FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Methods and Tools for the Formal Verification of Software

An Analysis and Comparison

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Marian Rainer-Harbach
Matrikelnummer 0325724

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Gernot Salzer

Wien, 20.11.2011

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Methods and Tools for the Formal Verification of Software

An Analysis and Comparison

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Medical Informatics

by

Marian Rainer-Harbach
Registration Number 0325724

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Gernot Salzer

Vienna, 20-11-2011

Erklärung zur Verfassung der Arbeit

Marian Rainer-Harbach
Otto-Bondy-Platz 1/6.04, 1120 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20.11.2011 _____

Acknowledgments

I would like to thank my parents Moni and Ulli for their extensive moral and financial support in all of my endeavors. I am very grateful to Elena and Kitty for proofreading, suggestions and a fun summer full of impossible Polaroid photos.

My advisor Gernot Salzer provided remarkably kind and professional assistance throughout the work on this thesis. Thank you!

Finally, I would like to thank the authors of some of the tools analyzed in this thesis: David Crocker offered help and comments regarding Escher C Verifier. Daniel Bruns answered my questions on the KeY mailing list. Forum members at the Codeplex site supported the implementation of the examples for VCC. VeriFast developer Bart Jacobs provided extensive and friendly help for his tool.

Abstract

The task of *proving* the correctness of software (formal verification) has been a research topic for many years. Despite that, formal methods still have not been widely adopted in practical areas. A key reason for this has been the lack of accessible yet powerful tools that are able to efficiently support the software engineer in this complex exercise. In the last few years, a new generation of tools has appeared: They are aimed at the verification of programs written in programming languages such as C or Java and claim to be usable by software engineers without education in formal methods.

This thesis gives an overview of some theoretical aspects of formal verification. A number of tools is extensively described, and some of them are selected to compete in a practical comparison. The comparison is based on tasks that are commonly encountered in software development. Some general thoughts on requirements for formal verification tools in industry and teaching are also given. The tools analyzed are CBMC, Escher C Verifier, Frama-C/Jessie, Frege Program Prover, KeY, Perfect Developer, Prototype Verification System, VCC and VeriFast.

Zusammenfassung

Die Aufgabenstellung, die Korrektheit von Software zu *beweisen* (Formale Verifikation), ist seit vielen Jahren Forschungsthema. Trotzdem ist die Nutzung von formalen Methoden in der Praxis noch nicht weit verbreitet. Ein zentraler Grund dafür ist der Mangel an leicht zugänglichen, aber trotzdem leistungsstarken Tools, die Softwareentwickler bei dieser komplexen Aufgabe unterstützen. In den letzten Jahren ist eine neue Generation von Tools erschienen: Sie zielen auf die Verifikation von Programmen ab, die in Programmiersprachen wie C und Java geschrieben sind und erheben den Anspruch, auch für Softwareentwickler ohne Ausbildung in formalen Methoden verwendbar zu sein.

Diese Arbeit gibt einen Überblick über einige theoretische Aspekte von formaler Verifikation. Mehrere Tools werden ausführlich beschrieben. Einige davon werden ausgewählt, um ihre Fähigkeiten in einem praktischen Vergleich zu beweisen. Dazu werden Beispiele verwendet, die auf häufig vorkommenden Problemstellungen basieren. Einige allgemeine Anforderungen an Verifikationstools in Industrie und Lehre werden ebenfalls diskutiert. Die in dieser Arbeit analysierten Tools sind CBMC, Escher C Verifier, Frama-C/Jessie, Frege Program Prover, KeY, Perfect Developer, Prototype Verification System, VCC und VeriFast.

Contents

1 Introduction **11**

2 Theoretical background **15**
2.1 Natural deduction 15
2.2 Hoare logic 20
2.3 Weakest preconditions 22
2.4 Dynamic logic 23
2.5 Symbolic execution 24
2.6 Model checking 25

3 Tools **29**
3.1 Not selected for comparison 30
3.1.1 Frama-C/Jessie 30
3.1.2 Frege Program Prover 34
3.1.3 Perfect Developer 34
3.1.4 Prototype Verification System 35
3.1.5 CBMC 36
3.2 Selected for comparison 41
3.2.1 Escher C Verifier 41
3.2.2 KeY 45
3.2.3 VCC 50
3.2.4 VeriFast 54

4 Comparison **59**
4.1 Escher C Verifier 60
4.2 KeY 69
4.3 VCC 79
4.4 VeriFast 88
4.5 Summary 97

5 Thoughts on requirements **105**

5.1 Industrial applications 106
5.2 Teaching 108

6 Conclusion **111**

A VeriFast: Factorial **115**

Bibliography **119**

CHAPTER 1

Introduction

Software is an important factor to almost all areas of life. Today, dependency on software is still increasing, especially in critical domains, where damage to human health, impact on the environment or vast economic effects could be caused by malfunctioning software and systems. Critical domains include medical applications (diagnostic and therapeutic devices), transportation (spacecrafts, aircrafts or trains) and industrial applications like power plants. The ubiquitous presence of software leads to the requirement of being able to *prove* its correctness.

Dynamic software tests, which verify programs by executing them, supplying them with various input data and analyzing the results, have been universally employed for decades, both in scientific research as well as in commercial environments. There exist numerous approaches for testing different aspects of software and tools for almost every development process, programming language and platform. The common problem with dynamic testing is that, even when structured procedures are used, it can never be proved that the code is completely correct, meaning it does exactly, in all situations, what is required by the specification. The only result that can be obtained is that under the used procedures in a certain environment, with certain input data, no more errors can be found. While this result may well be sufficient in some situations, for applications such as those outlined above a formal proof that the produced code is absolutely correct is desirable or even strictly necessary because of ethical, legal or economic considerations.

Formal verification, in contrast to dynamic testing, strives to accomplish this. Using a formalisation of the natural-language specification, it can be *proved* that software is correct. The downside is that formal verification is a complex task, both for humans to understand as well as computationally.

While techniques for formal verification have been the subject of scientific research for a long time, only few tools are available to support their use in settings outside the academic field and their usage in large or commercial projects is rather uncommon.

In contrast, formal verification is quite popular in hardware design processes, especially through the use of model checking. This is due to the high cost of correcting any design faults and due to the fact that most hardware designs can be more easily verified from a computational standpoint than software.

In the last few years, a new generation of tools supporting formal verification of software has appeared. In most cases, earlier tools were suitable only for use in research or for demonstration purposes. They only supported small fractions of the features of general-purpose programming languages and required excellent knowledge of the tool and formal methods in general. The new generation of tools aims to make formal verification of software accessible to a larger audience. They support the verification of programs written in substantial subsets of languages such as C or Java. In many cases they are able to automatically prove correctness, abolishing the need for the user to become an expert in formal methods in order to manually construct proofs. The developers of some tools even specifically address software engineers who have no training in formal methods and claim that their products are just as useful for this target group as they are for experts.

In 2004 four of the then current tools were evaluated by Feinerer et. al. [37; 38]. They concluded that only one tool, Perfect Developer, was suitable for general-purpose use by persons with little background knowledge in formal methods. The main impulse to start work on this thesis was that the appearance of the new generation of tools has improved the potential of formal verification considerably. In addition, a long time has passed since 2004 and no comprehensive analysis or comparison of current tools has been performed since then.

The aim is to provide an overview on today's state of formal verification of software on multiple levels and to analyze whether the claims made by the tools' developers are accurate. We will start by looking at the theoretical background of formal verification in chapter 2, for example by explaining the methods of natural deduction, Hoare logic and model checking. Many of the more powerful existing tools for the application of these methods are researched, analyzed and compared in chapter 3. The tools chosen for this analysis are CBMC, Escher C Verifier, Frama-C/Jessie, Frege Program Prover, KeY, Perfect Developer, Prototype Verification System, VCC and VeriFast. They are categorized based on theoretical background and by the developers' claims regarding their suitability for different programming

problems and environments, for example certain programming languages and platforms.

Those tools that show the most promising approaches are selected to compete in a practical comparison in chapter 4. Examples for different classes of programming problems are created. They are used to evaluate the claims the tool's developers made and the adequacy of the tools for specific target groups. Special attention is paid to examine possibilities and hindrances regarding the tools' practical use and to good approaches that could increase the adoption of formal verification outside of academic research. Finally, chapter 5 summarizes important properties tools should possess to be suitable for industrial or educational environments.

CHAPTER 2

Theoretical background

This chapter aims to give an overview of some approaches often used for the task of formal verification. The methods described are all used directly or in modified form in the tools that will be compared in section 3. As many approaches are quite complex and a vast amount of publications exists, just a small fraction of the aspects of each approach can be illustrated here. For further information, references to seminal and current publications will be given in each section.

The following descriptions assume the reader to have a basic knowledge of propositional and first-order predicate logic.

2.1 Natural deduction

For being able to reason about specifications of programs and systems some sort of formal calculus is needed. Such a calculus needs to provide rules that allow to draw conclusions from formulas.

Natural deduction is the basic calculus for reasoning about formulas in propositional logic. It has been described independently by Gerhard Gentzen [43] and Stanisław Jaśkowski [55] in 1934. The following description is based on [47, pp. 5 sqq.], where many additional explanations are given. We start with a set of formulas $\phi_1, \ldots, \phi_n$ and a formula ψ, which are called premises and conclusion, respectively. Deduction consists in applying rules to premises to obtain new formulas (consequences), and to obtain the conclusion eventually. This goal is expressed by the *sequent* $\phi_1, \ldots, \phi_n \vdash \psi$. If this process succeeds, the sequent is *valid*. In non-trivial formulas more than one rule can be applied in most cases. One possible approach for

constructing proofs is given in [47, p. 28]: The authors recommend working both from the premises and from the conclusion towards each other. The $\rightarrow i$ and $\neg i$ rules (which will be explained below) are said to most often improve the situation when applying them backwards to the conclusion, as they yield an extra premise and simplify the conclusion.

We will now look at the rules that natural deduction provides for transforming propositional logic formulas. In the rules, the premises are stated above the line and the conclusion that can be obtained below the line. For the rules dealing with logical connectives, two versions (for introduction i and elimination e of the respective connective) are given.

Conjunction $\dfrac{\phi \qquad \psi}{\phi \wedge \psi} \wedge i$ $\qquad$ $\dfrac{\phi \wedge \psi}{\phi} \wedge e_1$ $\qquad$ $\dfrac{\phi \wedge \psi}{\psi} \wedge e_2$

The rules for conjunction state that, if both ϕ and ψ have already been concluded, $\phi \wedge \psi$ may be concluded ($\wedge i$), and that, if $\phi \wedge \psi$ has already been concluded, ϕ as well as ψ may be concluded ($\wedge e_1$, $\wedge e_2$).

Disjunction $\dfrac{\phi}{\phi \vee \psi} \vee i_1$ $\qquad$ $\dfrac{\psi}{\phi \vee \psi} \vee i_2$ $\qquad$ $\dfrac{\phi \vee \psi \quad \boxed{\begin{array}{c}\phi \\ \vdots \\ \chi\end{array}} \quad \boxed{\begin{array}{c}\psi \\ \vdots \\ \chi\end{array}}}{\chi} \vee e$

The introduction rules for disjunction are easy to understand, as they just express that if a formula has been concluded, it is possible to conclude the disjunction with another formula as well (i_1, i_2). The elimination rule $\vee e$ is a bit different: To show that a formula χ is true starting from the premise $\phi \vee \psi$, it has to be shown that from both ϕ and ψ, χ can be concluded. This is necessary as it is unknown which of the subformulas in the premise is true. The contents of the boxes are thus subproofs, in which it is assumed that ϕ and ψ are true, respectively. From this assumption, χ needs to be concluded.

Bottom $\dfrac{\bot}{\phi} \bot e$

The rule for bottom elimination expresses the fact that from a contradiction (denoted by $\bot$), any formula can be derived. Informally, a contradiction, for example $\psi \wedge \neg\psi$, can be seen as a formula that carries more information than any other formula in the calculus could ever carry. Thus, any other formula can be concluded from a contradiction.

Negation $\dfrac{\begin{array}{|c|}\hline \phi \\ \vdots \\ \bot \\ \hline \end{array}}{\neg\phi}\ \neg i$ $\qquad$ $\dfrac{\phi \qquad \neg\phi}{\bot}\ \neg e$

To use $\neg i$, a subproof has to be performed in which it is shown that ϕ leads to a contradiction $\bot$. Then, $\neg\phi$ may be concluded. The elimination rule $\neg e$ defines that $\bot$ is the symbol for a contradiction.

Double negation $\dfrac{\phi}{\neg\neg\phi}\ \neg\neg i$ $\qquad$ $\dfrac{\neg\neg\phi}{\phi}\ \neg\neg e$

These rules express that in classical logic a formula and its double negation are equivalent.

Implication $\dfrac{\begin{array}{|c|}\hline \phi \\ \vdots \\ \psi \\ \hline \end{array}}{\phi \to \psi}\ \to i$ $\qquad$ $\dfrac{\phi \qquad \phi \to \psi}{\psi}\ \to e$ *(modus ponens)*

Introducing an implication is performed by conducting a subproof, in which it needs to be shown that, under the assumption that ϕ is true, ψ can be concluded. Then, $\phi \to \psi$ may be concluded. The elimination rule for implications, also called modus ponens, states that, given both ϕ and $\phi \to \psi$, ψ may be concluded.

Beside the necessary basic rules, additional *derived rules* can be defined. These rules are not strictly required for performing proofs, but can make proofs easier by combining applications of several basic rules into a single derived rule. We will look at three widely-known derived rules called *modus tollens*, *proof by contradiction* and *law of excluded middle*. Strictly speaking, the $\neg\neg i$ rule is a derived rule, too, as it can be constructed by applying $\neg i$ and $\neg e$ to ϕ.

Derived rules $\dfrac{\phi \to \psi \qquad \neg\psi}{\neg\phi}$ *modus tollens*

Modus tollens, similar to $\to e$, eliminates an implication. However, it deals with the implication's right-hand side and states that when $\phi \to \psi$ and $\neg\psi$ have been concluded, $\neg\phi$ can be concluded.

$$\dfrac{\boxed{\begin{array}{c}\neg\phi \\ \vdots \\ \bot\end{array}}}{\phi}$$ *proof by contradiction (reductio ad absurdum)*

A proof by contradiction for a formula ϕ is performed by assuming $\neg\phi$ in a subproof and deriving that this assumption results in $\bot$.

$$\dfrac{}{\phi \vee \neg\phi}$$ *law of excluded middle (tertium non datur)*

The law of excluded middle simply states that always either ϕ or $\neg\phi$ is true and thus no third possibility exists.

To illustrate how proofs are constructed, we will look at the example given in [47, p. 23] that incorporates various rules. The sequent to be proven is $p \wedge \neg q \to r$, $\neg r$, $p \vdash q$. To the right of the formulas it is shown which rules

1	$p \wedge \neg q \to r$	premise
2	$\neg r$	premise
3	p	premise
4	$\neg q$	assumption
5	$p \wedge \neg q$	$\wedge i$ 3, 4
6	r	$\to e$ 1, 5
7	$\bot$	$\neg e$ 6, 2
8	$\neg\neg q$	$\neg i$ 4–7
9	q	$\neg\neg e$ 8

are applied to which lines. Taking up the hint that both $\to i$ and $\neg i$ are often helpful when applying them to the conclusion, we look at which of the rules would be easier to use. Since the conclusion we want to proof is just q, we try $\neg i$, because the structure of its conclusion fits better than the conclusion of the $\to i$ rule. $\neg i$ enables us to derive $\neg\phi$ from ϕ. However, we want to conclude q and not $\neg q$. Thus, we need to add an additional negation: The $\neg\neg e$ rule applied backwards to line 9 provides line 8. Now

we can start the application of $\neg i$ by performing the subproof needed to establish $\neg i$'s premise. This is done between the horizontal bars (lines 4–7).

In the subproof, we want to go from $\neg q$ to $\bot$, which proves $\neg\neg q$. Again, it is easier to start at the bottom. To derive a contradiction, we need two formulas ϕ and $\neg\phi$. $\neg r$ is already provided in the premise, so we just need to derive r. One of the main premises states that $p \wedge \neg q \rightarrow r$, so we will try to derive r from this premise using the $\rightarrow e$ rule (line 6). For the application of this rule, we still need to show that $p \wedge \neg q$ is valid. This is done easily, as p is one of the premises and $\neg q$ is the assumption we make in the subproof. Thus, we can use rule $\wedge i$ to derive $p \wedge \neg q$ (line 5). Now, a continuous chain of proofs has been created which shows that $p \wedge \neg q \rightarrow r$, $\neg r$, $p \vdash q$. Note that instead of using the $\neg i$ rule, it would have also been possible to perform a proof by contradiction. In this case, lines 4–7 would have directly yielded q.

The calculus introduced until now was focused on propositional logic. To extend the calculus to first-order predicate logic we can use all rules defined above, but need to introduce additional rules for working with quantifiers and the equality predicate.

Equality $\qquad \dfrac{}{t = t} = i \qquad \dfrac{t_1 = t_2 \qquad \phi[t_1/x]}{\phi[t_2/x]} = e$

The rule for introducing equality simply states that any term t is equal to itself. The elimination rule allows to conclude $\phi[t_2/x]$ given the equality of t_1 and t_2 and the formula ϕ where every free occurrence of x was replaced by t_1. An occurrence of x is called *free* if it is not in the scope of some $\forall x$ or $\exists x$ and *bound* otherwise. Further information regarding this aspect can be found in [47, pp. 102 sq.].

Universal quantification $\qquad \dfrac{\boxed{\begin{array}{l} x_0 \\ \quad \vdots \\ \quad \phi[x_0/x] \end{array}}}{\forall x\, \phi} \forall x\, i \qquad \dfrac{\forall x\, \phi}{\phi[t/x]} \forall x\, e$

The $\forall x\, i$ rule looks similar to some rules defined for propositional formulas, but it works differently: In the box, a fresh variable x_0 is introduced. x_0 must not occur anywhere outside the box and is called eigenvariable or parameter. When $\phi[x_0/x]$ can be proven, it is possible to conclude $\forall x\, \phi$. This is because, as x_0 is a new variable, no assumptions were made about it. Thus, any x would work instead of x_0 and the generalization to $\forall x$ is valid. The rule for elimination states that when ϕ is true for all x (premise $\forall x\, \phi$),

it is possible to replace x by any term t (with t being free in ϕ), resulting in $\phi[t/x]$. t can be thought of "[...] as a more concrete *instance* of x" [47, p. 109].

Existential quantification

$$\frac{\phi[t/x]}{\exists x\, \phi}\ \exists x\, i \qquad \frac{\exists x\, \phi \qquad \boxed{\begin{array}{cc} x_0 & \phi[x_0/x] \\ & \vdots \\ & \chi \end{array}}}{\chi}\ \exists x\, e$$

Introducing an existential quantifier $\exists x$ to a formula ϕ is allowed when a formula $\phi[t/x]$ has already been concluded. A reason for this is that the formula in the premise contains more information than the conclusion, because in the premise a concrete value of x for which ϕ is valid occurs (t). The $\exists x\, e$ rule works by finding an x_0 that, when replacing x in ϕ, allows to conclude χ. Again, the eigenvariable x_0 must not occur outside the box.

Further information on the additional rules for first-order logic can be found in [47, pp. 107 sqq.].

2.2 Hoare logic

Hoare logic (also known as Floyd-Hoare logic) was described in 1969 by C. A. R. Hoare [46] with some ideas based on an article published by Robert W Floyd two years earlier [39]. The explanation below is based both on Hoare's paper as well as on [47, pp. 262 sqq.].

Hoare logic is based on properties of variables before and after execution of a part of a program. An important aspect is that the correctness of such a part can depend on the values of variables before execution of the part is initiated. Thus, certain requirements (assertions) for those variables can be specified. Such a specification is called *precondition*. Similarly, the program is meant to yield a result that has certain properties. Again, these properties are formalized in an assertion called *postcondition*. These assertions express that the program promises to return results that adhere to the postcondition under the requirement that, at the beginning of program execution, the precondition is satisfied. These aspects can be written formally as a *Hoare triple* $\{\phi\}P\{\psi\}$, with ϕ as the formula representing the precondition, ψ the postcondition and P the program. Pre- and postcondition are formulas in first-order logic.

A Hoare triple can be proved in regard to partial or total correctness. *Partial correctness* means that, "[...] for all states which satisfy ϕ, the state

resulting from P's execution satisfies the postcondition ψ, provided that P actually terminates" [47, p. 265]. Thus, any program that does not terminate, regardless of what it is computing, is partially correct. *Total correctness* additionally requires that the program always indeed does terminate.

Hoare logic consists of axioms and rules that are to be applied for constructing proofs.

$$\text{Assignment} \quad \frac{}{\{\psi[E/x]\}\, x = E\, \{\psi\}}$$

The assignment axiom states that in order to show that ψ holds after the execution of $x = E$, it needs to be shown that $\psi[E/x]$, i. e. ψ with every free occurrence of x replaced with E, holds before the assignment.

$$\text{Composition} \quad \frac{\{\phi\}\, C_1\, \{\eta\} \qquad \{\eta\}\, C_2\, \{\psi\}}{\{\phi\}\, C_1; C_2\, \{\psi\}}$$

The composition rule enables us to split a complex problem into two simpler problems by finding a suitable midcondition η that serves as postcondition of C_1 and as precondition of C_2.

$$\text{If statement} \quad \frac{\{\phi \wedge B\}\, C_1\, \{\psi\} \qquad \{\phi \wedge \neg B\}\, C_2\, \{\psi\}}{\{\phi\}\, \texttt{if}\ \ B\ \ \{C_1\}\ \ \texttt{else}\ \ \{C_2\}\, \{\psi\}}$$

The rule for if statements splits the proof into two triples: In the one triple it has to be shown that the program is correct in the case that B is true, which is reflected by the fact that B is added to the precondition as a conjunction. In the other triple correctness needs to be shown for the case that B is false.

$$\text{Implied} \quad \frac{\vdash_{\text{AR}} \phi' \to \phi \qquad \{\phi\}\, C\, \{\psi\} \qquad \vdash_{\text{AR}} \psi \to \psi'}{\{\phi'\}\, C\, \{\psi'\}}$$

The implied rule allows to prove $\{\phi'\}\, C\, \{\psi'\}$ in the case that $\{\phi\}\, C\, \{\psi\}$ could already be proven and that both ϕ' implies ϕ and ψ implies ψ'. Using this rule, it is possible to strengthen the pre- and weaken the postcondition. $\vdash_{\text{AR}}$ denotes that a proof for the respective sequent using the natural deduction calculus with the addition of standard arithmetic is required.

$$\text{Partial while} \quad \frac{\{\eta \wedge B\}\, C\, \{\eta\}}{\{\eta\}\, \texttt{while}\ \ B\ \ \{C\}\, \{\eta \wedge \neg B\}}$$

This rule can show partial correctness of a loop, which means it does not consider whether the loop terminates. Dealing with loops is the most complex aspect of Hoare logic and one that cannot be automated. The main feature

of the partial while rule is the *invariant* η: "In general, the body C [...] changes the values of the variables it manipulates; but the invariant expresses a relationship between those values which is preserved by any execution of C." [47, p. 273] The problem is that the rule cannot be applied directly in most cases, as normally triples such as $\{\phi\}\,\texttt{while}\ B\ \{C\}\,\{\psi\}$ need to be proven, where ϕ and ψ are not related in a way as would be required by the rule. Thus, an invariant η has to be found for which $\vdash_{\mathrm{AR}} \phi \to \eta$, $\vdash_{\mathrm{AR}} \eta \wedge \neg B \to \psi$ and $\vdash_{\mathrm{par}} \{\eta\}\,\texttt{while}\ B\ \{C\}\,\{\eta \wedge \neg B\}$ are all valid. The third sequent is proven using the partial while rule, and then, using the other two sequents with the implied rule, the original loop can be proven.

