n = 100;
  for (i = 0; i < n; i++)
  {
    a[i] = 5;
  }
}
```

6.7 Comparing Examples

In this section we will compare ESCJava/2, SATABS and BLAST on a small set of examples. All three tools are fully automatic, nevertheless they cannot be compared in a completely straight-forward manner.

The examples are presented here in pseudo-code. Each example was modified for each tool. The "ERROR" in pseudo-code for BLAST was translated as "`ERROR: goto ERROR`", for SATABS as "`assert(0)`" and for ESCJAVA/2 as "`//@ assert(false)`".

Fot the test containing a loop, ESCJava/2 was run with the "loopSafe" parameter and the loop invariant was provided.

The examples were all tested on 686 Intel(R) Pentium(R) 4 CPU 2.60GHz GenuineIntel GNU/Linux.

6.7.1 Test 1

a: integer;
if $odd(a)$ **then**
  $a \leftarrow a + 1$
end if
if $odd(a)$ **then**
  ERROR
end if

- This example was classified as erroneous by the BLAST tool, i.e. it returned a false positive.
- The SATABS tool classified this program as safe (ERROR is not reachable even in the case of integer overflow).
- ESCJava/2 didn't produce any warnings.

6.7.2 Test 2

res, a, b, c: integer;
if $a > b$ **then**
  if $a > c$ **then**
    $res \leftarrow a$;
  else
    $res \leftarrow c$;
  end if
else

if $b > c$ **then**
$res \leftarrow b$;
else
$res \leftarrow c$;
end if
end if
if $\neg(res \geq a \wedge res \geq b \wedge res \geq c)$ **then**
ERROR
end if
if $\neg(res = a \wedge res = b \wedge res = c)$ **then**
ERROR
end if

- The BLAST system verified the program in 0.4 seconds. The following set of predicates was discovered:

$$\{a = res, a \leq c, a \leq res, a \leq b, b = res, c = res, b \leq c, b \leq res\}$$

- The SATABS system verified the program in 3867 seconds, i.e. more than 1 hour.

$$\{b \geq a, c \geq a, c \geq b, res \geq a, res \geq b, res \geq c,$$
$$a = b, a = c, res = a, res = b, res = c\}$$

- ESCJava/2 didn't produce any warnings.

6.7.3 Test 3

```
1: i, N: integer;
2: i ← 0;
3: if N ≥ 0 then
4:   while i ≠ N do
5:     if i > N then
6:       ERROR
7:     else
8:       i ← i + 1
9:     end if
10:  end while
11: end if
```

- The BLAST tool classified this program as safe. The set of the discovered predicates was:

$$\{0 \leq n, i = 0, i = n, n = 0, i \leq 1, i \leq n, n \leq i - 1\}$$

- The SATABS tool classified this program as safe. It discovered the following list of predicates:

$$\{i = n, n \geq 0, n \geq i\}$$

6.7.4 Test 3.1

When $i > N$ in line 4 in the program from test 3 is changed to $i \geq N$:

```
 1: i, N: integer;
 2: i ← 0;
 3: if N ≥ 0 then
 4:   while i ≠ N do
 5:     if i ≥ N then
 6:       ERROR
 7:     else
 8:       i ← i + 1
 9:     end if
10:   end while
11: end if
```

- BLAST fails to verify the program. It reports that it wasn't able to find new predicates.
- SATABS verifies the program and returns the following set of predicates:

$$\{i = n, 0 \geq n, i \geq n, n \geq 0\}$$

ESCJava/2 reports no errors in both tests 3 and 3.1 if the loop is provided with the loop invariant $i \leq N$.

6.7.5 Test 4

The last test demonstrates problems with integer overflow. The following program is not safe when the integer's domain is bounded (which is the case for ANSI-C and Java).

