MAGISTERARBEIT

Formal Program Verification: a Comparison of Selected Tools and Their Theoretical Foundations

Ausgeführt am

Institut für Computersprachen
der Technischen Universität Wien

unter der Anleitung von

A.o.Univ.Prof. DI. Dr. Gernot Salzer

durch

Ingo Feinerer, Bakk.techn.

Felixdorfer Gasse 11
A-2700 Wiener Neustadt

Wien, Jänner 2005

MASTER THESIS

Formal Program Verification: a Comparison of Selected Tools and Their Theoretical Foundations

Ingo Feinerer

Theory and Logic Group
Institute of Computer Languages
Vienna University of Technology

Advisor

Gernot Salzer

Vienna, January 2005

Zusammenfassung

Formale Spezifikation und Verifikation sind durch die durch kontinuierliche Weiterentwicklung in letzter Zeit an einem Punkt angelangt, wo Programme beinahe automatisch verifiziert werden können.

Das Ziel dieser Magisterarbeit ist es, sowohl kommerzielle als auch für wissenschaftliche Zwecke entwickelte Verifikationsprogramme zu testen. Der Hauptaugenmerk liegt auf dem Nutzen dieser Werkzeuge in der Software-Entwicklung und in der Lehre. Hierzu wird diese Magisterarbeit die theoretischen Grundlagen vorstellen und auf die verschiedenen Fähigkeiten und Eigenheiten der ausgewählten Werkzeuge eingehen.

Die theoretischen Grundlagen behandeln einerseits Ansätze, die für die formale Verifikation gebraucht werden, andererseits wird die Funktionsweise der ausgewählten Werkzeuge erklärt.

Die begutachteten Programme sind der Frege Program Prover, KeY, Perfect Developer und das Prototype Verification System. Die Beispiele, mit denen diese Werkzeuge getestet werden, sind typische Problemstellung der Informatik. Bei der Evaluation wird auf den ganzen Ablauf beim Einsatz dieser Werkzeuge eingegangen und nicht nur auf das Endergebnis.

Abstract

Formal specification and verification of software have made small but continuous advances throughout its long history, and have reached a point where commercial tools became available for verifying programs semi-automatically or automatically.

The aim of the master thesis is to evaluate commercial and academic verification tools with respect to their usability in developing software and in teaching formal methods. The thesis will explain the theoretical foundation and compare the capabilities and characteristics of selected commercial and academic tools on concrete examples.

The theoretical foundations deal on the one hand with the general ideas and principles of formal software verification, on the other hand present some internals of the selected tools to give a comprehensive understanding.

The discussed tools are the Frege Program Prover, KeY, Perfect Developer, and the Prototype Verification System. The examples encompass simple standard computer science problems. The evaluation of these tools concentrates on the whole development process of specification and verification, not just on the verification results.

Acknowledgements

I would like to thank my family, especially my mother Inge, for supporting me.

Gernot Salzer, my advisor, helped me whenever he could and invested a lot of time in discussing and investigating problems together with me. David Crocker gave excellent support on Perfect Developer, Andreas Roth and Steffen Schlager offered helpful instructions on KeY, Jürgen Winkler provided papers and references on FPP. Also the subscribers of the PVS mailing list came up with nice ideas.

Contents

1 Introduction

Formal software verification has become a more and more important issue in developing security related software during the last decades. As a reaction, ISO — the International Organisation for Standardisation — issued the ISO 15408 Standard, defining exactly various quality levels for tested and verified software. This standard is represented in the Common Criteria Project, with members of security organisations around the globe.

During the last years, formal specification and verification tools have been introduced, especially designed for standard development processes. The focus ranges from security related projects, over hardware circuit verification to software driver verification. In particular model checking has been very successful.

Based on this evolution this thesis deals with four specification and verification tools that enable the user to build software complying with the most demanding restrictions of the ISO 15408 Standard. The aim is to construct software that meets the Evaluation Assurance Levels 6 and 7 (EAL 6, EAL 7) defined in the Common Criteria Project. In other words this means fully verified specification and code.