Finding loop invariants in general requires intelligence and intuition. Invariants are thus a rather problematic aspect with regard to the widespread acceptance of formal verification, as finding them not only cannot be automated, but can also be a quite complex task for the user. Automatic generation of invariants is a research topic, for example in [75].

$$\text{Total while}\quad \frac{\{\eta \wedge B \wedge 0 \leq E = E_0\}\,C\,\{\eta \wedge 0 \leq E < E_0\}}{\{\eta \wedge 0 \leq E\}\,\texttt{while}\ B\ \{C\}\,\{\eta \wedge \neg B\}}$$

This rule can show total correctness of a loop, as it uses additional information for showing that the loop terminates after a finite number of iterations. For this, an integral expression called loop *variant* is used, E. The value of the variant is decremented in each loop iteration, and, as it has a lower bound of 0, the loop terminates when E reaches this value.

Hoare logic is the foundation of many implementations for formal verification, although it is not directly implemented in most cases. In practice, other approaches, some of which are extensions of Hoare logic, are used.

2.3 Weakest preconditions

The idea of *weakest preconditions* was introduced by Edsger W. Dijkstra in 1975 together with the Guarded Command Language [26]. Weakest preconditions were inspired by Hoare logic, as they also use pre- and postconditions for describing the expected results of a program that gets supplied with certain inputs. The main difference is that in Hoare logic, for pairs of every program and every postcondition, it is possible to state infinitely many different preconditions in a way that the Hoare triple is valid. Most of the possible preconditions are much stronger than really necessary.

In contrast, $wp(S, R)$, with S being a list of commands and R the required postcondition, denotes the necessary and sufficient – and thus weakest –

precondition "[...] for the initial state of the system such that activation of S is guaranteed to lead to a properly terminating activity leaving the system in a final state satisfying the post-condition R" [26, p. 454].

Similar to Hoare logic, there exist rules to deal with certain command constructs. As most rules follow the same ideas as those in Hoare logic, we will give only the rule for the alternative construct as an example. While this rule is the analogon to the if statement rule in Hoare logic, Dijkstra describes a version suitable for dealing with an arbitrary number of guarded commands: First, two abbreviations are defined. *IF* denotes a set of guarded commands $\texttt{if}\ B_1 \rightarrow SL_1 \,\square \ldots\, \square B_n \rightarrow SL_n\ \texttt{fi}$; and *BB* denotes $(\exists i : 1 \leq i \leq n : B_i)$;. B_1 to B_n are called guards. When a guard is true, the corresponding guarded list $(SL_1, \ldots, SL_n)$ can be executed. $\square$ is a separator between guarded commands.

Then, $wp(IF, R) = (BB\ \texttt{and}\ (\forall i : 1 \leq i \leq n : B_i \Rightarrow wp(SL_i, R))$. *BB* expresses that at least one guard must be true, otherwise the program would abort. The main term requires "[...] that each guarded list eligible for execution will lead to an acceptable final state" [26, p. 455].

Further rules exist for other elements such as assignments and loops. Loops are handled in a way very similar to Hoare logic by requiring loop invariants and variants.

The weakest preconditions calculus is implemented in several tools that are analyzed in section 3.

2.4 Dynamic logic

Dynamic logic (DL) was described in 1984 by David Harel. An updated version of his book was published in 2000 [45]. DL can be seen as an extension of Hoare logic [4, p. 70]. A form of dynamic logic is constructed by extending a non-dynamic logic with modal operators associated to actions. The necessity (or box) operator [], for example used in $[p]\psi$ expresses that *if* the computation of p terminates, then ψ holds. The possibility (or diamond) operator $\langle\rangle$, for example used in $\langle p\rangle\psi$ expresses that the computation of p *will* terminate and that afterwards ψ will hold.

Formulas in dynamic logic are of the form $\phi \rightarrow [p]\psi$ or $\phi \rightarrow \langle p\rangle\psi$, which is similar to a Hoare triple $\{\phi\}\, p\, \{\psi\}$ for partial or total correctness, respectively. The extension in dynamic logic is that "[i]n Hoare logic, the formulae ϕ and ψ are pure first-order formulae, whereas in DL they can contain programs" [4, p. 70]. This provides greater expressiveness, as some conditions cannot be stated using just first-order logic.

KeY (see section 3.2.2 on page 45) uses a custom dynamic logic called Java Card DL and performs symbolic execution for deduction.

2.5 Symbolic execution

Symbolic execution deals with the analysis of programs by working with symbols representing arbitrary values instead of certain input values [59]. James King described in 1976 that the approach "[...] offers the advantage that one symbolic execution may represent a large, usually infinite, class of normal executions" [59, p. 394].

To illustrate symbolic execution, we will use the example that is given in [4, pp. 115 sq.] for KeY. The rule sets for Java Card DL employ symbolic execution to reduce complex programs to simpler ones. In contrast to Hoare logic, where relatively few rules are used, rule sets for dynamic logic can contain a vast number of complex rules. Therefore we will only look at two very basic symbolic execution steps.

The example sequent we will analyze is:
$\Rightarrow \langle \texttt{o.next.prev=o;} \rangle \texttt{o.next.prev} \doteq \texttt{o}$
The Java Card code in the diamond operator is symbolically executed, which yields a sequent containing a longer, but simpler program:
$\Rightarrow \langle \texttt{ListEl v; v=o.next; v.prev=o;} \rangle \texttt{o.next.prev} \doteq \texttt{o}$
The first of the simplified assignments can fail if `o` is null. Therefore, two distinctive cases have to be analyzed:
$\texttt{o} \neq \texttt{null} \Rightarrow \{\texttt{v} := \texttt{o.next}\} \langle \texttt{v.prev=o;} \rangle \texttt{o.next.prev} \doteq \texttt{o}$
$\texttt{o} \doteq \texttt{null} \Rightarrow \langle \texttt{throw new NullPointerException();} \rangle \texttt{o.next.prev} \doteq \texttt{o}$
Both subproofs then have to be continued with additional rule applications, which we omit here because of their complexity. Many rules are listed and explained in detail in [4, pp. 120 sqq.].

The example shows the methods of introducing case distinctions and performing syntactic updates. In general, symbolic execution cannot be applied to loops as no bound on the number of iterations can be determined when using symbolic values. Thus, symbolic execution needs to be combined with other methods for complete verification. In KeY, for example, verification of loops can be performed either by providing loop invariants or by using induction [4, p. 116], with both methods needing assistance from the user.

2.6 Model checking

Model checking was introduced by Edmund M. Clarke and E. Allen Emerson in 1981 [13]. It offers an approach different from the methods based on Hoare logic. Most explanations given here are based on the chapter on model checking in [47].

Model checking is based on temporal logic:

> The idea of temporal logic is that a formula is not *statically* true or false in a model, as it is in propositional and predicate logic. Instead, the models of temporal logic contain several states and a formula can be true in some states and false in others. Thus, the static notion of truth is replaced by a *dynamic* one, in which the formulas may change their truth values as the system evolves from state to state. [47, p. 174]

There exist various types of temporal logics, two that are widely researched are linear temporal logic (LTL) and computational tree logic (CTL). We will give a short introduction to LTL and also briefly look at CTL.

The set of LTL formulas is defined recursively as follows:
$\phi ::= \top \mid \bot \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (X\,\phi) \mid (F\,\phi) \mid (G\,\phi) \mid (\phi\,U\,\phi) \mid (\phi\,W\,\phi) \mid (\phi\,R\,\phi)$

p is any atom from propositional logic. X, F, G, U, W and R are called temporal connectives. LTL formulas are used for describing models of *transition systems*. To explain the semantics of LTL, we need the notion of a *path* π, which is defined as an infinite sequence of states $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$. π^i denotes a sub-path of π starting at state s_i. $L(s)$ is the set of atomic propositions which are true at a state s. The satisfaction relation $\models$ defines whether a path π satisfies an LTL formula. The definitions are taken from [47, pp. 180 sq.].

- $\pi \models \top$
- $\pi \not\models \bot$
- $\pi \models p$ iff $p \in L(s_1)$
- $\pi \models \neg\phi$ iff $\pi \not\models \phi$
- $\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$
- $\pi \models \phi_1 \vee \phi_2$ iff $\pi \models \phi_1$ or $\pi \models \phi_2$
- $\pi \models \phi_1 \rightarrow \phi_2$ iff $\pi \models \phi_2$ whenever $\pi \models \phi_1$

- Next state: $\pi \models X\,\phi$ iff $\pi^2 \models \phi$
- Some future state: $\pi \models F\,\phi$ iff there is some $i \geq 1$ such that $\pi^i \models \phi$
- All future states (globally): $\pi \models G\,\phi$ iff, for all $i \geq 1$, $\pi^i \models \phi$
- Until: $\pi \models \phi\,U\,\psi$ iff there is some $i \geq 1$ such that $\pi^i \models \psi$ and for all $j = 1, \ldots, i-1$ we have $\pi^j \models \phi$
- Weak until: $\pi \models \phi\,W\,\psi$ iff either there is some $i \geq 1$ such that $\pi^i \models \psi$ and for all $j = 1, \ldots, i-1$ we have $\pi^j \models \phi$, or for all $k \geq 1$ we have $\pi^k \models \phi$
- Release: $\pi \models \phi\,R\,\psi$ iff there is some $i \geq 1$ such that $\pi^i \models \phi$ and for all $j = 1, \ldots, i$ we have $\pi^j \models \psi$, or for all $k \geq 1$ we have $\pi^k \models \psi$

A system's state satisfies an LTL formula if *all* paths from this state satisfy the formula: "Thus, LTL implicitly quantifies universally over paths. Therefore, properties which assert the existence of a path cannot be expressed in LTL." [47, p. 207] If an answer to the question whether there *exists* a path from a state satisfying formula ϕ is required, it is possible to check whether *all* paths satisfy $\neg\phi$ and interpret the result negated. However, it is not possible to mix universal and existential quantification. For this, a different type of logics called branching-time logics is required. We will take a short look at computational tree logic (CTL).

In CTL, the temporal connectives are augmented by an additional symbol A or E, for example X is used in the form $AX\,\phi$ or $EX\,\phi$ and U is used in the form $A[\phi\,U\,\phi]$ or $E[\phi\,U\,\phi]$. A specifies that the connective has to be satisfied along all paths, while E specifies that there has to exist at least one path where the connective is satisfied. CTL therefore is more expressive than LTL in some cases – but the reverse is also true, as some facts cannot be expressed in CTL due to the requirement that every temporal connective has an associated A or E. To remedy this, a superset of LTL and CTL called CTL* was defined, where syntax consists of state formulas, that are interpreted with regard to the system's states, and path formulas, that are interpreted along paths. Thus, CTL* combines the expressive powers of its subsets.

As a model checker has to evaluate all paths a program might take and check whether they satisfy the system specification, model checking severely suffers from the problem of state space explosion [27, p. 37]: The number of states grows exponentially with every program variable and every thread in multi-threaded programs. Thus, model checking is more widely used for the verification of hardware or low-level software such as drivers

than general-purpose software [27, p. 39]. A number of approaches has been considered for alleviating this problem, for example the use of data structures that represent sets of states instead of individual states (ordered binary decision diagrams, symbolic model checking) [47, pp. 229, 383]. Another example is counterexample-guided abstraction refinement (CEGAR) [61], where model checking starts with a coarse abstraction that is iteratively refined only if necessary. A further approach is bounded model checking where verification of loops is just performed up to a certain number of loop iterations (implemented for example in CBMC, see section 3.1.5 on page 36).

CHAPTER 3

Tools

Although many foundations of concepts dealing with formal verification were described years or even decades ago, the number of tools available for applying those concepts to practical use on a larger scale is still rather small and their features and capabilities vary greatly. This chapter will describe those tools that are designed not just to implement or automate a certain theoretical foundation, but that also have potential for use in practice and in larger projects.

For each of them we are going to give an overview of its origin such as its developing institutions. Basic usage aspects will be covered, for example which operating systems are supported, which programming languages can be verified and in what way interaction with the tool is performed, such as the availability or absence of a graphical user interface, either for the tool itself or in the form of plug-ins or similar mechanisms for existing integrated development environments. Each tool's theoretical background will be described. The intended field of use and target groups are going to be analyzed: Is the tool aimed at academic or industrial use? What kind of knowledge of formal verification is necessary to successfully use the tool? In addition, aspects such as the license under which the respective tools are made available, the state of support and development and the prospect on long-term availability will be considered. Properties that have a large influence on the tools' suitability for industrial applications and teaching will only be touched in this chapter to be analyzed in more detail in chapter 5 on page 105.

In general, we are going to focus on those tools that make it possible to get results even for a user with little experience in the field of formal verification.

In 2004, Feinerer et. al. evaluated four tools [37; 38], namely *Frege Program Prover*, *KeY*, *Perfect Developer* and *Prototype Verification System*. In determining whether it would be worthwhile to analyze them in detail again, different aspects have to be considered. We will look at the progress the tools' features and usability have made, and whether or not they are still actively developed. Those that did not undergo substantial improvements are only covered in little detail.

3.1 Tools not selected for comparison

3.1.1 Frama-C/Jessie

Overview

Frama-C (FC) is developed by the French institutions CEA LIST (Lab of applied research on software-intensive technologies at Commissariat à l'énergie atomique et aux énergies alternatives) and INRIA Saclay (a research center of National Institute for Research in Computer Science and Control) [9, About us]. It is a framework "[...] dedicated to the static analysis of source code written in C" [18, p. 11]. In contrast to some of the other tools Frama-C is not monolithic, but is meant as a platform to be extended by a number of plug-ins for different purposes. It provides a parser for C programs that also understands annotations in *ACSL* (ANSI/ISO C Specification Language) [18, p. 12]. The project offers plug-ins for deductive verification (Jessie and wp), extracting semantic information from code, static analysis of variable values at different points in programs, determining what code parts are affected by a modification (impact analysis), and some more [18, p. 12][9, Plug-ins].

Background

Frama-C can be used either as a console application [18, p. 16] or by utilizing a graphical user interface [18, p. 35]. Two of Frama-C's plug-ins are in the scope of our comparison: Jessie and wp. Both of them implement deductive verification based on Dijkstra's weakest preconditions, an approach that is explained in section 2.3. According to wp's manual, wp is the newer plug-in, but it does not supersede Jessie and instead is aimed at different aspects:

> The Jessie memory model is very efficient for a large variety of well structured C-programs. However, it does not apply when low-level memory manipulations are involved, such as heterogeneous

> casts. Moreover, Jessie operates by translating the C program to Why, a solution that prevents the user from combining weakest precondition calculs [sic] with other techniques [...] [17, p. 7].

The main component of the *Why* platform mentioned in the citation is a general-purpose generator for verification conditions that is driven by input in the likewise *Why*-titled intermediate language.

The annotation language used by Frama-C, ACSL, is developed by the same institutions as the tool itself. It "[...] aims at specifying behavioral properties of C source code" [3, p. 11] and is, indirectly, by way of another tool called Caduceus, inspired by Java Modeling Language. ACSL annotations are added to the source code as comments and are thus ignored by a regular compiler [3, p. 12].

As our focus is on formal verification in general software development and teaching, Jessie seems to be the more suitable plug-in. In a diagram in Jessie's tutorial [64, p. 3], the chain of actions for conducting a proof with Frama-C/Jessie is given. It can be summarized as follows: The program in C annotated in ACSL is processed by Frama-C and converted into an intermediate language that is used as input for the Jessie plug-in. Jessie generates code in Why which is given to Why's verification condition generator. Why can generate verification conditions for many different provers, automated provers like Alt-Ergo, Simplify and Z3 or interactive ones such as Coq, Isabelle and PVS [62]. Resulting from this long chain of interdependencies, installation and configuration is a rather complex task. The positive aspect of this is that since Why can supply various provers with verification conditions, a proof can be attempted in more than one prover easily. Thus, the strong points of one prover may outweigh the weaknesses of another, as is demonstrated in the tutorial [64, p. 17]. Unfortunately, a negative aspect results from this, too: The tools of the Why platform provide no way of relating the generated constructs (in Why code) to the respective lines in the original C/ACSL code. This might greatly hinder analysis of any problems found by the provers. Indeed, in the tutorial it is stated that finding the respective line in the C code "[...] can be done by hand for very short functions" [64, p. 4], but not in longer ones.

In addition, the Why platform in the version required by Jessie (Why2) is no longer under development, but has been superseded by Why3 [62]. Although it is stated that Why2 is still maintained as necessary for Jessie, the thought of beginning development with or designing teaching materials for a partly deprecated platform is rather deterrent. Another downside is that Frama-C's current version Carbon is not available for the Windows platform, but only for Linux and Mac OS X, which might be acceptable for some

academic usage scenarios, but not for general industrial deployment. The latest version for Windows is Boron, which is from April 2010 [9, Download].

Documentation and target audience

The documentation provided seems comprehensive at first: Manuals about Frama-C, ACSL, plug-in development and about most plug-ins are available on the project's website [9, Support]. Links to a wiki, mailing list and blog can also be found there. However, the level of detail and quality varies considerably. For some components, like for the Frama-C kernel itself or for the ACSL specification, the documents are extensive and cover many topics. In other cases, like the Jessie plug-in, there is no proper manual available, but just a tutorial with a very short manual chapter, which, while answering some questions, provides only few detailed explanations [64, p. 24]. Additional information available for Jessie would consist of a demonstration video, however the link to it is broken [9, Plug-ins/Jessie]. Offering a wiki seems to be a good idea for documenting practical aspects of the project, but there is only very little content available. The only exception is the comprehensive FAQ page – unfortunately, its latest update was performed in 2009.

There have been quite a few publications on the foundations of Frama-C, ACSL and many plug-ins [8]. A comprehensive manual or book about the project as a whole, written in a consistent style and structure would greatly help understand the structure and usage, but unfortunately none exists.

No support for integration into existing integrated development environments is available. The main Frama-C component has a GUI that can be used as a simple IDE for writing ACSL annotations. For verification with Jessie, the tools available for the Why platform have to be used, in particular the gWhy GUI [64, p. 2].

Some facts could be gathered regarding the practical use of Frama-C/Jessie: In the list of publications in the wiki, an experience report on the verification of algorithms in the C++ standard library [5, p. 191], presented at the International Conference on Formal Verification of Object-Oriented Software, is mentioned. Another paper from this conference presents the experiences with Frama-C at Dassault Aviation [5, p. 205]. Three additional papers on experiences with Jessie are mentioned in the wiki [8].

No information is explicitly given about Frama-C's target audience on the website or in the manuals. The features and plug-ins provided could in general be useful in both academic and industrial environments. This is also because the amount of details the user gets provided with and the level of interaction needed when using Jessie depends on the prover being used and

can thus be chosen according to the user's expertise in the field. Still, no information could be found on the use of Frama-C in teaching, so at least in practice the focus seems to be more on industrial applications.

License and development

Frama-C is provided freely under the GNU Lesser General Public License version 2.1 [9, Download]. The Why platform the Jessie plug-in depends on is available under the GPL as well. Frama-C is actively developed: The latest release is from February 2011, with the exception of the Windows version that is far older. Since various institutions support it [9, About us], the chance of further development of the Frama-C kernel is high. However, the situation is not that clear for some plug-ins, especially for Jessie, because it depends on the deprecated Why2 software.

Support is provided via a mailing list where project members generally answer in timespans of hours to a few days.

The project shows great transparency in regard to bugs and known issues. A public bug tracking system is available on the website [9, Support] which can be viewed anonymously, to report new issues an account can be created easily. Unfortunately there is no redacted, structured list of known issues available that is structured by severity of issues. The filtering features of the bug tracking system cannot effectively be used to gather this information, as most issues do not have a priority level specified and there are also notes and development proposals in the system.

Summary

Frama-C/Jessie presents itself as an interesting platform with some disadvantages. In general, documentation, state of development and support for the main component leave a good impression, but it is not exactly so in case of the Jessie plug-in. The complex interaction between various components and the reliance on the Why2 platform are deterrent to starting development with Frama-C/Jessie for the sole purpose of formal verification. When static analyses like value and impact analysis or approaches like program slicing are desired in the development process besides formal verification, Frama-C might be a powerful option to consider.

For our focus on formal verification in industrial applications and teaching, Frama-C/Jessie seems to be too complicated and, when looking at the uncertain future of Why2, less attractive than other tools. When Why3 becomes stable and if Jessie is updated to use the new platform, a new look at Frama-C/Jessie might be worthwhile.

3.1.2 Frege Program Prover

Frege Program Prover (FPP) [80] is one of the tools that was already analyzed in [37]. It was developed at the Department of Mathematics & Computer Science at Friedrich-Schiller-Universität Jena. FPP performs verification by reasoning about code written in a subset of the language Ada that is augmented by annotations giving information about pre- and postconditions, loop invariants and the like. The tool does fully automatic analysis and verification by computing weakest preconditions for a given program and postconditions or by verifying whether Hoare triples of precondition, program fragments and postcondition hold [81, p. 119]. FPP is a purely online tool that is operated via its website.

An advantage compared to many other systems is that FPP concentrates on some few key aspects that are often focused on when teaching formal verification [47, p. 269; 44, p. 109]. However, supported language features are constrained so tightly that use outside of teaching is impossible and that they might not even be sufficient for entry-level courses. Important features missing are the support for additional data types beside the integer and boolean types that are currently available, for aggregate types like arrays or lists and for any kind of structuring like packages or procedures [80, Syntax]. None of these shortcomings have changed since the work in [37].