```
1: j, i: integer;
2: j ← i + 1;
3: if j < i then
4:   ERROR
5: end if
```

Both BLAST and ESCJava/2 do classify this example as safe; only SATABS detect the error and returns a counter-example.

6.7.6 Summary of Tests

The table 6.7.6 summarizes the test results.

Tool/Test	**Test 1**	**Test 2**	**Test 3**	**Test 3.1**	**Test 4**
BLAST	False positive	Safe	Safe	No result	Safe*
SATABS	Safe	Safe	Safe	Safe	Unsafe
ESCJava/2	Safe	Safe	Safe (li)	Safe (li)	Safe*

Table 6.1: Summary of tests

"li" denotes that a loop invariant was provided.
"Safe*" denotes that the result is not sound, i.e. that the tool has classified the program as safe even though it wasn't.

The tests has shown that predicate abstraction with counter-example refinement is capable of discovering non-trivial predicates. Nevertheless time requirements are rather high. Using a SAT solver has shown to be more accurate. Nevertheless, as test 2 has shown, the time requirements are very difficult to predict and can be very high.

The test 3.1 is an interesting one. To verify the loop, the predicate $i > N$ (or $\neg(i \leq N)$) is needed. Note that $i \leq N$ is a loop invariant for the loop. This predicate was present directly in the source code in the test 3. By replacing $i > N$ with $i \geq N$ we have "stolen" the clue for the refinement algorithm. Note that $i < N$ (or $\neg(i \geq N)$) is **not** a loop invariant of the loop, even though it holds at each entry into the loop (but not when the loop terminates). Obviously, the predicate $i > N$ can be supplemented by the combination $i \geq N \wedge \neg(i = N)$ (both predicates are present in the source code). Most likely, this combination is lost by the over-approximation that BLAST uses.

Chapter 7

Benefits and Drawbacks of the Techniques

7.1 Benefits of Annotated Code

If we use the framework thoroughly we are guaranteed that the program behaves as it was specified by the annotations.

If we follow design by contract we gain these main advantages:

- It is clear whom to blame for bugs. It is either the client who failed to establish the preconditions required by the supplier or the supplier has failed to deliver what was specified by the contract.
- Efficiency. For the same reasons as mentioned above, duplicated code can be avoided[1]. This effect can be strengthened by automated theorem proving that provides us with a guarantee, if it succeeds, that the duplicated checks are unnecessary.
- Documentation of the source code. The precondition, postcondition pair serves as a documentation of the (sub)program[2]. If this is done well, then in most cases the user (client) of the code doesn't have to fully understand the inner details of the code. Which is extremely useful in code reuse and team work.

[1] In some sense opposite approach is *defensive programming* where the code is robust as possible.

[2] Potentially, also object invariants and other constructs.

7.2 Problems with Annotations

One of the main problem of design by contract is *underspecification*. That is, when the contract does not entirely capture capabilities of the supplier. The problem is illustrated by the following example.

Example 7.2.1 *Example of an underspecified function.*

```
//@ ensures \result == a || \result == b;
int max(int a, int b)
{
 if (a > b)
   return a;
 else
   return b;
}

//@ ensures (\result == a || \result == b || \result == c) &&
//@         (\result >= a || \result >= b || \result >= c);
int max3(int a, int b, int c)
{
 int d = max(a, b);
 return max(d, c);
}
```

In the example 7.2.1 the specification of the function *max* is correct, i.e. the implementation obeys the contract. Nevertheless, the specification of *max* is not sufficient to prove correctness of the function *max*3. Obviously, the problem of underspecification sets high demands on the level of expressiveness of the contract language. Moreover, some properties are very hard to specify.

The original idea was that the programmer provides the annotation **before** actually writing the code and that program development would be **driven** by the annotations. After 30 years we must note that this hasn't become part of the general routine. There exists a bunch of opinions why this is so. Here are some of the major ones:

- Not everyone who writes software is familiar with mathematical logic.
- Writing annotations is tedious, usually the logic text is of the same size as the code.
- Writing annotations and reasoning is difficult. Again, additional skills are required. Especially, providing loop invariants can be difficult.

- Writing specifications is difficult in principle.

The problems connected to the loop invariant deserve a special care. Often, it is seen as one of the major obstacles to program annotation. As we have noted above, postconditions and preconditions serve also as documentation of a (sub)program. The loop invariant does not have this additional value, i.e. it is relevant only to the body of the procedure. The argument, that the loop invariant can serve as a "guide" to writing loops, is somewhat disputable. The problem is illustrated by the following example.

Example 7.2.2 *An algorithm for natural numbers division:*

N, Quotient, Remainder, Divisor: NaturalNumber;
Quotient $\leftarrow 0$*;*
Remainder $\leftarrow N$*;*
while *Remainder* $>$ *Divisor* ***do***
  Quotient $\leftarrow$ *Quotient* $+ 1$*;*
  Remainder $\leftarrow$ *Remainder* $-$ *Divisor;*
end while

The idea in the programmer's head is: "The division computes how many times I can fit the *Divisor* in N. So let's decrease N by *Divisor* until I can". The loop invariant "$N = Quotient * Divisor + Remainder$" looks more as a consequence and is not directly derivable from the idea[3].

[3]Nevertheless, the loop invariant can be a good guide for optimizations.

Chapter 8

Suggestions

8.1 Warnings for Specification Language

Compilers of programming languages provide a programmer with *warnings*. Warnings mark constructs that are not necessarily wrong but are in some sense odd (depreciated, unusual, etc.). Such a mechanism should be available for the specification languages too.

Example 8.1.1 *Out of bound array access:*

```
//@ ensures (\forall int i1; i1 >= 0 & i1 <= a.length; a[i1] == 1);
```

Example 8.1.2 *Null pointer dereference:*