For a long time users of these tools have been assumed to be experts in formal methods. With new target groups requirements changed. Therefore this thesis evaluates the tools with respect to two groups: software engineers with a good knowledge of computer science but without specific training in formal methods, and students of computer science and software engineering in the middle of their studies, being confronted with formal verification tools for the first time.

The four tools that will be investigated are the Frege Program Prover, KeY, Perfect Developer, and the Prototype Verification System. In the first part of this work, the theoretical background — main calculi and ideas of formal verification — is presented. Then the internals of the tools are discussed, showing the different approaches and techniques from the theoretical side. Finally, by going through a set of simple standard computer science examples, the different characteristics and capabilities are presented in a practical form. By examining the tools from both sides, theory and practice, their usability in developing software and in teaching formal methods for the above defined target group is discussed.

Historical perspective

Formal verification has always been a well discussed problem by a lot of excellent computer scientists. Jones [2003] mentions three phases of historical development:

Pre-Hoare Herman Goldstine, John von Neumann, Alan Turing, Robert Floyd and John McCarthy are only some famous computer pioneers that can be mentioned here. They started thinking about errors in their programs from the start on and had already ideas to avoid them. The idea of assertions for programs was borne.

Hoare's axiomatic approach Tony Hoare presented his axioms in his famous paper Hoare [1969] — the calculus is also discussed in section 2.4 of this thesis. This formulation led to new approaches towards formal verification in the late 1970s.

Post-Hoare After Hoare's axiomatic approach formal verification became a broad scientific research problem with connections to the semantics of programming languages. Many scientists, like Edsger Dijkstra, Tony Hoare himself and many more, continued to work on these foundations and made continuous improvements.

Until today automatic verification is an intensively considered problem. E.g. Tony Hoare stated the problem of building a "verifying compiler" as one of the big challenges of computer science in his paper Hoare [2003]. Also the idea of reusing verified software is appreciated by the scientific community — Meyer [2003] deals in detail with that idea.

2 Theoretical Foundations

This section deals with general ideas and principles underlying formal software verification and explains basic notions and calculus frameworks.

The reader is assumed to have a minimal background on formal logic, especially in classical propositional and first order logic. Detailed explanations of these basics are given in Gallier [2003] or Huth and Ryan [2004].

The following sections discuss an introduction to propositional logic, natural deduction, the sequent calculus, the Hoare calculus and weakest preconditions.

We present only rules for propositional logic to give the flavour of the various approaches. Real systems for proving statements about programs are more complex in at least two respects: first, propositional logic has to be extended to first-order or even higher-order logics, i.e., in general we need quantification over first-order or higher-order variables. Second, provers need built-in knowledge about the data types used in programs and formal specifications, like reals and integers, lists and sets. Moreover, special mechanisms have to be provided to deal with equalities, inequalities and other basic theories, using e.g. decision procedures.

2.1 Propositional Logic

This section deals with the basics of propositional classical logic. The following notations will be used as the basic formalism in later chapters.

Syntax The alphabet for propositional formulas consists of:

A countable set of propositional symbols A_i,

logical connectives $\vee, \wedge, \neg, \supset, \perp$ and

auxiliary symbols (parentheses).

Definition 2.1 *The set of well formed propositional formulas* PROP *is inductively defined as:*

Every propositional symbol A_i and $\perp$ are $\in PROP$,

Whenever $A, B \in PROP$, then $\neg A, A \supset B, A \vee B, A \wedge B \in PROP$.

Semantics

Definition 2.2 *A* valuation v *is a function that maps well formed propositional formula to truth values, hence* $v : PROP \mapsto \{true, false\}$.

Let $A, B \in PROP$. We write $v \models A$ iff A evaluates to *true* under the valuation v. The satisfaction relation is defined as:

$v \models A$ iff $v(A) = true$

$v \models A \wedge B$ iff $v \models A$ and $v \models B$

$v \models A \vee B$ iff $v \models A$ or $v \models B$

$v \models A \supset B$ iff $v \not\models A$ or $v \models B$

$v \models \neg A$ iff $v \not\models A$

2.2 Natural Deduction

Let $\phi_1, \phi_2, \phi_3, \ldots, \phi_n$ be formulas, which are called *premises*, and ψ be another formula called *conclusion*.