Because of this fact, and also because the website states that because of technical problems FPP is currently not available, it will not be included in this comparison.

3.1.3 Perfect Developer

Perfect Developer (PD) [33] is one of those tools that were compared in [37]. It is developed by British company Escher Technologies. It was previewed in 1999 at the World Congress of Formal Methods under the name Escher Tool and commercially released in 2002 under its current name [23, p. 10]. Perfect Developer is designed in a different way than many other tools mentioned here: Instead of writing code in a general-purpose programming language that gets annotated with specifications, the user has to adopt a different paradigm. Escher developed their own specification and programming language called *Perfect*, of which they claim it "[...] has the look and feel of an object-oriented programming language but the power of a specification language" [20, p. 1]. Perfect is used to create a formal specification of the system to be implemented, for example by writing class skeletons and function signatures including pre- and postconditions. No details on the implementation need to be given by the user in this step. Perfect Developer

generates about 50 types of verification conditions based on the specification, which the included automatic theorem prover tries to verify [23, p. 5].

Users have different options regarding how much of the development process should rely on Perfect Developer. In the one scenario, the use of Perfect Developer is complete after checking that the specification is consistent and the program is implemented by hand using regular tools and programming languages while trying to adhere to the specification [32, p. 1]. This approach might introduce arbitrary errors in the code, as the implementation solely is in the hands of the programmer. It is the only choice in some other verification tools, for example in Prototype Verification System. However, Perfect Developer allows to automatically generate program code conforming to the specification. The target languages currently available are Ada, C++, C# and Java [32, p. 2]. If Perfect Developer has trouble generating code for certain specifications or if the generated code's performance is not sufficient, the user can provide *refinements* to the specification. These are imperative annotations that tell Perfect Developer *how* the specifications should be implemented. Refinements are also devised using the Perfect language, can be formulated in an object-oriented way and can also be verified by PD [23, pp. 9 sq.].

Perfect Developer was very positively received in the comparison by Feinerer et. al., where it was stated that it is the only tool of those compared "[...] that comes close to the ideal of automatic and easy program verification" [37, p. 80], along with the criticism of missing support for induction and "[...] the lack of information concerning the inner workings of the prover" [37, p. 80], since the user does not get access to the logical rules used to construct proofs. These criticisms still apply to the current version of Perfect Developer.

Escher is currently developing a second tool called Escher C Verifier (eCv) that, similar to many other tools analyzed in this work, strives to verify programs written in C with additional annotations. As eCv uses the same theorem prover as PD [31], and since nothing has changed regarding the main criticisms, we will not include Perfect Developer in this comparison, but instead we will analyze Escher C Verifier.

3.1.4 Prototype Verification System

Prototype Verification System (PVS) [72] is another of the tools already analyzed in [37]. PVS is developed at the Computer Science Laboratory at SRI International and was introduced in 1992 [70]. PVS is a more formal tool. The source code is available under the terms of the GNU General

Public License. Binary distributions are provided for Linux and Mac OS X operating systems.

In PVS, specifications formulated in a higher-order logic language also called PVS are verified with the help of an interactive proof checker. For solving theorems, the prover depends on the user's knowledge and intuition for choosing the commands suitable for progressing towards the complete proof. To help with recurring problems, proof strategies "[...] from quite sophisticated primitive inference steps that employ arithmetic decision procedures, rewriting, and simplification" [73, p. 2] can be assembled. Some examples of readily available proof strategies for various knowledge domains are also mentioned there. Still, when there exist no proof strategies for a user's problem yet, creating them is a highly complex task requiring good knowledge of PVS itself as well as logic in general. Examples for designing proof strategies are given in [73, p. 12].

A positive aspect for use in teaching could be that the prover gives detailed output that shows which rule applications yielded which results, see for example [71, p. 13]. On the downside, there is no nexus between specifications and proofs in PVS and code in a programming language. This means that while a system's specification can be verified, its realization in a programming language is still completely up to the user and may introduce arbitrary errors that cannot be detected by the verification system. Combining these aspects results in the notion that PVS only has a small potential for industrial use. For teaching formal methods, as stated in [38, p. 300], PVS could be suitable for (very) advanced students, but not for entry-level courses.

Resulting from this, and from the fact that no substantial changes have occurred since the comparison in [37], PVS is not included in this comparison.

3.1.5 CBMC

Overview

CBMC [40] is developed by the Formal Verification Group of Carnegie Mellon University, University of Oxford and Eidgenössische Technische Hochschule Zürich. It is part of the CPROVER project that offers different tools for software and hardware verification. Its design is different from the other tools analyzed here in that it is the only tool that uses model checking for verifying software. Model checking is often used for verification of hardware or embedded software, and CBMC is also aimed at low-level, embedded software [12, p. 168]. However, features such as dynamic memory allocation that are seldom used in embedded software are supported [12, p. 170]. CBMC

is based on the method of bounded model checking which was proposed in 1999 in [7].

Background

CBMC is used to verify programs written in C. It is provided in binary form for Windows, Linux and Mac OS X as well as in source code by public access to the version control system [40]. On Windows, the compiler of Microsoft's Visual Studio is needed as a prerequisite before being able to use the tool. CBMC is a console application. The main paper contains screenshots of a graphical user interface [12, p. 174], but no evidence of a GUI could be found in the current version. Instead, a plug-in for the Eclipse IDE is available.

CBMC uses assertions to specify program properties. Assertions are either generated automatically by CBMC, or they are added to the code by the user. Automatically generated assertions aim to make C safer by checking for the presence of possible buffer overflows, the use of pointers, arithmetic properties (such as divisions by zero), the use of uninitialized variables and concurrent accesses to a single variable from more than one thread [41, Property instrumentation]. Many of these aspects are already intrinsic to "modern" languages like Java and C#. However, especially in the field of embedded software, C is very widely used and thus dealing with C's ambiguities is important when targeting the verification of embedded software.

The user can specify additional assertions to enable verification of more aspects of the program. Unfortunately, CBMC only provides the ability to process assertions that are in the form of standard ANSI C expressions [12, p. 173], which limits the power of constructs used in assertions. As program inputs normally can take any form, model checking uses nondeterminism: In the simulation, "[t]he program may follow any computation that results from any choice of inputs" [41, Nondeterminism]. Nondeterminism can also be explicitly introduced in a program, which can be used to reason about ranges of values. Still, nondeterminism cannot compensate for the lack of existential and universal quantifiers. Future releases are planned to have support for quantification [41, Modeling with Assertions and Assumptions].

CBMC implements the method of bounded model checking. Its aim is to circumvent the state explosion problem commonly associated with model checking. A program that contains loops is *unwound* a fixed number of times. This means that each loop in the code is replaced by a series of nested statements consisting of an `if()` condition with the semantics of the condition in the loop's `while()` statement and a copy of the loop body [12, p. 169]. Information on model checking in general is given in section 2.6.

After these changes, the program is transformed into a boolean satisfiability problem (SAT), which a solver tries to prove. If the SAT formula is satisfiable, the program contains an error [7, p. 194]. If the solver finds no proof, it is possible that either the program is correct or that unwinding needs to be performed for an increased number of loop iterations. CBMC is able to detect such a case by adding an *unwinding assertion* to the deepest copy of the loop body. If this assertion is encountered by the symbolic simulation performed by CBMC, this is evidence that more unwinding needs to be performed [12, p. 170]. If the assertion is not encountered and no other assertions are violated, the loop is proven to be correct. The necessary bound for the number of unwinding operations can in some cases be determined by CBMC itself, otherwise it has to be given by the user. Loops where a sensible bound is not known or which are expected to repeat indefinitely cannot be proven completely, it can only be said that the program is correct up to a certain number of loop iterations.

In the case that the SAT formula is satisfiable, a counterexample is extracted from the prover's output which can be presented to the user to enable him to understand the actions that led to the failure [12, p. 168].

An interesting aspect of CBMC stemming from the orientation to low-level software is the possibility to prove behavioral consistency between a C program and a system specified in the hardware description language Verilog. An approach that is often taken in hardware design according to [11, p. 308] is the following: First, a prototype implementation in C is created, which is used for testing. After completing this phase, the system is modeled in a hardware description language such as Verilog. This process can introduce new bugs that are seldom found since additional debugging and testing in hardware description languages are time-consuming tasks. CBMC can generate a SAT problem from both the C program and Verilog description and verify that their behavior is consistent. Details are given in [11, pp. 309 sqq.; 41, Hardware/Software Co-Verification].

Documentation and target audience

The website [40] offers some publications on CBMC. A manual for users in HTML format is available [41]. While it touches many different topics and gives some examples, it is very brief and does not give detailed information on each topic. It might be sufficient for the first steps in CBMC, but cannot be seen as a proper reference manual – a book neither is available. Information about the theoretical aspects of CBMC and bounded model checking in general is given in some papers. The main paper [12] introduces the approach, [11; 60] give more information on loop unwinding, the code

transformation process and the verification of behavioral consistency between C program and Verilog hardware description.

A section on the website lists different scenarios where CBMC was used [40, Applications of CBMC], mainly in verifying embedded or low-level software such as software for microcontrollers and device drivers. Integration into existing development processes is made possible by a plug-in for the popular IDE Eclipse. In the main paper about CBMC in the description of the then available GUI, it is stated that "[w]e hope to make formal verification tools accessible to non-expert users this way" [12, p. 168], so the intended target audience includes software developers without expert knowledge in formal methods. Unfortunately, it could not be verified how well this intention was realized, as we could not get the plug-in to work as intended.

Some university courses where CBMC was used could be found, mainly in the German language area, for example at Vienna University of Technology, Karlsruhe Institute of Technology and Technische Universität München.

License and development

CBMC is provided free of charge under a custom license allowing redistribution and use under the condition of retaining the copyright notice and informing the authors by e-mail when installing CBMC for any purpose [40, License].

The tool is actively developed, with new releases appearing every few months, at least since 2009. The only information that could be found about further development is the announcement of a version with support for quantifiers to be released sometime in the future.

No information could be found on the website regarding known issues with CBMC. There is also no public bug tracker available. Support is provided using a forum linked on the CPROVER website [42, Support]. While some questions posted there have been answered after a few days, many questions still remain unanswered. These aspects make CBMC unattractive for projects where accurate information about the state of the project is desired or where any kind of support is needed.

Summary

CBMC is different in concept to the other tools in this comparison in that it performs model checking for verification. As this is a completely different concept than the approaches taken with the other tools, and since – in contrast to many other model checking implementations – CBMC is suited

not only for embedded applications, a comparison could be interesting. However, a severe restriction exists: only being able to use standard C expressions in assertions can make the formulation of some conditions hard or impossible. Thus, we will favor other tools that do not have such limitations. When the announced new version with support for quantifiers is released, a re-evaluation of CBMC could be beneficial.

3.2 Tools selected for comparison

3.2.1 Escher C Verifier

Overview

Escher C Verifier (eCv) [31] is a new tool developed by Escher Technologies. It was designed as an alternative to Perfect Developer for areas where Perfect Developer's approach is not adequate. The paradigm of Perfect Developer is specifying the system in the Perfect language and, after verifying the specification, automatically generating code that complies – the performance of which might be inferior compared to code optimized by hand (see section 3.1.3 on page 34). For such cases, where formulating each aspect of the code imperatively is necessary, another approach is needed.

Based on this notion, Escher designed eCv, which, similar to some other tools in this comparison, strives to verify programs written in a subset of C. Verification can be performed on different levels. It always tries to prove that programs "[...] are free from out-of-bounds array indexing, null pointer de-referencing, arithmetic overflow and other 'undefined behaviour', and that each loop [...] will terminate" [29, p. 1]. When source code is annotated with additional knowledge like specifications, pre-/postconditions and invariants, Escher C Verifier aims to prove those too.

Background

Escher C Verifier is available for Windows and Linux operating systems in binary form. The final version was released in October 2011, although this description is based on the last pre-release version 5rc10. Escher combined Perfect Developer and Escher C Verifier into a product called Verification Studio that is available in different versions [34]. Verification Studio provides a common graphical user interface to both tools. Integration is not very extensive however, as the user has to decide whether to load an eCv or PD project into Verification Studio. Thus, in practice, the current state of the implementation can be seen as providing two separate tools sharing a very similar GUI, without the support of combined use of both tools in a single project.

eCv is similar to some other tools in that it strives to verify programs written in C with additional annotations. The verification approach used by the tool is called *verified design by contract*, an extension to the term *design by contract* coined in 1988 [65]. It was inspired by Hoare logic and weakest preconditions [20, p. 3]. The tool generates about 50 different verification conditions [22, p. 31] which are verified using a term rewriting system

together with a first-order theorem prover. The system can also handle some higher-order constructs by employing additional rules [21]. Escher C Verifier uses the same prover as Perfect Developer.

All tools have some limitations concerning the language features they support. An interesting aspect of Escher C Verifier is that the decision on which features should be supported was based on a proven standard called *MISRA C*. The C language is still often used because of its efficiency, for example in embedded and server applications, but it is missing some basic features present in more modern high-level languages, for example integrated checking whether accesses to elements of an array are within the array's bounds. Also, the C standards lack definitions of behavior in lots of cases, many of which are important in practice [48, p. 492]. Some institutions have specified subsets of C for use in certain fields. The *Motor Industry Software Reliability Association*'s (MISRA) mission statement is to provide "[...] assistance to the automotive industry in the application and creation within vehicle systems of safe and reliable software" [69]. In this role the MISRA C subset of the C language was developed where many "unsafe" features were removed [68] and which has been adopted in a variety of fields.

The features supported in Escher C Verifier are closely related to the 2004 version of MISRA C: Escher states that "[...] almost all the constructs that are prohibited in our subset are also prohibited in the MISRA subset, although the reverse is not true" [24, p. 8]. Thus, eCv supports even more aspects than MISRA C which backs the claim that existing programs written in MISRA C can be adapted for verification with eCv with relatively small effort [29, p. 1].

The additional keywords needed for annotating the code are implemented using preprocessor macros that expand to nothing at compilation. Thus, any compiler can be used, as all annotations are invisible when compiling [24, p. 8].

A unique feature of the tools developed by Escher is the provision of advanced support in the case that a code segment cannot be proven automatically. While other tools just show the line the problem occurred in – and sometimes do not succeed even in this aspect, see section 3.1.1 on page 30 – Escher C Verifier in some cases can give hints on how to solve the problem. For some types of missing annotations, the verifier suggests amendments:

> For example, if an unproven verification condition involves only the values of inputs to a function, it is very likely that the user forgot to state the required condition as a precondition

> of the function; so the verifier will suggest it as an additional precondition. [24, p. 12]

Although the helpfulness of the suggestions depends on the complexity of the situation, they can increase the user's productivity by decreasing the time spent on debugging "easy" parts of the specification.

Documentation and target audience

Several papers and slides from various conferences about the tools and the verified-design-by-contract approach are available on the website [31, Publications]. The founder of Escher Technologies, David Crocker, has a weblog where many practical aspects of eCv and verification of C programs in general are illustrated [19]. As Escher C Verifier has only just been released product documentation is not as extensive as that for Perfect Developer, for which there exists a comprehensive self-help section [33, Support]. Among the documents provided is a user guide, a language reference manual, tutorials for beginning work with the tool, a complete list of the verification conditions generated and a frequently asked questions section. Even though no book or other extensive work exists, the amount and quality of documentation can be considered as an advantage compared to the other tools. It will be very positive if the full documentation on eCv is going to be similar to that of PD.

Escher C Verifier is mainly targeted on industrial users, but support for the use in teaching is not neglected either. Escher claims its tools are well-suited for developing applications adhering to various safety standards [35] and that Perfect Developer has already been employed in various industrial projects – as Escher C Verifier is a new tool, no experiences with industrial usage exist yet. Unfortunately, there is no support for the most widely used integrated development environments Eclipse and Visual Studio. For the Perfect language, at least there exist customization files adding syntactic support to a few editors: To more common ones like Vim, but also to obsolete ones like Crimson. For the additional language keywords needed for Escher C Verifier even such customization files were not available as of version 5rc10, even though they were advertised for the final release [29, p. 2]. Instead, the user needs to configure the editor she uses herself, which is not very attractive.

Verification Studio itself does not offer a code editor either, but was designed as a project manager with verification abilities: Source files in C or Perfect are organized into projects and can all be verified in one step. Messages about the state of verification conditions are output in a relatively

well-arranged way (see figure 3.1), and it is possible to directly jump to the corresponding line of code (if the editor supports this), which is convenient. Still, integration into an IDE could increase productivity even further.

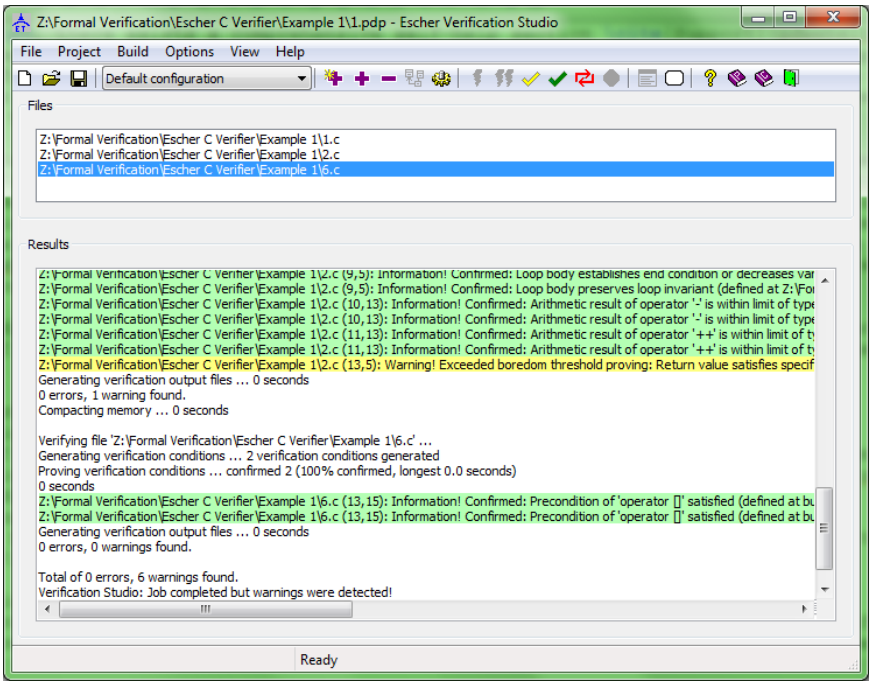

Figure 3.1: Main user interface of Verification Studio

Regarding use in teaching, we again have to fall back to looking at the current situation of Perfect Developer and assume that the materials provided for Escher C Verifier will be similar. The self-help section on the website [31, Support] offers a page with some teaching materials for Perfect Developer, for example a comprehensive one-day tutorial including a number of exercises, which should be well suited for an introductory lesson in a formal methods course. PD has already been used in courses at various universities [36].

License and development

Escher C Verifier is a commercial tool. In the form of the product Verification Studio, several combinations of Escher C Verifier and Perfect Developer are available. A free-of-charge version that is limited to noncommercial use (and subject to some additional terms) is also available upon request for evaluation and educational institutions [34].

Both tools are under active development, with the first release of eCv appearing as a component of Verification Studio in October 2011. Escher anticipates that some customers will be using both tools to work on the same projects and thus announced a closer integration of each other in future releases [34]. As mentioned above, eCv provides suggestions for annotations in some cases, for example when a loop invariant for the loop counter is missing. An important feature that could bring large gains in productivity would be the automatic generation of suggestions also for loop invariants that describe the meaning of a loop. The plan to work on this problem has already been announced in [24, p. 14], but it is not clear whether any progress has been made on this issue.

As Escher C Verifier is a commercial product, there is professional support included which probably is an advantage for companies thinking about introducing formal verification, but low-priority support is also provided for the free of charge version. For Perfect Developer, a clearly arranged list of known issues is available in the self-help section on the website [31, Support]. Issues are structured by severity and detailed explanations and, if applicable, examples are given for each of them. We expect that the same approach will be taken with Escher C Verifier.

Summary

Escher C Verifier follows a similar concept as Frama-C/Jessie and VCC (described in section 3.2.3 on page 50). An interesting aspect is that, in contrast to the other tools, the C language subset supported by eCv is based on the widely used standard MISRA C. Good documentation and the availability of professional support are advantageous especially in industrial environments. The unique ability to suggest missing annotations in the case that a proof fails can increase productivity in some cases. An area that is rather neglected in the design of Escher C Verifier is the integration into standard IDEs. Despite this criticism, eCv already seems like a mature tool suitable for general use. The analysis from a practical point of view will show whether this impression holds true.

3.2.2 KeY

Overview

KeY [58] was already analyzed in [37]. KeY is developed in a joint project of Karlsruhe Institute of Technology and Chalmers University of Technology. It is aimed at verifying properties of code written in *Java Card*, a "[...] superset

of a subset [. . .]" [4, p. 375] of Oracle's Java platform. Annotations are added to the code using Object Constraint Language (OCL) or Java Modeling Language (JML) [28, p. 2]. For verification, KeY transforms programs into a type of dynamic logic called Java Card DL and then performs symbolic execution [1, pp. 42 sq.].

Background

KeY is available for download in Java source code and binaries as well as a web application run directly in the browser using Java Web Start. Thus, all operating systems for which a Java runtime environment exists are supported.

Java Card was designed to be primarily used in memory-constrained, security-sensitive embedded devices such as smart cards [76, p. xvii]. While it is positive that a proven standard has been selected for the language basis of KeY, Java Card lacks many features the Java platform normally provides, for example multi-threading, object cloning, data types for floating point numbers, arrays with more than one dimension, and many classes such as `java.lang.String` [76, pp. 2-2–2-5]. In contrast, Java Card also provides some features not present in regular Java, for example object persistence and atomic transaction mechanisms [4, p. 375]. Important features of object-oriented languages such as inheritance, interfaces and overloading are also still supported [63, p. 5]. KeY extends the features of Java Card in some ways, for example the use of string classes is supported since version 1.6 [56]. Even so, these aspects point to the insight that developing in Java Card (and thus also the use of KeY) is probably not feasible for applications designed for general use. But still, for users working in the area of embedded or similar software, Java Card and KeY could be a very suitable choice. Use of KeY in teaching might also be worthwhile as the possibility of verifying programs written in a modern language with most features regarding object-orientation supported is an interesting aspect.

The code needs to be enriched with annotations for verification. In 2004, when the comparison in [37] was performed, only OCL was available, which was found to be insufficiently expressive for some problems, such as specifying the postcondition of an algorithm that computes factorials [37, p. 57]. Since release 0.99, KeY also supports specifications in JML [56]. OCL was designed as an addition to the Unified Modeling Language (UML), and as such cannot consider language-specific needs. JML is better suited for specifying properties such as exception handling, has support for concrete data types used in Java Card and supports additional clauses which can for

example be used to specify which variables a function is allowed to modify [63, p. 9].