```
//@ requires a.length > 0;
```

Instead we would expect:

```
//@ requires a != null && a.length > 0;
```

Example 8.1.3 *Trivially satisfiable statements*

```
/*@ requires (\exists int v, vi; vi >= 0 && vi < a.length;
@                                              a[vi] == v);
@*/
```

Such a specification is probably not correct, the preconditions is satisfiable whenever $a.length > 0$.

8.2 Extend Compiler Warnings

Since the verification tools already contain an automated prover, the prover should be employed to detect suspicious constructs.

Example 8.2.1 *Detection of infinite loops.*
In general, for a while loop of the form

while B **do** S **end while**

we are looking for such an invariant I that

$$\{I \wedge B\}\ S\ \{I \wedge B\}$$

Example of such a loop:

i: NaturalNumber;
$i \leftarrow 2$*;*
while *even*(i) ***do***
  $i \leftarrow i * 2$
end while

Example 8.2.2 *Detection of expressions (namely boolean expressions) that always evaluate to the same value.*
For example:

```
...some code...
   t = 1;
   if ((t - 1) < 2)
      ....some code...
```

Annotations could be employed in such a reasoning:

```
...some code...
   //@ invariant i > 0 && i < n;
   while (...) do
   {
      ....some code...
      if (i = n)
      //i = n evaluates to false if the loop invariant holds
      ....some code...
   }
```

8.3 User Interface Extensions

It should be possible for a user to invoke automated loop invariant guessing that would add the loop invariant directly into the source code. Moreover, it should be possible to chose from different loop invariant guessing algorithms. For example, predicate abstraction can be used for invariant guessing. Nevertheless, some frequent invariants can be derived simply from the syntactic information.

Example 8.3.1 *One of the most common patterns of a loop:*