Definition 2.3 *The expression* $\phi_1, \phi_2, \phi_3, \ldots, \phi_n \vdash \psi$ *is called* sequent.

But instead of $\phi_1, \phi_2, \phi_3, \ldots, \phi_n \vdash \psi$ we write

$$\frac{\phi_1 \qquad \phi_2 \qquad \phi_3 \qquad \ldots \qquad \phi_n}{\psi}$$

This means that if all premises $\phi_1, \phi_2, \phi_3, \ldots, \phi_n$ are *true*, we conclude that the conclusion ψ is *true*.

From now on let ϕ, ψ and χ denote propositional formulas. The natural deduction rules for propositional logic distinguish between introduction (i-rules) and elimination (e-rules) rules for connectives and are defined as follows:

Disjunction Rules:

$$\vee i_1 \ \frac{\phi}{\phi \vee \psi} \qquad \vee i_2 \ \frac{\psi}{\phi \vee \psi}$$

$$\vee e \ \frac{\phi \vee \psi \qquad \phi \vdash \chi \qquad \psi \vdash \chi}{\chi}$$

Conjunction Rules:

$$\wedge i \; \frac{\phi \qquad \psi}{\phi \wedge \psi}$$

$$\wedge e_1 \; \frac{\phi \wedge \psi}{\phi} \qquad \wedge e_2 \; \frac{\phi \wedge \psi}{\psi}$$

Implication Rules:

$$\supset i \; \frac{\phi \vdash \psi}{\phi \supset \psi}$$

$$\supset e \; \frac{\phi \qquad \phi \supset \psi}{\phi}$$

Negation Rules:

$$\neg i \; \frac{\phi \vdash \bot}{\neg\phi}$$

$$\neg e \; \frac{\phi \qquad \neg\phi}{\bot}$$

Bottom Rule:

$$\bot e \; \frac{\bot}{\phi}$$

Double Negation Rule:

$$\neg\neg e \; \frac{\neg\neg\phi}{\phi}$$

Some Derived Rules:

$$\text{Modus tollens} \; \frac{\phi \supset \psi \qquad \neg\psi}{\neg\phi}$$

$$\neg\neg i \; \frac{\phi}{\neg\neg\phi}$$

$$\text{Reductio ad absurdum}\ \frac{\neg\phi \vdash \bot}{\phi}$$

$$\text{Tertium non datur}\ \frac{}{\phi \vee \neg\phi}$$

Definition 2.4 *A* proof *in natural deduction is the smallest set X such that*

> *the one element tree ϕ belongs to X for all well formed propositional formulas ϕ and*
>
> *if $\phi \in X$, $\psi \in X$ and $\chi \in X$, then every application of the above defined natural deduction rules is $\in X$.*

Definition 2.5 *Logical formulas ϕ such that $\vdash \phi$ holds are called* theorems.

Definition 2.6 *Two formulas of propositional logic ϕ and ψ are called* provably equivalent *iff $\phi \vdash \psi$ and $\psi \vdash \phi$.*

Proposition 2.7 *The natural deduction calculus for propositional formulas is sound.*

Proposition 2.8 *The natural deduction calculus for propositional formulas is complete.*

More information on natural deduction can be looked up in Huth and Ryan [2004] and van Dalen [2004].

2.3 Sequent Calculus

The sequent calculus was originally developed by Gentzen, and published in one of his famous papers [Gentzen, 1935].

Definition 2.9 *A* sequent *is a pair* (Γ, Δ) *of finite multi-sets of propositional formulas.*

It should be mentioned that some authors (like Fitting [1990]) define a sequent as a set of formulas, others as a sequence.

Instead of (Γ, Δ) the notation $\Gamma \to \Delta$ is common. Γ is called *antecedent* and Δ *succedent*. For simplicity, propositional sequents $\{A_1, \ldots, A_n\} \to \{B_1, \ldots, B_m\}$ are denoted as $A_1, \ldots, A_n \to B_1, \ldots, B_m$. Similarly, we write $\to \Delta$ and $\Gamma \to$ if Γ and Δ is empty, respectively.