The annotated program is translated by KeY's verification middleware into a formalisation in a variant of dynamic logic [1, p. 34]. Java Card DL can be seen as being loosely related to Hoare logic [1, p. 42] and is used to express the program in a form that KeY's deduction component can reason about. The deduction component then performs symbolic execution and unfolds complex expressions into simpler ones. More information about symbolic execution in KeY can be found in [4, pp. 115 sq.]. Symbolic execution and dynamic logic are explained on a general level in sections 2.4 and 2.5. The simplified expressions are then transformed by using special rules called *taclets*: "A taclet combines the logical content of a sequent calculus rule with pragmatic information that indicates when and for what it should be used." [1, p. 45] The prover can be used with varying degrees of automation: Proofs can be attempted fully manually using an interactive prover interface, or different levels of heuristics can be activated to support partly to fully automatic proof search [1, p. 46].

KeY differs from many other tools in the fact that, while discharging proof obligations can be performed mostly automated, the choice of *which* obligations are to be proven is up to the user. Thus, a user cannot simply tell KeY to verify all possible aspects of a sufficiently annotated program. Instead, for each of the program's classes and methods, the user has to analyze by herself which obligations need to be proven and make sense, and select one of them in the proof obligation browser (see figure 3.2).

For some obligations, the user has additional choices. For example, in the very common proof of verifying a method's compliance with given postconditions, the user has to select the contract to use in the proof (see figure 3.3), if more than one contract is available. A contract specifies a behavioral scenario for a method, for example which postconditions to apply in relation to certain preconditions, or the expected behavior when problems like Java exceptions are encountered at runtime. Then, finally, the proof can be initiated. This process has to be repeated for each class and method that exists in the program and for every proof obligation that needs to be verified.

Documentation and target audience

The main paper about KeY was already published in 2005 [1], but many points made there are still valid. A book edited by some of the main authors of KeY is available as well, it is based on the outdated version 1.0 [4]. Beside these publications, the website offers an extensive list of papers, many of

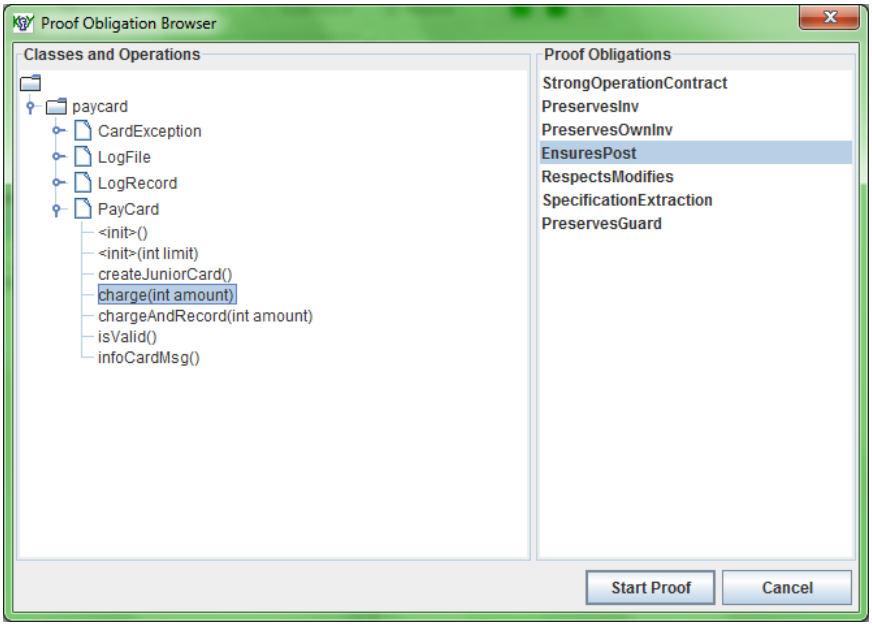

Figure 3.2: Proof obligation browser

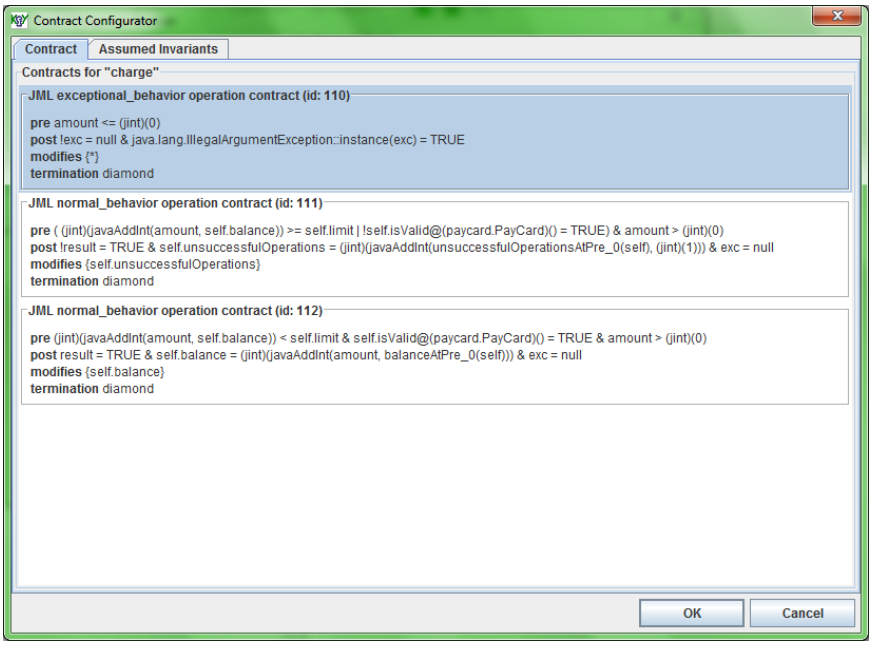

Figure 3.3: Contract configurator

which are however only marginally related to KeY itself and focus more on general or theoretical topics [58, Publications].

According to [1, p. 35], the goal for KeY was to make it usable in different scenarios, for example in general industrial software development, in the development of security critical software as well as in education. It is even explicitly stated that "the most important target user group for the KeY tool are people who are not experts in formal methods" [1, p. 35]. An attribute that was criticized in [37, p. 52] was the need for purchasing and setting up a commercial computer-aided software engineering tool that was then required for KeY's user interface. This criticism does not apply anymore: KeY is now stand-alone and thus has its own user interface that does not depend on any external components.

In contrast to many other tools, KeY has a vast amount of configuration options. For example, there is a large number of options regarding proof search strategies and how to apply which rules and taclet options. In the quicktour document, there even is a whole section dedicated to configuring KeY in the correct way for being able to follow the tutorial [28, p. 12]. The multitude of options offers enough flexibility for using KeY in different ways by different users: Beside formal verification, the tool can for example also be used as an interactive theorem prover for first-order predicate logic [1, p. 36] or for the generation of unit tests based on proof attempts [28, p. 28].

The project offers a plug-in for the popular IDE Eclipse. Unfortunately, it does not directly support the verification features of KeY. It just provides the possibility to call KeY from Eclipse and to visualize execution paths in the case that a proof cannot be performed automatically [28, p. 23]. The selection of classes, methods and contracts to prove and the proving process itself still need to be carried out in the completely separate KeY window and just the visualization is performed by coloring code in Eclipse's editor. A positive aspect regarding productivity is the use of Java Modeling Language as KeY's annotation language, as this enables the user to make use of existing utilities that support manipulating JML specifications.

KeY was already used in teaching, at least at the universities involved in its development. The teaching section on the website offers links to some courses, however they do not give much information as they are mostly outdated: one of the courses is from 2006 and two other course sites do not exist anymore. KeY seems to be more widely used in research than in the software industry, as only little information could be found regarding practical experiences. The website lists two case studies [58, Case Studies].

License and development

KeY is freely available under the GNU General Public License version 2. The project is actively developed, at the moment work is being done on version 1.7. What kind of changes are planned for this version is unknown since no documentation on this topic could be found. As KeY has been in development at least since 2002 (when the main paper was handed in for publication [1]) and some people involved in development then still are part of the project, there is good potential for long-term availability. It would be positive if future versions could increase the amount of language features supported even further, similar to the addition of strings with version 1.6.

There is no public bug tracking system available, but the website has a list of known issues with explanations and in some cases also hints for working around an issue [57]. An e-mail address for support on KeY is provided in the download section [58, Download].

Summary

KeY has evolved considerably since the analysis in [37]. The purely academical tool with very complex and frequent user interactions depending on a commercial development environment has changed into a stand-alone platform that can perform proofs automatically in many cases. As substantial changes have occurred in KeY, an analysis at its current state of development, with special attention paid to the amount and quality of user interactions seems to be beneficial.

3.2.3 VCC

Overview

VCC [67] is being developed at the Microsoft Innovation Center Europe in Germany and at the Microsoft Research in Software Engineering group in the United States. VCC, similar to Frama-C/Jessie, deals with the verification of C source code enriched with annotations. Another similarity is that VCC also provides an interface for extending functionality via plug-ins [15, p. 37]. In contrast to Frama-C, formal verification is the main feature in VCC, so any plug-ins will not be considered in this analysis. The unnamed annotation language used was developed especially for VCC. A noteworthy property of VCC is that it was designed especially with formal verification of multi-threaded (concurrent) programs in mind [15, p. 24].

Background

Microsoft Innovation Center Europe was one of the institutions taking part in the Verisoft XT research project on formal verification of computer systems [78]. Stemming from this program, a project to formally verify Microsoft's, at that time new, hypervisor Hyper-V was initiated [15, p. 23]. The requirements of this project drove the development of VCC and are heavily reflected in VCC's focus on "[...] sound verification of functional properties of low-level concurrent C code" [15, p. 27].

VCC is a console application only available for Windows. A current version of the .NET runtime environment and a redistributable of the F# programming language are required as additional components. The verification process is structured in a way similar to Frama-C/Jessie [15, p. 37]: The user writes annotations that are parsed by VCC. The code is then simplified and source code in the Boogie intermediate language is generated. Boogie, which is another project at Microsoft Research, is also the name of the tool that then processes the code. It is the analogon to the Why platform used by Jessie: Boogie is based on the approach of weakest preconditions [2, p. 380] and generates verification conditions that can be passed to a prover that supports satisfiability modulo theories (SMT) [66]. The default prover is Z3 (also a Microsoft Research project) but other SMT provers like Simplify are also supported, and there exists a backend for the interactive Isabelle prover, too [6]. For trying out VCC, a simple online version is also available on the website [67].

The annotation language introduces additional keywords, which are implemented as C preprocessor macros that are defined as to expand to nothing when compiling. This means that the annotations are invisible to a regular C compiler, and so any compiler can be used [15, p. 26].

Special attention was paid to providing suitable debugging tools. When the prover Z3 refutes a verification condition, it generates a counterexample which Boogie and VCC try to map to the respective lines in the original source code. To aid the user, VCC Model Viewer is available "[...] that allows inspecting the sequence of program states that led to the failure, including the value of local and global variables and the heap state" [15, p. 38]. Other tools offered are Z3 Inspector and Z3 Axiom Profiler. They are used in those cases where the prover needs an unacceptably long duration to refute or prove the program. In such situations they can provide an insight into the details concerning for which verification condition the prover takes long to find a counterexample (in the case the condition is invalid) or into which verification condition takes a long time to prove (in the case the condition is valid) [15, p. 38].

VCC uses an approach called locally checked invariants (LCI) for verification. "LCI assigns to each object (including threads) a two-state invariant, i.e., a predicate over pairs of states expected to hold for every pair of consecutive states in every execution." [14, p. 481] The idea is that the verifier checks whether all invariants and state transitions have certain properties, namely being *admissible* and *safe*, in the paper's terminology, respectively. When all invariants in the program are identified as being admissible, then, when checking whether a state update adheres to all invariants, VCC solely needs to check the invariants of the updated objects [15, p. 32]. As VCC targets the verification of system code, each memory write results in a new state, and thus arbitrary low-level features of C can be used, for example unions [67]. VCC supports most features of C, with probably the most notable exception of only restricted support for floating point numbers [67, Unsupported C Features]. A detailed explanation of LCI is given in the paper [14] and a good overview on VCC's methodology, including verification of concurrent programs, is given in [15, pp. 27 sqq.].

Documentation and target audience

There is a number of scientific papers available for VCC and its related components: The main paper on VCC [15], the description of the methodology used by VCC, LCI [14], details on the verification process, the memory model and some more are listed on the website [67, Papers]. Documentation from a practical point of view is available, too. A draft version of a comprehensive tutorial that is currently being worked on is provided on the main page. A separate page is dedicated to give examples and some explanations on frequently needed topics, such as loops, pointers and arithmetic operations. However, it is stated there that some content is outdated and even that "[m]ost of VCC syntax is in flux right now" [67, Documentation].

Indeed, there is currently a complete reimplementation of VCC underway. The current version 2 will be replaced by VCC3. It is based on a new memory model and redesigned axiomatization, which should result in better performance [67, p. VCC3]. VCC3 is already included in the same package as version 2 and can be used by supplying the `-3` parameter, but is not finished yet. Thus, most of the theoretical work published and the documentation section on the website do not reflect the current state of the project. The only current documentation seems to be the tutorial's working draft, which is written for VCC3 [16, p. 2].

Judging from VCC's origin in being designed to aid the verification of Hyper-V, the targeted fields of use are clearly industrial applications. VCC is one of the few tools that offer comprehensive integration into an existing

development workflow: The Visual Studio IDE is widely used for developing in C, C++ or C# for the Windows platform. VCC can be called directly from Visual Studio for verifying whole projects or just selected functions. The debugging aids mentioned on page 51 are also available in Visual Studio. This aspect might be a great incentive for teams already using Visual Studio to look at VCC when thinking about the introduction of formal verification into their projects.

Only little information could be found on practical use of VCC. The verification of Hyper-V still seems to be the main application of VCC. No current information on the state of verification is given. In the main paper on VCC [15, p. 39] from 2009, an experience report is given in which it is said that about 20% of Hyper-V's codebase had been successfully verified then. The only instance of use in teaching that could be found is a course at Technische Universität Berlin [77].

License and development

VCC is provided without cost under a proprietary license called Microsoft Research License Agreement, which basically allows non-commercial use such as in teaching, research and personal projects, but not the creation of any commercial products using the tool. The source code also provided may be used to create derivative works, again only in non-commercial applications. Boogie, which VCC depends on, is licensed under the Microsoft Public License which also allows commercial use. Detailed terms can be found in [67, License].

It seems that the VCC project is quite active: The version control system shows check-ins at least every few days [67, Source code]. Also, a number of people that authored the fundamental paper are still members of the team, implying serious commitment to the project. This, and the fact that VCC is currently being reimplemented with the goal of achieving better performance, seems positive in regard to long-term availability of VCC.

On VCC's website a discussion forum is provided where team members answer most questions in a few days. The project also offers a public bug tracking system. Issues can be viewed anonymously, to create issues the user needs to sign up for a CodePlex or Windows Live account. Similar to Frama-C, there is no redacted overview of relevant known issues, but the user needs to try using the bug tracking system's filtering features to get to the information she wants.

Summary

VCC has distinctive properties because it was designed to support an industrial project, the verification of Microsoft's Hyper-V hypervisor. Thus, special attention was paid to supporting low-level features of the C language and to providing efficient methods for verifying multi-threaded programs. Such features are not the primary concern for teaching formal methods, but multi-threading, although not in the focus of this comparison, is rather important also for general-purpose software nowadays. As the chances seem to be good that VCC will be supported and developed for some time to come thanks to the reimplementation currently underway, we will include VCC in the practical evaluation.

3.2.4 VeriFast

Overview

VeriFast [49] is developed at the Department of Computer Science at Katholieke Universiteit Leuven. It deals with verifying programs written in subsets of C or Java extended with annotations written in separation logic [52, p. 304]. VeriFast performs symbolic execution for verification [54, p. 42].

Background

The verifier is available for download at its website in binary versions for Linux, Mac OS X and Windows platforms. It uses the Z3 SMT solver by Microsoft Research by default, but can also be configured to use the Redux solver. The project offers both a console version and a version with a graphical user interface. Programs can be written using C or Java, but only subsets of the regular language features are supported in both cases. Unfortunately, not much information could be found regarding the current state of language support. For C, the grammar of the then current implementation was given in the main paper on VeriFast in 2008 [50, p. 7]. An updated version is listed in [51, pp. 3 sqq.]. Both versions are not suited for providing a quick overview about features not supported as there is no concise explanation given. For Java, no information could be found at all.

The user needs to annotate programs with pre-/postconditions and loop invariants in the form of separation logic. This type of logic was introduced in 2002. Separation logic is an extension of Hoare logic that deals with specifying behavior for programs accessing shared mutable data structures [74, p. 55]. It provides an additional rule called frame rule, that "[...] states

that to reason about the behavior of a command `C`, it is safe to ignore memory locations not accessed by `C`" [52, p. 304]. Thus, VeriFast also supports the verification of multi-threaded programs where threads may access shared variables. Annotations are added to the code using the respective language's comment syntax, so they are invisible to regular compilers [52, p. 305].

For verifying a method, VeriFast takes the method's precondition as the initial symbolic state, symbolically executes the method's body and compares the consequential state with the postcondition. Detailed information is given in [54, p. 42; 79, p. 321].

The tool's verification approach needs relatively elaborate annotations in some cases, for example for opening and closing predicates: Predicates stand, in VeriFast's terminology, for named assertions [52, p. 306] that encapsulate other assertions on the symbolic execution heap. Predicates need to be opened to enable the prover to reason about the fields the encapsulated assertions encompass. An extreme example for this is given in [79, p. 327]. The VeriFast team works on generating such, and some additional, annotations automatically. The paper [79] gives an experience report at an early stage.

Documentation and target audience

The project offers a number of papers on VeriFast itself as well as on topics related to the tool [49]. Among the documents is the draft version of a tutorial for beginning work with VeriFast. It consists of many examples sorted by different topics along with, in most cases, rather extensive and helpful comments and explanations. For many topics, one or more exercises are given, with solutions in an addendum to enhance understanding of the matter. A reference manual, also in a draft version, is included in the download package. It seems to be best to use both documents together, as the reference manual offers very few descriptions, but mainly consists of an uncommented list of the supported C and separation logic syntax [51, pp. 3 sqq.]. A more extensive, mature work on VeriFast such as a book or similar publication is not available.

A positive aspect is that there is a large number of example programs available for download that demonstrate many features, including a small game and chat server [49, Example programs].

VeriFast does not offer any integration into standard development environments. Instead, a GUI version of the tool is available. The user interface consists of a code editor and several lists giving information about the current symbolic state when symbolic execution was interrupted, see figure 3.4. The user can also select a symbolic state for which the conditions

applying to the current execution path, the contents of the symbolic heap and the assignments of symbolic variables can be inspected. Retracing the symbolic states with this feature can help to identify mistakes in the code made earlier that led to an error in the current line of code. Another helpful

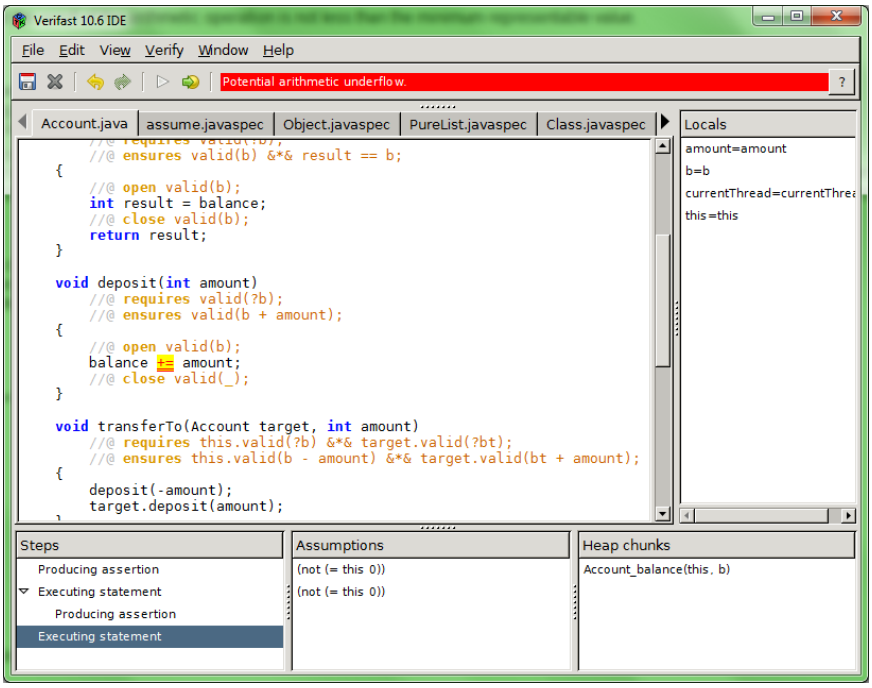

Figure 3.4: GUI version of VeriFast

feature is that in some cases where VeriFast cannot verify a method, the GUI displays a help button that leads the user to a documentation page where information and suggested solutions for the error can be found. Sometimes there are examples available, too, complete with code to solve the problem. This feature can improve productivity for beginners, but is currently only available in "easy" cases, for example for possible arithmetic overflow. While the described functionality is useful, the user interface offers only very few additional features. Hence, it cannot replace a fully functional IDE, so the software engineer still has to work with at least two tools.

VeriFast was already used in teaching at Katholieke Universiteit Leuven and Eidgenössische Technische Hochschule Zürich [52, p. 310]. No information was found about any industrial use of VeriFast. A paper lists some projects oriented on practical use that are currently in progress at the group VeriFast's authors work at: Verification of Java Card programs, integration of shape analysis to automatically generate some annotations, and some early efforts

to verify device drivers for Linux [54, pp. 53 sqq.].

License and development

VeriFast is available for download freely without any license terms. For the Z3 prover it depends on, the Microsoft Research License Agreement applies (see section 3.2.3 on page 53 for a description).

The project is in active development. New releases have been appearing every few months [49, RSS feed]. It is unclear what kind of plans the authors have for future improvements. The only mention of a direction to work on is the improved automatic generation of annotations [79, p. 332].

Unfortunately, the project is not open about known problems and restrictions regarding the tool. There is no list of known issues, public bug tracker, forum or a similar source of information. Also, no e-mail address or other means of direct communication are provided for questions or support, just links to the authors' websites.