```
//@ invariant i >= 0 & i <= 10;
for (int i = 0; i < 10; i ++)
{
   ... some code that does not affect i ...
}
```

8.4 Automation

Nowadays, there is a great amount of SW applications that are not very complex. On the contrary, similar tasks are repeated over and over again.

A good amount of SW applications rely on large and powerful libraries. Because these libraries are being used by so many programmers their correctness is extremely important. Full verification of complicated algorithms is a difficult task. Nevertheless, it can be done (see e.g. [19]). Moreover, the amount of work needed to verify these libraries per user is very little.

Semi-automated tools, exploiting annotated and verified libraries, could be used for program development. A similar (in a smaller scale) approach was already used (e.g. the Amphion system[1]).

[1] See `http://ti.arc.nasa.gov/ase/synpub.html` or [38].

Chapter 9

Summary

In this Thesis we have addressed one particular set of techniques that can help in software development – formal methods. Specifically, we have focused on the approaches where automated theorem proving is exploited. In the first part of the Thesis we have introduced two main techniques of program verification – program annotation and predicate abstraction. On various examples, we have analyzed the benefits these techniques convey and also pointed out some of their weak points.

The two techniques introduced in the first part of the Thesis were chosen deliberately. The applicability of these techniques has been proven by various implementations which is reflected by the second part where we have presented a set of tools that are based upon these techniques[1]. Furthermore, we have devised a set of examples on which we studied capabilities and drawbacks of these tools and differences between them. The Thesis was closed by various suggestions on specification languages design and tool support.

Software development has many aspects – economical, scientific or even sociological. For successful employment of formal methods, all of these aspects should be taken into account. In this Thesis we have demonstrated that there is a vast unused potential in formal methods, specifically in:

1. Automated reasoning,
2. Tool combination and
3. General user comfort

We have shown that automated reasoning plays a crucial role in software verification. The automated theorem provers battle with a delicate balance between

[1]It should be noted that all the presented tools are freely available (at least for research purposes), i.e. we have not dealt with commercial tools.

computational complexity and the range of proof tasks that the prover can target. We have presented two different tools (BLAST, ESCJava) that both rely on the same theorem prover (Simplify theorem prover). Even though Simplify is nowadays one of the best automatic theorem provers for SW verification we have shown it suffers from serious problems. Therefore it is obvious that there still is a great potential in adjusting automated theorem proving to SW verification.

We have demonstrated that nowadays tools are capable of automatic handling of non-trivial tasks. Each tool can help a software developer in solving things of various nature. Each tool conveys results of different type. These results vary in reliability and precision and how much effort is needed to be invested by the user to obtain the result. We should note, that there is a lack of combinations of these tools.

It should not be neglected that some techniques and approaches may be rejected simply because developers don't like to use them for no particular rational reasons, i.e. the reasons might be more of a psychological nature. Moreover, the software industry is driven, as any other industry, mainly by economical results. From managerial point of view the application of formal methods may seem slow and inefficient. Therefore, formal methods should be made more easy to use and employ. Users should be allowed to chose from tools and their combinations according to their needs; the tools should reflect the fact that the users are generally programmers, not mathematicians.

9.1 Future Work

The suggestion in the end of this Thesis should be enhanced[2]. Naturally, it would be interesting to implement some of the suggestion – as for example tool combinations – and perform more tests. Similar tests could also be done for other types of tools[3] .

The suggestions proposed in this Thesis should also be integrated in other formal methods research[4].

[2]For example, more types of warnings for specification languages should be explored and these should be provided with severity level.

[3]For example the tool PREfix [30].

[4]For general proposals on future research see e.g. [26, 40].

Bibliography

[1] K. R. Apt and E.-R. Olderog. *Verification of sequential and concurrent programs*. Springer-Verlag, 1991.

[2] Tom Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. *International Journal on Software Tools for Technology Transfer (STTT), Special Section on TACAS'01*, 2003. To appear.

[3] Kent Beck. *eXtreme Programming eXplained, Embrace Change*. Addison Wesley, 2000.

[4] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 2004. To appear.

[5] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based Predicate Abstraction for ANSI-C . In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005.

[6] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[8] P. Cousot and R. Cousot. Static analysis of embedded software: Problems and perspectives, invited paper. In T.A. Henzinger and C.M. Kirsch, editors, *Proc.* First Int. Workshop on Embedded Software, EMSOFT 2001, volume 2211 of *Lecture Notes in Computer Science*, pages 97–113. Springer, 2001.

[9] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, , and Mandayam Srivas. A tutorial introduction to PVS. April 1995.

[10] Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, 2001. June 2001, Boston, USA.

[11] Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design*. Springer-Verlag, November 2002.

[12] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *11th International Conference on Computer-Aided Verification*. Springer-Verlag, July 1999. Trento, Italy.

[13] E. Denney, B. Fischer, and J. Schumann. Using Automated Theorem Provers to Certify Auto-generated Aerospace Software. In M. Rusinowitch and D. Basin, editors, *Proceedings of the 2nd International Joint Conference on Automated Reasoning*, number 3097 in Lecture Notes in Artificial Intelligence, pages 198–212, 2004.

[14] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report #159, Palo Alto, USA, 1998.

[15] David L. Detlefs, Greg Nelson, and James B. Saxe. Simplify: A Theorem Prover for Program Checking. Technical report, Palo Alto, USA, July 2003. Available at http://research.sun.com/people/detlefs/bib.html.

[16] Edsger W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, October 1972.

[17] Gerald L. Dillingham. Role of FAA's Medernization Program in Reducing Delays and Congestion.

[18] Bruce Eckel. *Thinking in Java 2*. Prentice-Hall, 2000.

[19] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université Paris-Sud, July 1999.

[20] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and N. Shankar. ICS: integrated canonizer and solver. To be presented at CAV'2001, 2001.

[21] M. J. Fisher and M. O. Rabin. Superexponential complexity of presburger's arithmetic. *SIAM-AMS Proceedings*, 7:27–41, 1974.

[22] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 191–202, New York, NY, USA, 2002. ACM Press.

[23] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[24] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages (POPL), ACM Press, 2002, pp. 58-70*, 2002.

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[26] Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[27] Heinrich Hußmann. *Formal Foundations for Software Engineering Methods*, volume 1322 of *LNCS*. Springer-Verlag, 1997.

[28] ISO/IEC, editor. *Programming Language C*. ISO/IEC, 1999.

[29] Bell Labs. Basic Spin Manual. Available at http://cm.bell-labs.com/cm/cs/what/spin/Man/Manual.html.

[30] C. H. Levy, Luiz Henrique de Figueiredo, Marcelo Gattass, Carlos Jose Pereira de Lucena, and Donald D. Cowan. IUP/LED: A Portable User Interface Development Tool. *SP&E*, 1996.

[31] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis*. North-Holland Publishing Co., Amsterdam, 1978.

[32] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[33] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS*. Springer-Verlag, 2004.

[34] Nimal Nissanke. *Formal Specification: Techniques and Applications*. Springer, 1999.

[35] P. Odifreddi. *Classical Recursion Theory*. North-Holland, Amsterdam, 1989.

[36] J. Paris and L. Harrington. A mathematical incompleteness in peano arithmetic. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 1133–1142. North-Holland, Amsterdam, 1977.

[37] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[38] Johann M. P. Schumann. *Automated Theorem Proving in Software Engineering*. Springer, 2000.

[39] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 132–143, New York, NY, USA, 1977. ACM Press.

[40] The British Computer Society. Grand Challenges in Computing - Research. Edited by Tony Hoare and Robin Milner, available at http://www.ukcrc.org.uk/.