Summary

In general, it seems that VeriFast's separation logic approach, while offering the possibility to verify multi-threaded programs, results in more elaborate, lower level annotations than necessary in the approaches taken by some other tools. The possibility of generating more annotations automatically in a future version could greatly enhance usability. While the graphical user interface is advanced in comparison to many other tools, it is still minimalistic and cannot replace a proper integrated development environment. A negative aspect, especially when thinking about using VeriFast in industrial environments, is the low amount (or even lack) of documentation in some cases, for example regarding restrictions on the language features supported or known issues. Still, as VeriFast is in active development and the user interface is an advantage over many other tools, it will be included in the practical evaluation.

CHAPTER 4

Comparison

This chapter compares the tools that were selected in chapter 3 from a practical point of view. The goal is to find out how well the features promised in the tools' documentation work in practice, what kind of problems are encountered and to identify areas where improvements are needed.

The comparison will be performed by verifying small example programs. Examples will consist both of simple arithmetic calculations and basic program constructs. We will also attempt the verification of more complex functions, for example of standard library functions. The programs will be annotated at least with postconditions that describe the values the functions return as precisely as possible.

In addition to this core aspect of formal verification, we will also look at how the tools handle programs that are incorrect from a technical point of view. Standard integrated development environments contain state of the art support for dealing with common problems like syntax errors. Therefore, such simple aspects will not be considered when comparing the verification tools. More complex questions however, for example the detection of faulty memory accesses or going beyond an array's bounds, are in the tools' scope. Finding such errors is an important aspect in verifying correctness, especially for programs written in C.

The amount and quality of information a tool provides regarding problems is crucial for aiding the user in debugging. Attention will therefore be paid to the way the tools present information to the user and any distinctive aspects will be noted.

In the case that the implementation of an example proves to be troublesome, help from the respective tool's developers will be requested.

The following gives a short description of algorithms that will be verified.

- Multiplication: Custom implementation of the multiplication of non-negative integers
- Division: Custom implementation of division of a non-negative dividend by a positive divisor
- Factorial: Recursive implementation of factorial calculation
- Index of maximum array element: Returning the index of one of the maximum elements in an integer array
- Faulty pointers: Accessing an uninitialized pointer and a pointer whose target is out of scope, as well as returning a pointer to a local variable
- Out of bounds array access: Accessing array elements beyond array bounds, both for an array passed as parameter and a statically defined array
- String length: Determining the length of a string
- String comparison: Comparing two strings with each other

All tools provide different annotation languages and thus different ways of formulating formal aspects are required. In some cases the programming language features supported also differ between tools. Still, we will strive to state the algorithms in a way as similar as possible for all tools so that the results can be easily compared.

Each of the following sections is devoted to one particular tool. At the beginning of each section a short overview of language constructs commonly used in annotations will be given. In some cases additional general remarks on the tool will be made. The annotated source code of each algorithm is followed by a description of the proof process, the difficulties encountered and any further noteworthy aspects. In section 4.5 on page 97 the experiences made in the practical comparison will be summarized.

4.1 Escher C Verifier

Escher C Verifier supports various compiler settings. We configured the tool as described for the Visual C++ compiler in the manual [30, Appendix A]. eCv version 5rc10 was used in this comparison. The following list provides a basic overview of some constructs often used in annotations.

`pre` Function precondition

`post`, `returns` Function postcondition

`writes` Lists variables a loop is permitted to write to

`keep` Loop invariant

`decrease` Loop or recursive function variant

`arr.lwb`, `arr.upb`, `arr.lim` Lowest, highest and one past the highest index of an array `arr`

`a..b` Set of integers from `a` to `b`

`=>` Material implication

`:-` When used in an expression together with the quantifier `forall` means "it is the case that", together with `exists` means "such that"

Multiplication

```
#include <ecv.h>
unsigned int multiply(unsigned int termA,
        unsigned int termB)
pre((termA * termB) <= maxof(unsigned int))
returns(termA * termB)
{
        unsigned int ret = 0;
        unsigned int i;
        for (i = 0; i != termB; ++i)
        writes(i; ret)
        keep(ret == termA * i)
        decrease(termB - i)
        {
                ret += termA;
        }

        return ret;
}
```

The formulation of the annotations was easy. A first version of the program was susceptible to arithmetic overflow at the addition inside the loop. To rule this out, eCv suggested to add the precondition `(termA *`

`termB) <= maxof(unsigned int)`. A syntactically similar assertion like `assert(termA * termB <= UINT_MAX)` that could be made in the program code in standard C would not be correct as the assertion itself would be affected by arithmetic overflow. The suggested annotation works because in specifications all variables are handled as if they were typed in eCv's unbounded internal data types, in this case `integer`. The precondition thus always correctly represents the user's intention.

eCv generates 16 verification conditions for this program, all but one could be proven straight away. The only problematic aspect was that eCv could not prove that the value of `ret` adhered to the postcondition. For this, after some trial and error, the loop invariant `keep(i <= termB)` was added. The whole program could then be proven. Later, it was found out that the additional invariant was only necessary because of the way the loop was specified: From the loop header in the usual form `for (i = 0; i < termB; i++)` it cannot be derived that `i == termB` is true after loop termination, as `i` could be larger as well. Thus, it is advisable to use the form `for (i = 0; i != termB; i++)` for specifying the loop, an idea that was promoted in [25, p. 56]. Note that this is no limitation only applicable to eCv, but rather a general requirement.

Division

```
#include <ecv.h>
unsigned int divide(unsigned int dividend,
        unsigned int divisor)
pre(divisor > 0)
returns(dividend / divisor)
{
        unsigned int remainder = dividend;
        unsigned int quotient = 0;

        while (remainder >= divisor)
        writes(remainder; quotient)
        keep(dividend == remainder +
                quotient * divisor)
        decrease (remainder)
        {
                remainder = remainder - divisor;
                quotient++;
        }
```

```
        return quotient;
}
```

The division algorithm could be proven easily. The loop invariant directly expresses the relationships between the variables. In the postcondition the standard division operator is used to clearly express the intention of this function.

Factorial

```
#include <ecv.h>
unsigned int factorial(const unsigned int num)
pre(num >= 0)
pre(factorialGhost(num) <= maxof(unsigned int))
decrease(num)
post(result >= 1)
post(result == factorialGhost(num))
{
        if (num == 0)
        {
                return 1;
        }
        else
        {
                return (num * factorial(num - 1));
        }
}

ghost (
integer factorialGhost(const integer num)
pre(num >= 0)
decrease(num)
returns(num == 0 ? 1 : num * factorialGhost(num - 1))
)
```

Although recursion is supported in eCv, recursive function calls cannot be used in specifications in general. An exception is when formulating the postcondition using the `returns()` statement which just gives information on the function's return value, as opposed to `post()` statements with which multiple, more complex facts can be stated. The specification language offers no keyword for directly denoting the factorial of a number, so the

postcondition basically encompasses the whole program: `returns(num == 0 ? 1 : (num * factorial(num - 1)))` was the first approach. In this version it could not be proven that no arithmetic overflow occurred. As the use of recursion in preconditions is not allowed, no suitable precondition could be found that would have enabled to rule out arithmetic overflow.

A solution was achieved by writing a *ghost function*, which is only used for verification. As a ghost function solely contains annotations and no implementation it is ignored when compiling the program, just as eCv's other keywords. Using the ghost function in a precondition is allowed. With the help of the precondition `factorialGhost(num) <= maxof(unsigned int)` all verification conditions could be proven.

Index of maximum array element

```
#include <ecv.h>
int getMaximum(const int* array numbers,
        const int numbersSize)
pre(numbersSize >= 1)
pre(numbers.lwb == 0)
pre(numbers.lim == numbersSize)
post(result in 0..(numbersSize - 1))
post(numbers[result] == numbers.all.max())
{
        int maximumPosition = 0;
        int maximum = numbers[maximumPosition];
        int i;
        for (i = 1; i != numbersSize; ++i)
        writes(i; maximum; maximumPosition)
        keep(i in 1..numbersSize)
        keep(maximumPosition in 0..(numbersSize - 1))
        keep(forall a in numbers.all.take(i) :-
                maximum >= a)
        keep(numbers[maximumPosition] == maximum)
        decrease(numbersSize - i)
        {
                if (numbers[i] > maximum)
                {
                        maximum = numbers[i];
                        maximumPosition = i;
                }
```

```
        }
        return maximumPosition;
}
```

This program required considerable effort: A mistake was made in one of the loop invariants, which led to the problem that eCv could not prove invariants later on in the program. After many attempts at correcting the latter invariants it was discovered by chance that in fact an earlier invariant caused the problem. The warnings eCv returned were of no help in this case. After gaining this insight, the formulation of the remaining annotations was rather simple.

The `array` ghost qualifier at the `numbers` parameter tells eCv that the pointer references an array. Only then does eCv allow indexing or pointer arithmetic on `numbers`. It is not necessary to think about any checks regarding the possibility of encountering null pointers: eCv always requires that pointers are not null, except when explicitly adding the `null` qualifier to a pointer declaration.

In addition to the user specifying ghost functions, eCv also provides built-in ghost functions that can be used in annotations to easily express some facts. The `max()` function was used to express that the function returns the index of the array's maximum element. This feature and the `forall` keyword enable the powerful and relatively clear formulation of the facts needed to prove the program.

Faulty pointers

```
#include <ecv.h>
int* testPointers()
{
        int a = 1;
        int* ap = &a;

        int* bp;
        *bp = 2;

        int c = *ap + *bp;

        int *dp;
        {
                int e = 3;
                dp = &e;
```

```
        }
        int f = *dp;

        return &a;
}
```

eCv correctly returns messages warning about uninitialized variables for both lines `*bp = 2;` and `int c = *ap + *bp;`. There exist two more problems: The access to `*dp` is illegal after the nested block as the memory previously occupied by `e` and thus pointed to by `*dp` might be reallocated for other purposes. The same point applies to the function's return statement which returns the address of a local variable as well. Unfortunately, both errors were not detected by eCv.

Out of bounds array access

```
#include <ecv.h>
void outOfBounds(int* array arrA)
{
        int arrB[2] = {0, 1};

        arrA[2];
        arrB[2];

        int i;
        int incorrectBound = 3;
        for (i = 0; i < incorrectBound; ++i)
        writes(i)
        keep(i in 0..incorrectBound)
        decrease(incorrectBound - i)
        {
                arrA[i];
                arrB[i];
        }
}
```

eCv correctly detects all illegal accesses. The statements accessing the array passed via parameter are flagged with warnings that the precondition of the `[]` operator may not be satisfied and the suggestion of a correct precondition that would enable eCv to prove these statements. The first of the statements accessing the statically defined array, `arrB[2];`, is refuted. The other statement is warned about and a suggestion for an additional loop

invariant is provided. In this case, however, the suggestion makes no sense and, if implemented by the user, only leads to further misleading suggestions.

String length

```
#include <ecv.h>
#include <stddef.h>
size_t strlen(const char* array str)
pre(str.lim <= maxof(int))
pre(exists i in 0..str.upb :- str[i] == '\0')
post(result in 0..str.upb)
post(str[result] == '\0')
post(forall i in 0..(result - 1) :- str[i] != '\0')
{
        size_t i;
        for (i = 0; str[i] != '\0'; ++i)
        writes(i)
        keep(i in 0..str.upb)
        keep(forall j in 0..(i - 1) :- str[j] != '\0')
        decrease(str.upb - i)
        {}
        return i;
}
```

As strings in C are just arrays of characters and an array's length is not implicitly known, the convention is to signal the end of a string by a null character. The precondition `exists i in 0..str.upb :- str[i] == '\0'` expresses the fact that the string is null-terminated. This function was easy to verify, the conditions and invariants could be formulated in a natural way.

A problem in an early version of the function was that the postcondition `result in 0..str.upb` was missing. As the constructs used in postconditions need to be proven independently of the program, eCv warned that the `[]` operator's precondition (staying within the array's bounds) might not be satisfied and suggested adding the postcondition stated above. Doing so enabled the proof of all verification conditions.

String comparison

```
#include <ecv.h>
int strcmp (const char* array s1,
```

```
        const char* array s2)
pre(s1.lim <= maxof(int))
pre(s2.lim <= maxof(int))
pre(exists i in 0..s1.upb :- s1[i] == '\0')
pre(exists i in 0..s2.upb :- s2[i] == '\0')
post(result in -1..1)
post((exists i in 0..min(s1.upb, s2.upb) :-
        ((forall j in 0..(i - 1) :- s1[j] == s2[j]) &&
        s1[i] == '\0' && s2[i] == '\0')) =>
        result == 0)
post((exists i in 0..min(s1.upb, s2.upb) :-
        ((forall j in 0..(i - 1) :- s1[j] == s2[j] &&
        s1[j] != '\0') && s1[i] < s2[i])) =>
        result == -1)
post((exists i in 0..min(s1.upb, s2.upb) :-
        ((forall j in 0..(i - 1) :- s1[j] == s2[j] &&
        s1[j] != '\0') && s1[i] > s2[i])) =>
        result == 1)
{
    unsigned int i;
    for (i = 0; s1[i] == s2[i]; ++i)
    writes(i)
    keep(i in 0..s1.upb)
    keep(i in 0..s2.upb)
    keep(forall j in 0..(i - 1) :- s1[j] != '\0')
    keep(forall j in 0..(i - 1) :- s2[j] != '\0')
    keep(forall j in 0..(i - 1) :- s1[j] == s2[j])
    decrease(s1.upb - i)
    decrease(s2.upb - i)
    {
        if (s1[i] == '\0')
        {
            return 0;
        }
    }
    return s1[i] < s2[i] ? -1 : 1;
}

ghost (
integer min(integer a, integer b)
returns(a <= b ? a : b)
```

)

Using the loop invariants from the previous program it was easy to completely verify the loop. Formulating the postconditions for this program took a few tries. However, the quantifier keywords `exists` and `forall` provide the ability to precisely state under which conditions which return value is expected. To keep the ghost variable `i` in the postconditions in the bounds of both `s1` and `s2`, a ghost function that returns the smaller of two numbers needed to be added.

4.2 KeY

KeY is different from the other tools presented in that it was designed as a tool for interactively proving certain properties. Although the capabilities for performing proofs automatically were extended with each version, we still expect that not all examples can be proven automatically. As manual proofs are unfeasible in most industrial settings and present a considerable challenge when using a verification tool in teaching, no manual proofs will be performed in this comparison. If no formulation of an example program can be found that allows automatic verification then the program will be skipped.

KeY version 1.6.0 was used in this comparison. The following list provides a basic overview of some constructs often used in KeY annotations.

`normal_behavior` Specifies behavior for execution of a method in the case that no exceptions occur. `exceptional_behavior` could be used to specify behavior in the case that exceptions occur, but this feature is not used in this comparison. For each method there could be multiple specifications for both normal and exceptional behavior.

`requires` Function precondition

`ensures` Function postcondition

`loop_invariant` Loop invariant

`assignable` Lists variables a loop is permitted to write to

`decreasing` Loop or recursive function variant

`pure` Function that has no side effects

Multiplication

```
public class Multiplication {
/*@ public normal_behavior
  @ requires termA >= 0;
  @ requires termB >= 0;
  @ requires termA * termB <= 2147483647;
  @ ensures \result == termA * termB;
  @*/
public static /*@ pure @*/ int multiply(int termA,
                int termB) {
        int ret = 0;
        /*@ loop_invariant i >= 0 && i <= termB &&
          @      ret == termA * i;
          @ assignable i, ret;
          @ decreasing termB - i;
          @*/
        for (int i = 0; i != termB; i++)
        {
                ret += termA;
        }
        return ret;
}
}
```

The formulation of pre- and postconditions was simple. A minor flaw is the use of the literal `2147483647` as upper bound for `termA * termB`. Normally the maximum value for integers in Java is given by the constant `java.lang.Integer.MAX_VALUE`, but KeY does not seem to support the `java.lang.Integer` class.

The proof of this algorithm halted rather early as KeY at first could not prove that the loop adhered to the loop invariant. After trying different invariants and settings, the cause was discovered to be the accidental omission of `ret` in the loop's `assignable` clause. While this was certainly not KeY's fault, it did not provide much help in finding this trivial mistake. Instead of a notification that a variable is written to which is not declared as assignable, KeY stops when working on the verification condition "body preserves invariant and decreases variant" and returns the following output as current goal:

```
{_termA:=termA || _termB:=termB || i:=0 || ret:=0}
anon_0(_termB, ret, _termA, i),
```

```
 i_0 = 0,
 termB >= 1,
 inReachableState,
 termA >= 1,
 termB <= 2147483647,
 termA <= 2147483647,
 termB * termA <= 2147483647
==>
 {_termA:=termA || _termB:=termB || i:=0 || ret:=termA}
 anon_0(_termB, ret, _termA, i)
```

While even an inexperienced user may conclude that something is wrong with the value of `ret`, the real mistake is not revealed.

As explained in section 3.2.2 on page 47, KeY supports a large number of different settings. One of them determines how arithmetics are handled: Either the program is checked using Java semantics (which dictate modulo arithmetics in case of overflow), or semantics where overflow is ignored (which treats Java's data types as if they could hold unbounded numbers), or semantics where it is verified that overflow does not occur.

The second setting is the easiest to prove but cannot assert program correctness. Using the first setting, correctness with regard to Java arithmetics can be proven, but the program could still exhibit undesired behavior due to overflow. The third setting is the only that rules out any unwanted side effects, but is the hardest to prove.

The multiplication algorithm could be proven using the first and second, but not using the stronger third setting. This could be caused by the fact that the Java Modeling Language specification defines that JML expressions are to be interpreted using Java semantics. Thus, the specification itself is susceptible to arithmetic overflow in this case: Arbitrary values for `termA` and `termB` are valid because multiplying them will always result in a number `<= 2147483647` in Java semantics. An excellent overview on the state of JML arithmetics and a proposal for an adaptation of JML providing more advanced arithmetics is given in [10].

Division

```
public class Division {
/*@ public normal_behavior
  @ requires dividend >= 0 && divisor > 0;
  @ ensures \result == dividend / divisor;
  @*/
```

```
public static /*@ pure @*/ int divide (int dividend,
                int divisor) {
        int remainder = dividend;
        int quotient = 0;
        /*@ loop_invariant dividend == remainder +
          @      quotient * divisor && remainder >= 0;
          @ assignable quotient, remainder;
          @ decreasing remainder;
          @*/
        while (remainder >= divisor) {
                remainder = remainder - divisor;
                quotient++;
        }
        return quotient;
}
}
```

The division algorithm could be proven without difficulty using the setting for ignoring overflow. Unfortunately, no formulation of this method could be found that would have enabled an automatic proof using either of the two stronger arithmetic settings. Correspondence with KeY's developers resulted in the statement that such a proof would be very difficult or even impossible. Thus, the proof obtained can only be considered a coarse estimate on program correctness.

Factorial

The recursive factorial implementation cannot be verified by KeY at the time of writing. Recursive methods are not supported in KeY version 1.6.0 [57].

Index of maximum array element

```
class ArrayMaximum {
/*@ public normal_behavior
  @ requires numbers != null && numbers.length > 0 &&
  @      numbers.length < 2147483647;
  @ ensures \result >= 0 && \result < numbers.length;
  @ ensures (\forall int j; j >= 0 &&
  @      j < numbers.length;
  @      numbers[\result] >= numbers[j]);
```

```
  @*/
public static /*@ pure @*/ int getMaximum(
                int[] numbers) {
        int maximumPosition = 0;
        int maximum = numbers[maximumPosition];
        /*@ loop_invariant
          @ i >= 1 && i <= numbers.length &&
          @ (\forall int j; j >= 0 && j < i;
          @      maximum >= numbers[j]) &&
          @ maximumPosition >= 0 &&
          @ maximumPosition < numbers.length &&
          @ numbers[maximumPosition] == maximum;
          @ assignable i, maximum, maximumPosition;
          @ decreasing numbers.length - i;
          @*/
        for (int i = 1; i < numbers.length; i++) {
                if (numbers[i] > maximum) {
                        maximum = numbers[i];
                        maximumPosition = i;
                }
        }

        return maximumPosition;
}
}
```

Java Modeling Language provides quantifier keywords similar to those already used with Escher C Verifier. The `\forall` keyword is used for the loop invariant and method postcondition. JML also offers a quantifier that maximizes a certain value. This would have enabled the slightly more compact formulation `ensures \result == (\max int j; j >= 0 && j < numbers.length; numbers[j])`, but this keyword is not supported by KeY.

KeY could prove this method without any user interaction.

Faulty pointers

```
class NullPointer {
//@ requires true;
//@ ensures true;
public static void testPointers()
{
```

```
        String a = "1";
        String b = null;
        String c = a.concat(b);
}
}
```

Since the user needs to select a certain task for KeY to perform, this method was annotated with simple pre- and postconditions in order to try proving the postcondition holds.

As Java is more robust and offers less features regarding memory management and pointers, this method only tests behavior when a null pointer is passed to a method. The other errors that the version for eCv contained are not applicable to Java.

The `NullPointerException` that will occur at the statement `String c = a.concat(b);` when running this method is detected by KeY in that it could not prove the postcondition is valid. Unfortunately, there is no clear information about what the problem that was found means. KeY halts the proof with one unproven condition and the following information:

```
pool("1").<created> = TRUE,
 inReachableState
==>
 pool("1") = null,
 {a:=pool("1") ||
   arg0:=null ||
   exc:=null ||
   content(pool("1")):="1"}
   \<{try {method-frame(source=NullPointer)
       : { {
             v_String=a.concat(arg0)@java.lang.String;
           }
           c=v_String;
         }
       }  catch (java.lang.Throwable e) {
         exc=e;
       }
     }\> exc = null
 }
```

While it is certainly possible to conclude from this output that a `NullPointerException` might occur in `String c = a.concat(b);` it cannot be said that this presentation efficiently aids the user in doing so.

Out of bounds array access

```
class ArrayBounds {
//@ requires arrA != null;
//@ ensures true;
public static void outOfBounds(int[] arrA)
{
        int arrB[] = {0, 1};
        int element;
        element = arrA[2];
        element = arrB[2];

        int incorrectBound = 3;
        /*@ loop_invariant i >= 0 &&
          @      i <= incorrectBound;
          @ assignable i, element;
          @ decreasing incorrectBound - i;
          @*/
        for (int i = 0; i < incorrectBound; ++i)
        {
                element = arrA[i];
                element = arrB[i];
        }
}
}
```

KeY is able to detect all problematic array accesses. The information is again presented in a quite intricate way. For example, the goal that KeY stops at because of `element = arrA[2];` is given as:

```
 java.lang.ArrayIndexOutOfBoundsException.<nextToCreate> >= 0,
 jint[].<nextToCreate> >= 0,
 arrA.length >= 3,
 inReachableState,
 arrA.<created> = TRUE
==>
 arrA = jint[]::<get>(jint[].<nextToCreate>),
 arrA = null
```

The output regarding the statements in the loop body is about three times as long.

String length

```
class StringLength {
/*@ public normal_behavior
  @ requires str != null && str.length > 0 &&
  @      str.length < 2147483647 &&
  @      (\exists int i; i >= 0 && i < str.length;
  @      str[i] == '0');
  @ ensures str[\result] == '0' &&
  @      (\forall int i; i >= 0 && i < \result;
  @      str[i] != '0');
  @*/
int getStringLength(char[] str)
{
        int i;
        /*@ loop_invariant
          @ i >= 0 && i <= str.length &&
          @ (\forall int j; j >= 0 && j < i;
          @              str[j] != '0');
          @ assignable i;
          @ decreasing str.length - i;
          @*/
        for (i = 0; str[i] != '0'; ++i)
        {}

        return i;
}
}
```

While Java's `java.lang.String` class internally stores string values in character arrays (which is a similarity to string handling in C), the arrays are normally not visible on the surface and neither does the developer have to be concerned with topics such as string termination using null characters. We still implemented C's `strlen` function as an analogous Java method to test KeY's abilities for reasoning about more complex specifications.

The annotations could be formulated in a natural way using the quantifier keywords. KeY could prove that the method adheres to the postconditions without any problems.

String comparison

```
class SimpleString {
/*@ public invariant value.length < 2147483647 &&
  @ count >= 0 && count < 2147483647 &&
  @ offset >= 0 && offset < 2147483647 &&
  @ offset + count < 2147483647 &&
  @ value.length >= offset + count;
  @*/
private char value[];
private int offset;
private int count;

//@ requires anotherString != null;
//@ ensures true;
public int compareTo(SimpleString anotherString) {
        int len1 = count;
        int len2 = anotherString.count;
        int n = min(len1, len2);
        char v1[] = value;
        char v2[] = anotherString.value;
        int i = offset;
        int j = anotherString.offset;
        if (i == j) {
                int k = i;
                int lim = n + i;
                //@ loop_invariant k >= 0 && k <= lim;
                //@ assignable k;
                //@ decreasing lim - k;
                while (k < lim) {
                        char c1 = v1[k];
                        char c2 = v2[k];
                        if (c1 != c2) {
                                return c1 - c2;
                        }
                        k++;
                }
        } else {
                /*@ loop_invariant n >= 0 &&
                  @ n <= min(len1, len2) &&
                  @ i - offset ==
```

```
                    @       min(len1, len2) - n &&
                    @ j - anotherString.offset ==
                    @       min(len1, len2) - n;
                    @*/
                  //@ assignable n, i, j;
                  //@ decreasing n;
                  while (n-- != 0) {
                          char c1 = v1[i++];
                          char c2 = v2[j++];
                          if (c1 != c2) {
                                  return c1 - c2;
                          }
                  }
          }
          return len1 - len2;
}

//@ requires true;
//@ ensures \result == (n1 < n2 ? n1 : n2);
private int min(int n1, int n2) {
          return n1 < n2 ? n1 : n2;
}
}
```

For the task of comparing two strings no port of C's `strcmp()` function to Java was used, but instead it was attempted to verify the analogous `java.lang.String.compareTo()` method of the standard Java Class Library. To simplify the situation for this comparison, a downscaled string class containing only the fields relevant to this method was created. The `min()` method was added instead of using the one in `java.lang.Math` as this class is not part of Java Card.

The first attempt consisted of trying to prove that no exceptions occurred in this method by specifying the postcondition as `ensures true`. The correctness of the array accesses in the second loop could not be proven at first by using the invariants `i >= offset && i <= offset + min(len1, len2)` regarding `i` (and analogous invariants regarding `j`). One of KeY's developers provided help by suggesting to add invariants that more directly relate the array index variables `i` and `j` to the loop counter `n`. KeY was then able to prove the absence of unexpected behavior.

Unfortunately, full verification of the method did not succeed. Various postconditions were tested. Even after lengthy attempts, the only postcon-

dition that could be proven was one specifying that the method returns 0 in the case that the strings are identical.

It is suspected that the loop invariants might not be strong enough to allow proving the other cases. For example, for the second loop an additional invariant like `(\forall int k; k >= 0 && k < (\old(n) - n); v1[this.offset + k] == v2[anotherString.offset + k])` would be an obvious choice. `\old(n)` refers to the value of `n` before the first loop iteration. The invariant states that at all indices that were already passed both arrays contain the same character. Unfortunately, we could not get KeY to automatically verify this or similar invariants and finally gave up. The lack of simple and concise information regarding the facts which KeY was unable to prove was a large obstacle for working on this algorithm.

4.3 VCC

The VCC team provides no distinction between unstable or release versions but just offers automated daily builds for downloading. The version used in this comparison is 2.1.40918.0 from September 18, 2011.

The following list provides a basic overview of some constructs often used in VCC annotations.

`requires` Function precondition

`ensures`, `returns` Function postcondition

`invariant` Loop invariant

`==>` Material implication

Multiplication

```
#include <vcc.h>
#include <limits.h>
unsigned int multiply(unsigned int termA,
        unsigned int termB)
_(requires termA * termB < UINT_MAX)
_(returns termA * termB)
{
        unsigned int ret = 0;
        unsigned int i;
        for (i = 0; i != termB; ++i)
```

```
        _(invariant i >= 0)
        _(invariant i <= termB)
        _(invariant ret == termA * i)
        {
                _(assert lemma(termA, termB, i))
                ret += termA;
        }

        return ret;
}

_(ghost _(pure) bool lemma(\integer termA,
        \integer termB, \integer i)
  _(requires termA * termB < UINT_MAX)
  _(requires termA >= 0 && i < termB)
  _(ensures termA * i + termA < UINT_MAX)
  _(returns \true)
{
  _(assert i + 1 <= termB)
  _(assert termA * (i + 1) <= termA * termB)
  return \true;
})
```

VCC could verify most aspects of this program, the only fact that could not be proven was that `ret += termA;` in the loop body did not cause arithmetic overflow. Additional assertions were then added to the loop body to find out what exactly VCC was unable to prove. One of the assertions that did not verify was `termA * (i + 1) < UINT_MAX)`, even though at this point it was already proven that `termA * termB < UINT_MAX)` and `i < termB`. When then adding `termA < 65500 && termB < 65500` as precondition (for which `termA * termB` is just smaller than `UINT_MAX` using default limits), the whole function was proven, however this unnecessarily constrained the function's scope.

One of VCC's developers gave the hint of using a ghost lemma function that details the steps needed to prove `termA * (i + 1) < UINT_MAX)` and guides the prover to the correct result. While this method indeed enables verification of the function, it seems rather complex and lengthy for a simple task.

It is not clear how VCC checks for loop termination: No information regarding this topic could be found in the tool's documentation. By chance it was found out that there exists a `decreases` keyword in VCC's annotation

language, but regardless of what was specified as decreasing, no influence on the prover's results could be detected.

Similar to eCv, VCC automatically treats variables in specifications as if they were typed in an unbounded type, in this case `\integer`.

Division

```
#include <vcc.h>
#include <limits.h>
unsigned int divide(unsigned int dividend,
        unsigned int divisor, unsigned int* remainder)
_(requires dividend >= 0 && divisor > 0)
_(writes remainder)
_(ensures dividend == *remainder + divisor *
        \result && *remainder < divisor)
{
        unsigned int lRemainder = dividend;
        unsigned int quotient = 0;

        while (lRemainder >= divisor)
        _(invariant dividend == lRemainder +
                quotient * divisor)
        {
                lRemainder = lRemainder - divisor;
                quotient++;
        }
        *remainder = lRemainder;
        return quotient;
}
```

In this function the loop could be verified without problems. Unfortunately it could not be shown that the program satisfied the postcondition stated simply as `\result == dividend / divisor`. VCC's tutorial contains an example for verifying division [16, p. 18] and this is where the decisive hint was found. It seems that in order to show that the postcondition is satisfied, the value of the division's remainder has to be taken into consideration.

The additional parameter `unsigned int* remainder` was added to the function signature. Using this pointer it is possible to formulate the postcondition very similar to the loop invariant. While this version of the program was completely verified, requiring the user to add an additional parameter

and to write the postcondition in this rather complex form is unsatisfactory.

Factorial

```
#include <vcc.h>
#include <limits.h>
unsigned int factorial(const unsigned int num)
_(requires num >= 0)
_(requires factorialGhost(num) <= UINT_MAX)
_(ensures \result >= 1)
_(ensures \result == factorialGhost(num))
{
        if (num == 0)
        {
                return 1;
        }
        else
        {
                return (num * factorial(num - 1));
        }
}

_(ghost _(pure) \integer factorialGhost(\integer num)
        _(requires num >= 0)
        _(returns num == 0 ? 1 :
                num * factorialGhost(num - 1))
)
```

The state of support for recursive functions in VCC could not be established without any doubt: Recursion is supported in general, the tutorial contains a recursive implementation for example [16, p. 14]. Unfortunately no background information regarding recursion is provided. There is nothing in the documentation about whether or how VCC checks for termination of recursive function calls and whether they may be freely used in specifications. It could only be found out that, in order to call any functions from specifications, the functions called need to be annotated as `_(pure)`. Such functions "[...] are not allowed to allocate memory, and can write only to local variables" [16, p. 62].

As no definite information was found, verification was attempted with the almost trivial specifications `_(requires num >= 0)`, `_(requires num > 0 ==> num * factorial(num - 1) <= UINT_MAX)` and `_(ensures \result`

`== (num == 0 ? 1 : num * factorial(num - 1))`. VCC quickly output a rather generic error message that the call `factorial(num - 1)` in the precondition could not be verified but continued running for a few minutes without further results.

Therefore, verification was tried with a new version using a ghost function. In this case VCC claims that the verification succeeded and no errors are returned. Regrettably, there is a warning `[possible unsoundness]: cycle in pure function calls: factorialGhost -> factorialGhost`. How severely this warning affects the validity of the verification result is unknown. In the discussion forum on VCC's website a post from 2010 was found that states that for recursive functions it is the user's responsibility to make sure recursion is finite. If the warning only pertains to this fact, the result would still be valid.

Index of maximum array element

```
#include <vcc.h>
unsigned int getMaximum(const int* numbers,
        const unsigned int numbersSize)
_(requires \thread_local_array(numbers, numbersSize))
_(requires numbersSize >= 1)
_(ensures \result >= 0 &&
        \result <= (numbersSize - 1))
_(ensures \forall unsigned int a; a < numbersSize ==>
        numbers[\result] >= numbers[a])
{
        unsigned int maximumPosition = 0;
        int maximum = numbers[maximumPosition];
        unsigned int i;
        for (i = 1; i != numbersSize; ++i)
        _(invariant i >= 1 && i <= numbersSize)
        _(invariant maximumPosition >= 0 &&
                maximumPosition < numbersSize)
        _(invariant \forall unsigned int a; a < i ==>
                maximum >= numbers[a])
        _(invariant numbers[maximumPosition] ==
                maximum)
        {
                if (numbers[i] > maximum)
                {
```

```
                        maximum = numbers[i];
                        maximumPosition = i;
                }
        }
        return maximumPosition;
}
```

This program could be verified very quickly. As VCC supports multi-threaded programs, all accesses to data structures like arrays need to be annotated in a way that allows to exclude the possibility of another thread writing to the same structure. The precondition `\thread_local_array(numbers, numbersSize)` specifies the fact that the array belongs to the local thread. Implicitly it also specifies that the array pointer must not be null.

Similar to eCv and KeY, keywords for reasoning with quantifiers are available, `\forall` and `\exists`. VCC provides no ghost function for determining the maximum array element. Therefore, the postcondition cannot be formulated as simple as is possible in the eCv version, but it can still be stated in a concise way.

Faulty pointers

```
#include <vcc.h>
int* testPointers()
{
        int a = 1;
        int* ap = &a;

        int* bp;
        *bp = 2;

        int c = *ap + *bp;

        int *dp;
        {
                int e = 3;
                dp = &e;
        }
        int f = *dp;

        return &a;
}
```

VCC detects all faulty pointer accesses in this function, although the messages provided are not as precise as those that eCv provides: For example, regarding the line `*bp = 2;` the message `Assertion 'bp is writable' did not verify` is output. For the lines containing the addition and the assignment to `f` VCC returns an analogous message concerning the assertion `bp` [or `dp`] `is thread local`. In contrast to eCv, the error in `int f = *dp;` is detected. Unfortunately, like eCv, VCC could not detect the error in the return statement where the address of a local variable is returned.

Out of bounds array access

```
#include <vcc.h>
void outOfBounds(int* arrA)
{
        int arrB[2] = {0, 1};

        arrA[2];
        arrB[2];

        int i;
        int incorrectBound = 3;
        for (i = 0; i < incorrectBound; ++i)
        _(invariant i >= 0 && i <= incorrectBound)
        {
                arrA[i];
                arrB[i];
        }
}
```

In this program all out of bounds accesses are detected, however they are only flagged using the generic message telling that it could not be verified that the respective variable is thread-local. In addition to the errors regarding the invalid accesses VCC also returns warnings that an expression like `arrA[2];` has no side effect and that an operation with side effect is expected.

String length

```
#include <vcc.h>
size_t strlen(const char* str _(ghost size_t strSize))
_(requires \thread_local_array(str, strSize))
_(requires \exists unsigned int i; i < strSize &&
```

```
        str[i] == '\0')
_(ensures str[result] == '\0')
_(ensures \forall unsigned int i; i < \result ==>
                str[i] != '\0')
{
        size_t i;
        for (i = 0; str[i] != '\0'; ++i)
        _(invariant i >= 0 && i < strSize)
        _(invariant \forall unsigned int j; j < i ==>
                str[j] != '\0')
        {}

        return i;
}
```

VCC provides no ghost fields for reasoning about array bounds which posed a problem for devising correct specifications for this function. After a few unsuccessful attempts, a ghost parameter that specifies the array's size was added to the function signature. With the help of this parameter it was possible to completely verify the function. There is a downside to this approach though: When calling `strlen` in a program that is to be verified a value has to be supplied for the ghost parameter `strSize`, which is quite inconvenient. In contrast, the ghost fields that for example eCv provides enable verification without the need for introducing any ghost parameters for expressing information regarding array bounds.

All other annotations could be stated in a straightforward manner.

String comparison

```
#include <vcc.h>
int strcmp(const char* s1, _(ghost size_t s1Size)
        const char* s2 _(ghost size_t s2Size))
_(requires \thread_local_array(s1, s1Size))
_(requires \thread_local_array(s2, s2Size))
_(requires \exists unsigned int i;  i < s1Size &&
        s1[i] == '\0')
_(requires \exists unsigned int i;  i < s2Size &&
        s2[i] == '\0')
_(ensures \result >= -1 && \result <= 1)
_(ensures (\exists unsigned int k;
        (k < min(s1Size, s2Size) &&
```

```
        (\forall unsigned int j;
        j < k ==> s1[j] == s2[j]) &&
        (s1[k] == '\0' && s2[k] == '\0'))) ==>
        \result == 0)
_(ensures (\exists unsigned int k;
        (k < min(s1Size, s2Size) &&
        (\forall unsigned int j;
        j < k ==> s1[j] == s2[j] && s1[j] != '\0') &&
        (s1[k] < s2[k]))) ==> \result == -1)
_(ensures (\exists unsigned int k;
        (k < min(s1Size, s2Size) &&
        (\forall unsigned int j;
        j < k ==> s1[j] == s2[j] && s1[j] != '\0') &&
        (s1[k] > s2[k]))) ==> \result == 1)
{
        unsigned int i;
        for (i = 0; s1[i] == s2[i]; ++i)
        _(invariant i >= 0 && i < s1Size)
        _(invariant i >= 0 && i < s2Size)
        _(invariant \forall unsigned int j; j < i ==>
                s1[j] != '\0')
        _(invariant \forall unsigned int j; j < i ==>
                s2[j] != '\0')
        _(invariant \forall unsigned int j; j < i ==>
                s1[j] == s2[j])
        {
                if (s1[i] == '\0')
                {
                        return 1;
                }
        }

        return s1[i] < s2[i] ? -1 : 1;
}

_(ghost _(pure) size_t min(size_t a, size_t b)
_(ensures \result == (a <= b ? a : b))
)
```

Taking the implementation for eCv and the knowledge about the need of introducing array size ghost parameters the function could be implemented

for VCC with little effort. The annotations for this function could be stated using the quantifier keywords `\exists` and `\forall` in a precise way. VCC has a more verbose syntax for quantifiers than eCv which makes the pre- and postconditions longer and a bit harder to read in this version.

4.4 VeriFast

In section 3.2.4 on page 57 we expressed the concern that VeriFast seems to require more elaborate and lower level annotations than the other tools. This suspicion turned out to be true, at least for the algorithms compared here. VeriFast offers the possibility to write powerful lemma functions to express different facts. While this can be very useful for advanced users, it causes a steep learning curve for beginners. Therefore, for almost all algorithms in this comparison the help of VeriFast's authors was necessary.

VeriFast supports the verification both of programs written in C and in Java. Only the verification of C programs was analyzed in this comparison. The version used was 11.9.19.

The following list provides a basic overview of some constructs often used in VeriFast annotations.

`requires` Function precondition

`ensures` Function postcondition

`invariant` Loop invariant

`decreases` Loop variant

`produce_limits(var)` Generates assumptions on arithmetic properties of `var` which is not done by default [53, p. 18]

`cons(z, zs0)` Used with list data types. Splits a list into an element `z` and the list of further elements `zs0`

Multiplication

```
/*@
lemma void lemma_mult(int x, int y, int z)
requires 0 <= x &*& y <= z;
ensures x * y <= x * z;
{
        for (int i = 0; i < x; i++)
```

```
        invariant i * y <= i * z &*& i <= x;
        decreases x - i;
        {}
}
@*/

int multiply(int termA, int termB)
/*@ requires termA >= 0 && termB >= 0
        && termA * termB < INT_MAX;
@*/
//@ ensures result == (termA * termB);
{
        //@ produce_limits(termA);
        //@ produce_limits(termB);
        int ret = 0;
        int i;
        for (i = 0; i != termB; ++i)
        /*@ invariant i >= 0 && i <= termB &&
                ret == termA * i;
        @*/
        //@ decreases termB - i;
        {
                //@ lemma_mult(termA, i + 1, termB);
                ret += termA;
        }

        return ret;
}
```

VeriFast does not take differences between platforms or compilers into account when checking arithmetic properties of data types. Instead, it always assumes that the `int` type is signed with a size of 32 bits. No information could be found regarding the possibility of reasoning using limits of unsigned types, and thus only signed types were used in the functions verified with VeriFast.

For a first version of this program without the lemma function it could not be proven that no arithmetic overflow occurred in the loop body. For debugging purposes some additional assertions were added before the return statement: `i == termB`, `ret == termA * i` and `ret == termA * termB`. The first two of these assertions could be proven. Strangely, the third one, and thus also the function postcondition, was not verified.

One of VeriFast's developers advised not to use Z3 but instead the Redux solver for this function, as Z3 apparently often has difficulties with certain arithmetic properties. Using Redux all assertions were proven, but the overflow warning for the loop body still appeared. VeriFast's developer kindly provided a lemma function that helps showing that multiplication preserves the smaller-than relation. It therefore enables the proof that when `termA * termB < INT_MAX` and `i <= termB` it is also valid that `termA * i < INT_MAX`. This results in the fact that no arithmetic overflow can occur in the loop body.

The conclusion is similar to multiplication in VCC: While a complete automatic proof is possible the way to get there was rather complicated.

Division

```
/*@
lemma void lemma_mult(int a, int b)
requires 0 <= a && 1 <= b;
ensures a <= a * b;
{
        for (int i = 1; i < b; i++)
        invariant i <= b && a <= a * i;
        decreases b - i;
        {
        }
}
lemma void axiom_div(int a, int b, int q, int r);
requires 0 <= a && 1 <= b && 0 <= q && 0 <= r &&
        a == q * b + r && r < b;
ensures q == a / b && r == a % b;
@*/

int divide(int dividend, int divisor)
//@ requires dividend >= 0 && divisor > 0;
//@ ensures result == dividend / divisor;
{
        //@ produce_limits(dividend);
        //@ produce_limits(divisor);
        int remainder = dividend;
        int quotient = 0;
        while (remainder >= divisor)
```

```
        /*@ invariant dividend == remainder +
                quotient * divisor &&
                remainder >= 0 && quotient >= 0 &&
                remainder <= dividend -
                quotient * divisor;
        @*/
        //@ decreases remainder;
        {
                remainder = remainder - divisor;
                //@ lemma_mult(quotient, divisor);
                quotient++;
        }

        /*@ axiom_div(dividend, divisor, quotient,
                remainder);
        @*/
        return quotient;
}
```

For the division algorithm help from VeriFast's developers was necessary again. Two difficulties were encountered: Related to the problem in the previous algorithm a lemma had to be added that specifies that, in the allowed variable ranges, `quotient` is always smaller than or equal to `quotient * divisor` – and thus also smaller than or equal to `dividend`.

VeriFast provides no full axiomatization for division. It is therefore necessary to add an axiom in the form of a lemma function without body that defines division in a way that corresponds to the implemented algorithm. Only with this axiom the postcondition could be verified.

Factorial

The situation becomes even more complicated when trying to prove the factorial algorithm. Again, VeriFast's developers provided help by supplying extensive annotations for this task. Unfortunately, the code is about one hundred lines long and quite complicated. Due to the length it is not reproduced here, but can be found in appendix A on page 115. It consists of various lemmas that show different arithmetic aspects and fixpoint functions that deal with specifying properties of the factorial function itself.

While VeriFast was able to automatically proof the program, the complexity of the annotations is very high. The user is required to have an excellent understanding of VeriFast's theoretic aspects and annotation language in

order to being able to devise such enormous annotations by himself.

Index of maximum array element

```
//@ #include "listex.h"
/*@
fixpoint bool ge(int x, int y) { return x >= y; }

lemma void ge_max(int x, int y, list<int> zs)
requires x >= y &*& forall(zs, (ge)(y)) == true;
ensures forall(zs, (ge)(x)) == true;
{
        switch (zs) {
                case nil:
                case cons(z, zs0):
                        ge_max(x, y, zs0);
        }
}
@*/
int getMaximum(int* numbers, int numbersSize)
    /*@ requires array<int>(numbers, numbersSize,
        sizeof(int), integer, ?numbersList) &*&
        numbersSize > 0 &*& numbersList != nil &*&
        length(numbersList) == numbersSize;
    @*/
    /*@ ensures array<int>(numbers, numbersSize,
        sizeof(int), integer, numbersList) &*&
        result >= 0 &*& result < numbersSize &*&
        forall(numbersList, (ge)(nth(result,
                numbersList))) == true;
    @*/
{
        /*@ switch (numbersList) {
                case nil: case cons(h, t): }
        @*/
        int maximumPosition = 0;
        int maximum = numbers[maximumPosition];
        int i;
        for (i = 1; i != numbersSize; ++i)
        /*@ invariant array<int>(numbers, numbersSize,
```

```
        sizeof(int), integer, numbersList) &*&
        i >= 1 &*& i <= numbersSize &*&
        maximumPosition >= 0 &*&
        maximumPosition < i &*&
        nth(maximumPosition, numbersList) ==
            maximum &*&
        forall(take(i, numbersList),
            (ge)(nth(maximumPosition,
            numbersList))) == true;
    @*/
    //@ decreases numbersSize - i;
    {
    //@ take_plus_one(i, numbersList);
    /*@
    if (nth(i, numbersList) >= maximum) {
            ge_max(nth(i, numbersList),
            maximum, take(i, numbersList));
            forall_append(take(i, numbersList),
                    cons(nth(i, numbersList),
                            nil),
                    (ge)(nth(i, numbersList)));
    } else {
            forall_append(take(i, numbersList),
                    cons(nth(i, numbersList),
                            nil),
                    (ge)(maximum));
    }
    @*/
            if (numbers[i] >= maximum)
            {
                    maximum = numbers[i];
                    maximumPosition = i;
            }
    }

    return maximumPosition;
}
```

The annotations for this algorithm employ some of the powerful aspects of VeriFast's annotation language. They also show that in some cases VeriFast requires more elaborate information than the other verification tools.

To successfully verify this algorithm the user needs to introduce a fixpoint function `ge` that returns whether or not one number is greater than or equal to another number. The lemma `ge_max` shows the fact that when it has already been concluded that a number `y` fulfills the `ge` relation for all elements of a list, and a number `x` is not smaller than `y`, then `x` also fulfills the `ge` relation for all elements. The lemma also demonstrates that VeriFast internally handles the contents of arrays as lists of generic data types and provides various predefined utility functions, for example the `forall` fixpoint that is used here to reason about all elements of a list.

Pre- and postconditions for the algorithm itself are rather straightforward. The `array<int>` clause expresses that the pointer `numbers` points to an array of length `numbersSize` of `integer` elements and that the array contents can be accessed in annotations by using the `numbersList` variable. `nth(result, numbersList)` returns the element with index `result` from the `numbersList`.

In the loop invariant `take(i, numbersList)` returns a list containing the first `i` elements of `numbersList`. The annotations discussed to this point are quite similar to those required by the other tools that were compared.

Unfortunately, the situation gets more complex. The switch statement in the annotation right at the beginning of the function body deserves an explanation, which was kindly given by VeriFast's developers. The `take` fixpoint in the loop invariant is defined by recursion on the list parameter. Out of performance considerations VeriFast does not automatically perform case splitting. However, case splitting is necessary in this scenario: In order to reason on `take`, it needs to be discerned whether or not the list parameter is `nil`. The switch statement causes VeriFast to symbolically execute the function body in two paths, one where `numbersList` equals `nil` and one where it does not. VeriFast can discard the prior path immediately because the function preconditions rule this situation out and can continue just with the latter path. Still, in order to reach this conclusion the case split has to be introduced manually.

The predefined `take_plus_one` axiom in the loop body defines the fact that taking the first `i + 1` elements of a list yields an identical result as taking the first `i` elements and appending the element at index `i`. This, together with the following annotation, is required to enable the proof of the `forall` statement in the loop invariant, which otherwise fails. The conditional statement in the annotation that follows is then providing details on the loop body's intended semantics.

While it is positive that the algorithm can be completely proven in this way, the requirement to duplicate many program statements in the annotation language is unsatisfactory. Also, even though most users will

probably be able to comprehend the annotations with relatively small effort by looking at the definitions of the predefined lemma functions, writing them requires a thorough understanding of VeriFast's methodology. In addition, no documentation on the currently implemented state of array support is available at the time of writing. VeriFast's developers mentioned that work is under way to improve and simplify array handling. Therefore extensive help was needed again from VeriFast's developers, who provided the fixpoint and lemma functions as well as the annotations inside the loop body.

Faulty pointers

```
int* testPointers()
//@ requires true;
//@ ensures true;
{
        int a = 1;
        int* ap = &a;

        int* bp;
        *bp = 2;

        int c = *ap + *bp;

        int *dp;
        {
                int e = 3;
                dp = &e;
        }
        int f = *dp;

        return &a;
}
```

Empty pre- and postconditions needed to be added to this method because VeriFast enforces that all non-fixpoint functions have a method contract.

VeriFast detects all faulty pointers in this function. Unfortunately, the messages returned contain only a similar level of detail as those returned by VCC. Still, the output is reasonably suitable for finding errors, especially when compared to KeY. From the first erroneous line the message `No matching heap chunks: integer(bp, _)` results. The other two errors

are reported as a similar `No matching pointsto chunk` message.

Like Escher C Verifier and VCC, VeriFast does not detect the problem in the return statement.

Out of bounds array access

```
void outOfBounds(int* arrA)
//@ requires true;
//@ ensures true;
{
        int arrB[2];
        arrB[0] = 0;
        arrB[1] = 1;
        //@ assert array<int>(?a, _, _, _, _);

        arrA[2];
        arrB[2];

        int i;
        int incorrectBound = 3;
        for (i = 0; i < incorrectBound; ++i)
        /*@ invariant array<int>(a, incorrectBound,
                sizeof(int), integer, _) &*&
                i >= 0 && i <= incorrectBound;
        @*/
        //@ decreases incorrectBound - i;
        {
                arrA[i];
                arrB[i];
        }
}
```

Checking this function first resulted in the peculiarity that VeriFast returned a syntax error at the array initialization `int arrB[2] = {0, 1}`. Omitting the explicit declaration of the array length, which is valid as well, even results in an "internal error". Thus, array initialization was modified as in the code above.

This program reflects the work-in-progress state of VeriFast's array support. The `array` clause only supports pointers and no locally defined arrays. As a workaround, a new pointer `a` is introduced which automatically takes the address of the array defined before, `arrB`. The pointer can then be

used to enable access to the array in the loop body.

VeriFast detects all out of bounds array accesses in this function. The messages returned are `No matching array chunk` outside the loop and `No matching heap chunks: array(arrB_addr, 3, 4, integer, _)` inside the loop.

String length, string comparison

Since these two algorithms depend heavily on array handling which, as already mentioned above, is currently in the process of being adapted, we refrained from implementing them in VeriFast. `strlen` is discussed in the tutorial [53, p. 50]. An implementation is contained in the collection of solutions to the tutorial examples that is included in the VeriFast distribution. This implementation however is based on an older array handling paradigm. A new look on possible implementations of `strlen` and `strcmp` could be worthwhile when VeriFast's array support has stabilized and proper documentation is available.

4.5 Summary

The comparison was conducted with great care. Still, it is possible that some problems could have been solved with one or another tool more efficiently than the way it was done here. To become fully proficient with a tool the investment of a lot of time and work is necessary and thus not all possibilities of all tools could be explored in this work. Even so, a very positive insight comes from this comparison: In view of the features and accessibility the tools analyzed in [37] provided, it can be said that the situation improved considerably.

Today, there are a number of tools that support formal verification of regular, widely-used programming languages, or at least of substantial subsets. Each of those that were compared here showed its own strengths and weaknesses.

Escher C Verifier

The feature of suggesting some types of missing annotations is unique to eCv. Although they only appear in some cases, the suggestions can have a large impact on productivity. In a number of simple cases where the developer inadvertently forgot necessary function preconditions or loop invariants, eCv's hints can reduce the time spent on debugging. This may not apply

to very experienced users, as they can probably find missing annotations quickly by themselves in those simple cases where suggestions are available. Still, for users just starting work with formal verification the hints given can greatly lower the obstacles encountered right at the beginning. In some scenarios experienced users can profit from suggestions too, as they may point to errors in existing annotations.

Thus, suggestions are helpful most of the time, but they can also be misleading. When testing the detection of out of bounds array accesses eCv suggested annotations that only complicated the situation and could never lead to a proof, as the program simply was incorrect. In some cases it was observed that suggestions were made that were syntactically incorrect. Examples include the suggestion of a precondition that contains a recursive call to a C function (which is not allowed in eCv by design), as well as an annotation that contained an inline conditional operator with a missing else branch.

Another unique feature in Escher C Verifier is that the user can easily configure the properties of the compiler that is used in a project, most importantly the sizes of different data types. The verifier is thus enabled to provide correct results almost regardless of compiler features or target platform.

A further positive aspect is that of all tools eCv provides the most detailed and understandable result messages to the user. Therefore, in many cases, the user is led directly to the relevant mistake in the program or specification and does not have to find it by trial and error. The user can also configure the level of detail provided in error and success messages. In most cases eCv offers different formulations of each fact that could not be proven and often also uses case distinctions for different variable ranges to aid the user in retracing the process that led to the verification failure. While these detailed messages are not perfect and may mislead the user in some cases, in general they help tremendously in getting positive results rather easily.

Unfortunately, Escher C Verifier does not support the verification of multi-threaded programs. While multi-threading was no part of this practical comparison, it is widely used in general-purpose and server software and might thus rule out the use of eCv in some industrial environments. This is less a factor for Escher's core target group that deals with security critical, often embedded, software. Similarly, the suitability for teaching formal methods does not depend on multi-threading support either. The only additional negative technical aspect observed was that eCv did not detect one pointer-related error which VCC and VeriFast did notice.

In general, Escher C Verifier presents itself as a very accessible tool

that enables a novice user, regardless of being a software engineer in an industrial environment or a student attending a course, to have a positive experience when first dealing with practical aspects of formal verification. More advanced users are also supported in a helpful way by the precise output provided and the possibility of configuring architectural properties of the target platform.

KeY

KeY differs from the other tools in that it only supports programs written in Java Card (with some extensions) and has a rather long development history. It was already available when the comparison in [37] was performed, and we will therefore especially look at the progress KeY has made since then.

In the previous comparison, a multitude of very complex interactions was required for every single algorithm that was analyzed. This was necessary even though in some cases postconditions had to be formulated much weaker than intended, because there was no way of precisely specifying more complex behavior using Object Constraint Language. The situation today is quite different: The possibility of using Java Modeling Language in annotations greatly enhances the number and quality of facts that can be expressed. Unfortunately, JML also introduces the problem that arithmetics are handled in annotations according to Java semantics instead of providing unbounded data types that would further improve specifications.

KeY's features and tactics for performing automatic proofs seem to have improved vastly. Most aspects of the algorithms could be proven automatically today. Notable exceptions are proofs for ruling out arithmetic overflow in the methods for multiplication and division (which is arguably impossible due to the use of JML) as well as for the postconditions of the string comparison method.

Unfortunately, even though great progress has been made, KeY can still present a major challenge to the user. By design KeY is still a tool meant for interactive verification. Therefore, when KeY cannot continue a proof by itself, it does not return results that are easily comprehensible, but just the description of the next goal that needs to be proven. The user must interpret the output by himself and find out whether there is an error in the specification, the program, or whether a manual proof is necessary. Some hints are provided, for example the name of the current branch of the proof tree, or the possibility to view the Java execution path that led to the current goal, but these features cannot replace the output of precise and helpful result messages. At the very least, in common cases like

some syntactical errors in annotations, or the omission of certain required annotations, KeY should be able to clearly state what the problem is to avoid tedious debugging.

A further possible improvement could be to introduce support for JML assertion statements in KeY. In the other tools it was found that the debugging process could in some cases be made faster and easier by adding assertions to the program to find out at which points in the code which facts could be deduced by the prover. KeY only supports standard Java assertions, which provide much fewer possibilities than JML assertions.

KeY positively surprised in this comparison because of its improved abilities to perform proofs automatically and the support of the powerful Java Modeling Language. Still, caused by the fact that it is aimed at interactive proofs and in certain cases requires them, it cannot be recommended for industrial environments. Using KeY in teaching could be more interesting, because – in contrast to the other tools – it allows presenting the principles of proof construction using practical examples. It is expected that a typical university course that also discusses theoretical aspects does not provide enough time for students to become proficient using KeY in all situations. Therefore, the examples presented in courses would need to be restricted to simple ones, and enough staff to support students in the construction of manual proofs would be necessary.

VCC

VCC's history of being used for formal verification of parts of Microsoft's hypervisor Hyper-V is reflected in that VCC appeared as a stable, mature tool in this comparison, despite the fact that the reimplementation that is currently being performed is not fully finished and that documentation for the new syntax is not yet complete. The tutorial document already available for the new VCC version provided answers to most questions that came up when working on the example algorithms.

VCC, similar to Escher C Verifier, was able to completely prove all algorithms, with the only exception of the warning regarding "possible unsoundness" in the recursive factorial specification. However, in comparison to eCv the annotations required were more elaborate and less convenient. Examples for this concern are found in the arithmetic algorithms: Multiplication required a lengthy lemma function for ruling out arithmetic overflow. To prove the division algorithm, it was necessary to introduce an additional function parameter and state the postcondition analogously to the loop invariant instead of simply being able to use the division operator.

A similar criticism applies to the functions for determining the length of a string and comparing two strings. Since no predefined ghost fields exist that could be used for specifying properties of arrays, ghost parameters have to be introduced that specify the length of arrays for use in annotations. The syntax for quantifiers is similar to JML and thus more verbose than in eCv, mostly due to the fact that the bounds for the quantifying variable cannot be stated as simply as is possible there.

It was disappointing not to find clear information regarding how termination is ensured for loops or recursive function calls. At least for recursion, it seems that termination is currently not checked (as mentioned in the description of the factorial algorithm), which makes VCC less useful in cases where recursion is needed.

Another area that showed room for some improvement is the level of detail of the result messages provided to the user. They are far more helpful than the output of KeY, but in many cases not as specific as those provided by eCv. This may make the use of VCC less convenient than eCv, especially in larger projects where debugging is more complex.

VCC offers the most extensive support for existing development workflows of all tools analyzed here: eCv does not offer any integration into standard IDEs. Developers of KeY provide a plug-in for Eclipse, but it has only rudimentary features. VeriFast includes a simple GUI with code editor, but this cannot compete with a full IDE. In contrast, VCC can be integrated into the Visual Studio platform: Options for VCC can be set directly in Visual Studio's configuration windows. Verification can be started for opened source files with a single click or keyboard shortcut. Any errors that are found are displayed directly in the source code by underlining the respective elements and displaying tooltips when hovering the cursor above them.

The support for a proper IDE was perceived to be VCC's biggest advantage in the practical comparison. VCC was not as easily accessible as eCv because it required more complicated, sometimes not immediately obvious annotations. Still, it is a powerful tool that has the advantage of being open source and supporting the verification of multi-threaded programs. Judging from the results of this comparison, VCC should be considered both for use in industry and teaching, especially when users are already proficient in Visual Studio. Unfortunately the current license agreement permits only non-commercial use, ruling VCC out for many applications.

VeriFast

VeriFast is unique in that it does not only support a single programming language, but can be used for programs written in C as well as in Java. This

comparison was restricted to looking at the verification of C programs with VeriFast.

The first thing that attracts attention when using VeriFast is the speed of verification. In all situations encountered in the comparison, VeriFast returned results without any noticeable delay. This is clearly much faster than the other tools: In most cases they are able to assert the correctness of an algorithm in a matter of seconds to a minute which is rather quick as well. More problematic is the aspect that for programs or annotations containing errors it often takes much longer to yield a result. This interferes with effective debugging since a lot of time may pass when several iterations of verifying and modifying annotations are required until a positive result is obtained. The problem could be noticed especially when using eCv and KeY. VeriFast does not suffer from this: It returns results immediately even for erroneous programs and thus does not impede productivity.

The graphical user interface, although only offering basic features, was helpful when working on specifications. For example, preconfigured header files (where default lemma and fixpoint functions are defined) are opened automatically when verifying a program to simplify looking up definitions, if needed. VeriFast supports the use of two different provers. It would be useful if the selection of which prover to use could be integrated into the user interface. Currently, VeriFast has to be restarted with certain parameters to select a prover.

Annotations for VeriFast are quite different from those used by the other tools. For example, the annotation language misses constructs for specifying quantification. Instead, VeriFast relies on the user to implement lemmas and fixpoints to express those facts that cannot be stated directly in the annotation language. Thus, VeriFast's approach is quite hard to grasp for a user just introduced to the tool. This is aggravated by the fact that there exists no proper documentation regarding predefined lemmas. Only some of them are mentioned in the tutorial where they are applied to examples, and no overview in the style of an API documentation or similar is available.

At least for the algorithms presented here, annotations for VeriFast are considerably more lengthy and complex than annotations for the other tools. In some cases, like the tutorial solution to the `strlen` function that was discussed above, annotations for VeriFast have only very little resemblance to the other implementations made in this comparison.

In summary, VeriFast is an interesting tool that supports the verification of programs written in C or Java, including the possibility of using multiple threads. It offers a unique annotation language that was designed to heavily rely on the formulation of custom lemmas. While this feature is quite powerful, it causes a novice user to be overwhelmed by the different possibilities

of handling a problem, especially considering that there exists no complete documentation. As VeriFast supports two programming languages, it is the obvious choice in cases where development or teaching is done in both supported languages. For scenarios where this is no criterion, novice users can attain positive results more easily using the other tools. When more documentation regarding the annotation language and the included lemma functions is available, a new look at VeriFast could be worthwhile.

CHAPTER 5

Thoughts on requirements for better adoption of formal verification

This chapter aims to give a short overview on some more general aspects regarding the use of formal verification in industry or teaching. It is not meant as an extensive analysis, but rather as a starting point for own thoughts. Requirements differ between fields, so aspects regarding use in industry and teaching will be looked at separately.

Still, there exist some common obstacles and noteworthy points. Using formal verification in practice requires learning a new language that is used for annotations or, in the cases of Prototype Verification System and Perfect Developer, also for the algorithm itself. For most annotation languages there is only little educational material like practice-oriented documentation, literature or tutorials available. Thus, in general it is harder and more time-consuming to learn working with an annotation language than to learn a standard programming language. Fortunately, for most annotation languages the number of constructs available is not very high, and therefore the basics of each language can still be comprehended rather quickly.

Most tools use an annotation language that was developed in conjunction with the tool itself. Since no other sources of information exist in this case, it is important that the tools' developers are aware of the necessity of providing good documentation. It is crucial that documentation is extensive enough to cover both introduction and regular use, and also of high enough quality to support users efficiently. In some cases, documentation was inferior to

the quality and features of the tool itself or lagged behind the current state of development.

The practical comparison showed that it is crucial for a verification tool to provide precise and easily comprehensible information to the user. This is especially important to users that are new to the area. Thus, the tools' developers need to give sufficient attention to improving the capabilities for providing result messages, instead of solely focusing on proofs themselves: A tool that provides good output in most situations but has a less capable prover that sometimes requires additional hints in annotations will serve most users better than another tool that has a better prover but can only provide more complex output.

5.1 Industrial applications

The adoption of formal verification in industrial environments is marginal in most areas of software development. In most projects, resources are severely constrained: The number of developers as well as the financial and time budget available is not supportive of the introduction of experimental techniques. Therefore, for formal verification to gain acceptance in the software industry, a few key points need to be fulfilled by verification tools as well as possible.

Most software developers only have little or even no knowledge about formal methods and do not have the possibility to extensively study the topic. Thus, tools only qualify if they do not absolutely require such knowledge. It is desirable that tools support well-known programming languages like C, C++ and Java. In each case, restrictions to supported language features are not welcome – although they currently need to be tolerated, as there is no tool available that supports all features of a language. In some fields, such restrictions might be less of a factor, for example the absence of multi-threading support in embedded software development.

The time span needed to get acquainted with a tool is, at least in large projects, only of secondary importance. In a large project, time taken for training is going to be rather short in relation to the overall time spent working with the tool, anyway. More important is the usability, which is reflected in the amount of time a developer needs to correctly annotate program methods after the initial training and in the degree of a tool's integration into an existing development workflow. These factors have a large influence on a project's duration and cost. Therefore, it is important that tools offer good reference documentation and ideally also provide

a collection of examples that prototypically show how certain tasks and constructs can be handled.

One of the main aspects regarding workflow integration is the support for a preferably large number of integrated development environments or code editors. Optimal support would include syntax highlighting and code completion features not only for the programming language but also for annotations. Verification would be started directly from the IDE, and results would be displayed directly at the respective lines of code. None of the existing tools provide an optimal integration (some none at all), although VCC shows a promising approach for Visual Studio. This is an area where additional work is needed in order to make the introduction of formal verification more efficient.

Fortunately, none of the tools requires developers to change the compilers or build systems they are using: Annotations are either added as comments or they are defined as macros that disappear in a preprocessing step. Thus, regular compilers never see any annotations. Depending on a project's target platform, it might be necessary to have the ability to configure the verification tool regarding details such as the sizes of data types. This is currently only extensively supported by Escher C Verifier.

In many areas it may be advantageous if the possibility of negotiating a dedicated support contract exists. Such commercial agreements typically include guaranteed response times and can therefore positively influence productivity in case problems are encountered. As most tools currently are developed by research institutions, support agreements are seldom available. Of all tools analyzed in section 3, only Perfect Developer and Escher C Verifier offer one. Speaking from the author's own experience, the authors of noncommercial tools provide very good support in most cases as well: They have an interest in promoting and enhancing their developments and are thus happy to help quickly and efficiently.

The programs analyzed in the practical comparison show that pre- and postconditions can be quite complex in some cases. This should not discourage potential users: In industrial software engineering function contracts are very often specified, anyway. Formalizing those natural-language specifications using an annotation language can be performed with rather small effort. Another important aspect when newly introducing formal verification is the possibility of annotating only parts of a program. This way, verification can be started with the most important (or most easily verifiable) functions, and then be extended step by step to the rest of the program. Fortunately, this can be done easily with all tools. An example is the insertion of `assume` statements in annotations for Escher C Verifier. They tell the verifier to take the facts in this statement as a given and can thus be used to establish

facts required by another function's precondition.

A problematic aspect that still remains is the way programs need to be formulated in order to verify correctly: In general-purpose programming languages, there is a large (or even infinite) number of ways to solve most tasks. However, verification tools generally cannot handle all possible formulations automatically. Thus, when annotating existing programs the user needs to be prepared to make (sometimes substantial) changes to the code in order to being able to verify them.

5.2 Teaching

Formal verification has a largely different target in teaching than in the software industry. The main points of interest are the teaching of theoretical aspects of formal verification and illustrating them with the help of tools and examples. The larger aim is for students to become acquainted with formal methods, in order for them to be open-minded regarding use on the job later.

The resources available for university courses are constrained in most cases: There is only little time, funding and few staff attainable. A tool suitable for teaching therefore has to be offered at little or no cost, it must be possible to convey the basics in a rather short time span, and students should be able to work with the tool without the need for constant support.

The time period needed to get acquainted with a tool is more important here than in industrial use: Courses are normally allotted a few hours of work per week. A tool should therefore not be too complex – otherwise there could be too little time for working on meaningful examples after studying theoretical aspects and the general handling of the tool. In contrast, the time needed for realizing examples using a tool is less relevant in teaching: The principles that need to be conveyed can be illustrated with very small examples in most cases. Therefore, the size of examples given to students can be chosen according to the amount of work a tool causes.

Depending on a course's target group and structure the use of a tool that requires background knowledge of formal methods is less problematic than in industrial environments. It might even be preferable to employ a tool that offers more low-level features, for example for displaying the proof steps taken for each line of program code or for optionally performing assisted manual proofs. The level of detail of the information a tool provides should be adequate for a student to see the connection between the theoretical aspects that have been studied and their practical application.

In order to enable students to work with a tool by themselves without requiring a lot of assistance it is required that at least some kind of extensive documentation is available, for example a book or a good tutorial document that also describes advanced aspects. The availability of direct support by the developers is less a factor in teaching. If at all, it could be advantageous in the planning phase of a course.

Similarly, other topics important in industrial environments are less relevant in teaching: Restrictions to the language features supported have no influence on a tool's suitability, at least as long as the fundamental properties of a language are preserved. Good integration into development environments and code editors may help students to be open-minded when beginning to study formal verification, but is not indispensable.

CHAPTER 6

Conclusion

This thesis dealt with the current state of formal verification of software. After an introduction to the topic we looked at the foundations of formal verification in chapter 2. Methods like natural deduction, Hoare logic, weakest preconditions and model checking were explained in a concise way to give the reader basic knowledge about theoretical aspects. Chapter 3 gave a description of several tools for applying formal verification in practice. The focus was on those tools that are capable of automatic verification, without needing the user to manually construct proofs. Each tool's properties were extensively analyzed by researching theoretical background, target groups, features, usability, documentation, license and state of development. The most advanced tools Escher C Verifier, KeY, VCC and VeriFast were selected to compete in a comparison in chapter 4. Care was taken to devise relevant examples that clearly demonstrate whether or not the tools are suitable for users in industrial software development and in educational institutions. They included arithmetic calculations and basic program constructs, but also more complex examples like functions from standard string libraries for C and Java. A detailed description of the performance of each tool was given in section 4.5. Some general thoughts on the requirements of users in teaching and software industry were stated in chapter 5.

This work contains the first comprehensive analysis and comparison of the new generation of software verification tools. When looking at the previous comparison in [37], several points stand out. The capabilities of the available tools have been greatly improved: Today, there is a number of tools available that explicitly target users with little or no background knowledge of formal methods and that support the automatic verification of substantial subsets of general-purpose programming languages. In comparison to a few

years ago the specification languages have become much more powerful, too. They allow the precise specification of complex issues with relatively clear and easily readable syntax. Therefore, the expected behavior can be specified completely even for sophisticated functions.

In general, the promise of being both more powerful and more easily accessible was kept by the tools. The comparison showed that formal verification of software can be considered suitable for practical use. It is still necessary to budget additional costs and time in a development process where formal verification is involved. However, proving program correctness will lead to a better product that not only causes fewer costs in maintenance but may, in certain application areas, also prevent accidents and save lives.

Of course, some areas for improvement remain: The developers of most tools need to work on the output returned to the user in order to more efficiently help in debugging. Making the handling of loops easier by automatically suggesting invariants that describe a loop's semantics is another research topic.

A fictitious ideal system would encompass properties of various existing tools:

- Support for easily using multiple provers like Frama-C/Jessie
- Field-tested in industrial environments like VCC
- Support for multi-threaded programs like VCC and VeriFast
- IDE integration like VCC
- Result messages like Escher C Verifier
- Avoiding hand-coded loops like Perfect Developer, or suggesting invariants like (in a basic form) Escher C Verifier
- Object-orientation like Perfect Developer and KeY
- Detailed target platform settings like Escher C Verifier
- Instant results like VeriFast
- Good documentation and accessibility like Escher C Verifier
- Transparency with regard to known bugs or problems like Escher C Verifier or KeY
- Open-source software like Frama-C/Jessie, KeY or VCC

The list shows that a lot of work is still necessary in order to yield a tool that performs optimally on different levels.

A future comparison could look at several aspects. Trials could be extended with more elaborate tasks, for example by verifying programs that encompass complex C structured types or Java class hierarchies. Since multi-threading becomes more important each year, an in-depth look at the capabilities the tools have for verifying multi-threaded programs could be worthwhile. The use of formal verification in larger projects might be impeded by long verification times. As explained in section 4.5, verifiers often take a long time to return results in the case of incorrect programs or annotations, which is problematic when debugging programs. Future work could therefore also deal with this aspect.

All tools analyzed in this thesis support verifying only parts of programs. Thus, a pragmatic approach can be taken: Every bit of formal verification introduced into a development process is better than no verification at all. The most important conclusion from this work is that today formal verification is already suitable for industrial use in many fields. Curricula need to be adapted in order for more students to become acquainted with the positive aspects of the topic.

APPENDIX A

VeriFast: Factorial

This is the code for verifying the factorial function in VeriFast that was provided by developer Bart Jacobs and was omitted from section 4.4 on page 91 due to its length.

```
//@ #include "nat.h"

/*@

lemma void nonneg_mult(int a, int b)
    requires 0 <= a &*& 0 <= b;
    ensures 0 <= a * b;
{
    for (int i = 0; i < b; i++)
        invariant 0 <= i &*& i <= b &*& 0 <= a * i;
        decreases b - i;
    {
    }
}

lemma void mult_le(int a, int b)
    requires 0 <= a &*& 1 <= b;
    ensures a <= a * b;
{
    for (int i = 1; i < b; i++)
        invariant i <= b &*& a <= a * i;
        decreases b - i;
    {
```

```
    }
}

lemma void mult_commut(int a, int b)
    requires true;
    ensures a * b == b * a;
{
}

fixpoint int factorial(nat n) {
    switch (n) {
        case zero: return 1;
        case succ(m): return int_of_nat(n) *
                factorial(m);
    }
}

lemma void mult_cong(int a1, int a2, int b)
    requires a1 == a2;
    ensures a1 * b == a2 * b;
{
}

lemma void pos_factorial(int m)
    requires 0 <= m;
    ensures 1 <= factorial(nat_of_int(m));
{
    for (int i = 0; i < m; i++)
        invariant 0 <= i &*& 1 <= factorial
                (nat_of_int(i)) &*& i <= m;
        decreases m - i;
    {
        nat_of_int_of_nat(succ(nat_of_int(i)));
        mult_le(factorial(nat_of_int(i)), i + 1);
        int_of_nat_of_int(i + 1);
        mult_cong(int_of_nat(nat_of_int(i + 1)),
                i + 1, factorial(nat_of_int(i)));
    }
}

lemma void le_factorial(int m, int n)
```

```
    requires 0 <= m &*& m <= n;
    ensures factorial(nat_of_int(m)) <=
        factorial(nat_of_int(n));
{
    pos_factorial(m);
    for (int i = m; i < n; i++)
        invariant 0 <= i &*& 1 <= factorial(
                nat_of_int(m)) &*& factorial(
                nat_of_int(m)) <= factorial(
                nat_of_int(i)) &*& i <= n;
        decreases n - i;
    {
        nat_of_int_of_nat(succ(nat_of_int(i)));
        mult_le(factorial(nat_of_int(i)), i + 1);
        int_of_nat_of_int(i + 1);
        mult_cong(int_of_nat(nat_of_int(i + 1)),
                i + 1, factorial(nat_of_int(i)));
    }

}

@*/

int fact(int n)
    /*@ requires 0 <= n &*&
                factorial(nat_of_int(n)) <= INT_MAX;
    @*/
    //@ ensures result == factorial(nat_of_int(n));
{
    int result = 1;
    //@ nat_of_int_of_nat(succ(zero));
    int i = 2;
    //@ if (n <= 0) {} else if (n <= 1) {} else {}
    if (n < 2)
        return result;
    for (;;)
        /*@ invariant 2 <= i &*& i <= n &*&
                        1 <= result &*& result ==
                        factorial(nat_of_int(i - 1));
        @*/
    {
```

```
        //@ nonneg_mult(result, i);
        //@ le_factorial(i, n);
        /*@ nat_of_int_of_nat(succ(
                nat_of_int(i - 1)));
        @*/
        //@ mult_le(result, i);
        result *= i;
        if (i == n)
            break;
        i++;
    }
    return result;
}
```

Bibliography

[1] Wolfgang Ahrendt et al. "The KeY tool". In: *Software and Systems Modeling* 4 (1 2005), pp. 32–54. ISSN: 1619-1366. DOI: 10.1007/s10270-004-0058-x.

[2] Mike Barnett et al. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Formal Methods for Components and Objects*. Vol. 4111. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 364–387. DOI: 10.1007/11804192_17.

[3] Patrick Baudin et al. *ACSL: ANSI/ISO C Specification Language. Version 1.5 – Carbon-20110201*. 2011. URL: http://frama-c.com/download.html.

[4] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, eds. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007. ISBN: 3-540-68977-X.

[5] Bernhard Beckert and Claude Marché. *Formal Verification of Object-Oriented Software. Papers presented at the International Conference, June 28-30, 2010, Paris, France*. Technical report. Fakultät für Informatik, Universität Karlsruhe, 2010. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000019083.

[6] Sascha Böhme et al. "HOL-Boogie – An Interactive Prover-Backend for the Verifying C Compiler". In: *Journal of Automated Reasoning* 44 (1–2 Feb. 2010), pp. 111–144. ISSN: 0168-7433. DOI: 10.1007/s10817-009-9142-9.

[7] Armin Biere et al. "Symbolic Model Checking without BDDs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 1579. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1999, pp. 193–207. DOI: 10.1007/3-540-49059-0_14.

[8] CEA – LIST. *Frama-C publications*. 2011. URL: http://bts.frama-c.com/dokuwiki/doku.php?id=mantis:frama-c:publications#jessie.

[9] CEA – LIST. *Frama-C/Jessie website*. 2011. URL: http://frama-c.com/jessie.html.

[10] Patrice Chalin. "Improving JML: For a Safer and More Effective Language". In: *FME 2003: Formal Methods*. Vol. 2805. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003, pp. 440–461. ISBN: 978-3-540-40828-4. DOI: 10.1007/978-3-540-45236-2_25.

[11] Edmund Clarke and Daniel Kröning. "Hardware Verification using ANSI-C Programs as a Reference". In: *Proceedings of ASP-DAC 2003*. IEEE Computer Society Press, Jan. 2003, pp. 308–311. ISBN: 0-7803-7659-5.

[12] Edmund Clarke, Daniel Kröning, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 168–176. ISBN: 3-540-21299-X.

[13] Edmund M. Clarke and E. Allen Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic". In: *Logics of Programs*. Vol. 131. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1982, pp. 52–71. DOI: 10.1007/BFb0025774.

[14] Ernie Cohen et al. "Local Verification of Global Invariants in Concurrent Programs". In: *Computer Aided Verification*. Vol. 6174. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010. DOI: 10.1007/978-3-642-14295-6_42.

[15] Ernie Cohen et al. "VCC: A Practical System for Verifying Concurrent C". In: *Theorem Proving in Higher Order Logics*. Vol. 5674. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, pp. 23–42. DOI: 10.1007/978-3-642-03359-9_2.

[16] Ernie Cohen et al. *Verifying Concurrent C Programs with VCC. Working draft, version 0.2*. Apr. 18, 2011. URL: http://vcc.codeplex.com/.

[17] Loïc Correnson, Zaynah Dargaye, and Anne Pacalet. *WP Plug-in (Draft) Manual. Version 0.3 for Carbon-20110201*. 2011. URL: http://frama-c.com/download.html.

[18] Loïc Correnson et al. *Frama-C User Manual. Release Carbon-20110201*. 2011. URL: http://frama-c.com/download.html.

[19] David Crocker. *David Crocker's Verification Blog*. 2011. URL: http://critical.eschertech.com/.

[20] David Crocker. "Perfect Developer: A tool for Object-Oriented Formal Specification and Refinement". In: *Tools Exhibition Notes at Formal Methods Europe*. 2003. URL: http://www.eschertech.com/papers/index.php.

[21] David Crocker. *Personal communication*. Oct. 26, 2011.

[22] David Crocker. "Safe Object-Oriented Software: the Verified Design-by-Contract paradigm". In: *Practical Elements of Safety: Proceedings of the Twelfth Safety-critical Systems Symposium*. Springer, 2004, pp. 19–41. ISBN: 9781852338008.

[23] David Crocker and Judith Carlton. *A High Productivity Tool for Formally Verified Software Development*. Technical report. Presented at the Formal Methods Workshop in Industrial Applications. Ghent, Belgium. Escher Technologies, 2003. URL: http://www.eschertech.com/papers/index.php.

[24] David Crocker and Judith Carlton. "Verification of C Programs Using Automated Reasoning". In: *Fifth IEEE International Conference on Software Engineering and Formal Methods*. 2007, pp. 7–14. ISBN: 978-0-7695-2884-7.

[25] Edsger W. Dijkstra. *A Discipline of Programming*. Upper Saddle River, NJ, USA: Prentice Hall, 1976. ISBN: 013215871X.

[26] Edsger W. Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs". In: *Communications of the ACM* 18 (8 Aug. 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975.

[27] E. Allen Emerson. "The Beginning of Model Checking: A Personal Perspective". In: *25 Years of Model Checking*. Vol. 5000. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 27–45. DOI: 10.1007/978-3-540-69850-0_2.

[28] Christian Engel et al. *KeY Quicktour for JML*. Oct. 6, 2010. URL: http://www.key-project.org/case_studies/.

[29] Escher Technologies. *Escher C Verifier datasheet*. 2011. URL: http://www.eschertech.com/papers/index.php.

[30] Escher Technologies. *Escher C Verifier Reference Manual*. 2011. URL: http://www.eschertech.com/product_documentation/ecvReference/eCv_Manual.html.

[31] Escher Technologies. *Escher C Verifier website*. 2011. URL: http://www.eschertech.com/products/ecv.php.

[32] Escher Technologies. *Perfect Developer for Critical Systems datasheet*. 2011. URL: http://www.eschertech.com/papers/index.php.

[33] Escher Technologies. *Perfect Developer website*. 2011. URL: http://www.eschertech.com/products/perfect_developer.php.

[34] Escher Technologies. *Product Announcement*. 2011. URL: http://www.eschertech.com/products/announcements/2011may04.pdf.

[35] Escher Technologies. *Safety-critical software*. 2011. URL: http://www.eschertech.com/products/safety.php.

[36] Escher Technologies. *Universities using PD in teaching*. 2011. URL: http://www.eschertech.com/educational/universities_using.php.

[37] Ingo Feinerer. "Formal Program Verification: a Comparison of Selected Tools and Their Theoretical Foundations". MA thesis. Technische Universität Wien, 2005.

[38] Ingo Feinerer and Gernot Salzer. "A Comparison of Tools for Teaching Formal Software Verification". In: *Formal Aspects of Computing* 21.3 (May 2009), pp. 293–301. DOI: 10.1007/s00165-008-0084-5.

[39] Robert W Floyd. "Assigning meanings to programs". In: *Proceedings of the American Mathematical Society Symposia on Applied Mathematics* 19 (1967), pp. 19–32.

[40] Formal Verification Group. *CBMC website*. 2011. URL: http://www.cs.cmu.edu/~modelcheck/cbmc/.

[41] Formal Verification Group. *CPROVER Manual*. 2011. URL: http://www.cprover.org/cprover-manual/.

[42] Formal Verification Group. *CPROVER website*. 2011. URL: http://www.cprover.org/.

[43] Gerhard Gentzen. "Untersuchungen über das logische Schließen I". In: *Mathematische Zeitschrift* 39 (2 1934), pp. 176–210.

[44] David Gries. *The Science of Programming*. Springer, 1981. ISBN: 038790641X.

[45] David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic*. Cambridge, MA, USA: MIT Press, 2000. ISBN: 0262082896.

[46] C. A. R. Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12 (10 Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.

[47] Michael Huth and Mark Ryan. *Logic in Computer Science: modelling and reasoning about systems (second edition)*. Cambridge University Press, 2004. ISBN: 052154310X. URL: http://pubs.doc.ic.ac.uk/logic-computer-science-second/.

[48] ISO/IEC 9899:1999 Cor. 3:2007(E). *Programming languages – C, WG14 N1256*. Geneva, Switzerland: International Organization for Standardization, 2007. URL: http://www.open-std.org/jtc1/sc22/wg14/www/standards.

[49] Bart Jacobs. *VeriFast website*. Katholieke Universiteit Leuven. 2011. URL: http://people.cs.kuleuven.be/~bart.jacobs/verifast/.

[50] Bart Jacobs and Frank Piessens. *The VeriFast Program Verifier*. Technical Report CW-520. Department of Computer Science, Katholieke Universiteit Leuven, Aug. 2008. URL: http://www.cs.kuleuven.be/publicaties/reports/cw/CW520.abs.html.

[51] Bart Jacobs and Frank Piessens. *VeriFast Reference Manual*. Jan. 14, 2011.

[52] Bart Jacobs, Jan Smans, and Frank Piessens. "A Quick Tour of the VeriFast Program Verifier". In: *Programming Languages and Systems*. Vol. 6461. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 304–311. DOI: 10.1007/978-3-642-17164-2_21.

[53] Bart Jacobs, Jan Smans, and Frank Piessens. *The VeriFast Program Verifier: A Tutorial*. Nov. 17, 2010. URL: http://people.cs.kuleuven.be/~bart.jacobs/verifast/.

[54] Bart Jacobs et al. "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java". In: *NASA Formal Methods*. Vol. 6617. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 41–55. DOI: 10.1007/978-3-642-20398-5_4.

[55] Stanisław Jaśkowski. "On the rules of suppositions in formal logic". In: *Studia Logica* 1 (1934), pp. 5–32.

[56] Karlsruhe Institute of Technology. *KeY changelog*. Oct. 1, 2011. URL: http://www.key-project.org/download/CHANGELOG.

[57] Karlsruhe Institute of Technology. *KeY known issues*. Oct. 1, 2011. URL: http://www.key-project.org/download/KnownIssues.txt.

[58] Karlsruhe Institute of Technology. *KeY website*. 2011. URL: http://www.key-project.org/.

[59] James C. King. "Symbolic execution and program testing". In: *Communications of the ACM* 19 (7 July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252.

[60] Daniel Kroening, Edmund Clarke, and Karen Yorav. "Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking". In: *Proceedings of DAC 2003*. ACM Press, 2003, pp. 368–371. ISBN: 1-58113-688-9.

[61] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: the Automata-Theoretic Approach*. Princeton University Press, 1994. ISBN: 9780691034362. URL: http://books.google.com/books?id=iB5066RPRxsC.

[62] Laboratoire de Recherche en Informatique. *Why website*. 2011. URL: http://why.lri.fr/.

[63] Daniel Larsson and Wojciech Mostowski. "Specifying Java Card API in OCL". In: *Electronic Notes in Theoretical Computer Science* 102 (Nov. 2004), pp. 3–19. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2003.09.001.

[64] Claude Marché and Yannick Moy. *Jessie Plugin Tutorial. Frama-C version: Carbon, Jessie plugin version: 2.28*. 2010. URL: http://frama-c.com/download.html.

[65] Bertrand Meyer. *Object-Oriented Software Construction*. 2nd ed. Prentice Hall, 1997. Chap. 11. ISBN: 0-13-629155-4.

[66] Microsoft Research. *Boogie website*. 2011. URL: http://research.microsoft.com/en-us/projects/boogie/.

[67] Microsoft Research. *VCC website*. 2011. URL: http://vcc.codeplex.com/.

[68] MISRA. *Guidelines for the Use of the C Language in Critical Systems*. MISRA, 2004. ISBN: 0952415623.

[69] MISRA. *MISRA website*. 2009. URL: http://www.misra.org.uk/MISRAHome/WhatisMISRA/tabid/66/Default.aspx.

[70] S. Owre, J. M. Rushby, and N. Shankar. "PVS: A Prototype Verification System". In: *11th International Conference on Automated Deduction (CADE)*. Vol. 607. Lecture Notes in Artificial Intelligence. Saratoga, NY, USA: Springer-Verlag, June 1992, pp. 748–752. URL: http://www.csl.sri.com/papers/cade92-pvs/.

[71] S. Owre et al. *PVS System Guide*. 2001. URL: http://pvs.csl.sri.com/documentation.shtml.

[72] Sam Owre. *Prototype Verification System website*. SRI International. 2011. URL: `http://pvs.csl.sri.com/`.

[73] Sam Owre and Natarajan Shankar. "Writing PVS Proof Strategies". In: *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*. NASA Conference Publication CP-2003-212448. Hampton, VA, USA: NASA Langley Research Center, Sept. 2003, pp. 1–15.

[74] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. LICS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 55–74. ISBN: 0-7695-1483-9.

[75] Peter H. Schmitt and Benjamin Weiß. "Inferring Invariants by Symbolic Execution". In: *Proceedings, 4th International Verification Workshop (VERIFY'07)*. Vol. 259. CEUR Workshop Proceedings. CEUR-WS.org, 2007, pp. 195–210.

[76] Sun Microsystems. *Virtual Machine Specification. Java Card Platform, Version 2.2.2*. 2005. URL: `http://java.sun.com/javacard/specs.html`.

[77] Technische Universität Berlin, Institut für Softwaretechnik und Theoretische Informatik. *Verifikation von C Programmen in der Praxis*. 2011. URL: `http://www.swt.tu-berlin.de/menue/studium_und_lehre/lehrveranstaltungen/vcc/`.

[78] Verisoft XT consortium. *Verisoft XT website*. 2011. URL: `http://www.verisoftxt.de/`.

[79] Frédéric Vogels et al. "Annotation Inference for Separation Logic Based Verifiers". In: *Formal Techniques for Distributed Systems*. Vol. 6722. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 319–333. DOI: `10.1007/978-3-642-21461-5_21`.

[80] Jürgen Winkler. *Frege Program Prover website*. Friedrich-Schiller-Universität Jena. 2009. URL: `http://psc.informatik.uni-jena.de/fpp/fpp-main.htm`.

[81] Jürgen Winkler. "The Frege Program Prover". In: *42. Internationales Wissenschaftliches Kolloquium, Technische Universität Ilmenau, Band 1*. 1997, pp. 116–121.