A valuation v makes a sequent $A_1, \ldots, A_n \to B_1, \ldots, B_m$ true iff

$$v \models (A_1 \wedge \ldots \wedge A_n) \supset (B_1 \vee \ldots \vee B_m)$$

Let Γ, Δ be arbitrary propositional sequents and let A and B be propositions. The rules in the Gentzen System are then as follows:

Disjunction Rules:

$$(\vee\text{-l})\ \frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}$$

$$(\vee\text{-r})\ \frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \vee B}$$

Conjunction Rules:

$$(\wedge\text{-l})\ \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}$$

$$(\wedge\text{-r})\ \frac{\Gamma \vdash \Delta, A \qquad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B}$$

Implication Rules:

$$(\supset\text{-l})\ \frac{\Gamma \vdash \Delta, A \qquad \Gamma, B \vdash \Delta}{\Gamma, A \supset B \vdash \Delta}$$

$$(\supset\text{-r})\ \frac{\Gamma, A \vdash \Delta, B}{\Gamma \vdash \Delta, A \supset B}$$

Negation Rules:

$$(\neg\text{-l})\ \frac{\Gamma \vdash \Delta, A}{\Gamma, \neg A \vdash \Delta}$$

$$(\neg\text{-r})\ \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \Delta, \neg A}$$

Every rule consists of one or two upper sequents called *premises*, and one lower sequent called *conclusion.*

Definition 2.10 *A sequent $A_1, \ldots, A_n \to B_1, \ldots, B_m$ is* falsifiable *iff there exists a valuation v such that*

$$v \models (A_1 \wedge \ldots \wedge A_n) \wedge (\neg B_1 \wedge \ldots \wedge \neg B_m).$$

Definition 2.11 *A sequent $A_1, \ldots, A_n \to B_1, \ldots, B_m$ is* valid *iff for every valuation v*

$$v \models (A_1 \wedge \ldots \wedge A_n) \supset (B_1 \vee \ldots \vee B_m).$$

This is also denoted by

$$\models (A_1, \ldots, A_n) \to (B_1, \ldots, B_m).$$

Definition 2.12 *An* axiom *is any sequent $\Gamma \to \Delta$ such that Γ and Δ contain some common formula.*

Proposition 2.13 *Every axiom is valid.*

A *deduction tree* is a tree whose nodes are labelled with sequents. Every sequent at an inner node must be obtained from the sequents at its children nodes by applying one of the rules of sequent calculus. The label on the root is the sequent that is proved. It is called the *conclusion.* A *proof* is a deduction tree that has only axioms as leaves. A *counterexample* is a sequent consisting only of propositional letters that is no axiom. A *failed deduction tree* is a deduction tree with a counterexample as one of its leaves. A sequent is *provable* iff there is a proof tree of which it is the conclusion. If a sequent $\Gamma \to \Delta$ is provable, it is denoted as

$$\vdash \Gamma \to \Delta.$$

Proposition 2.14 *The Gentzen calculus for formulas in propositional logic is sound. Thus, if a sequent $\Gamma \to \Delta$ is provable, then it is valid.*

Proposition 2.15 *The Gentzen calculus for propositional formulas is complete. Thus, every valid sequent is provable. Furthermore there exists an algorithm for deciding whether a sequent is valid and if so, a proof tree is generated.*

The interested reader may find additional material in Gallier [2003] and Salzer [2002].

2.4 Hoare Calculus

This calculus was introduced in Hoare [1969].

The input-output relation for a program S is specified as follows:

$$\{P\}\ S\ \{Q\}.$$

P and Q are logic formulas. In this context they are often called assertions or conditions. P is the *precondition* and Q is the *postcondition*. The precondition describes the set of intended initial states for the program S, whereas the postcondition describes the set of final states for S, if S terminates.

Definition 2.16 $\{P\}\ S\ \{Q\}$ *is* partially correct *if every terminating computation of S starting from a P-state ends up in a Q-state.*

Definition 2.17 $\{P\}\ S\ \{Q\}$ *is called* totally correct *if every computation of S starting from a P-state terminates and ends up in a Q-state.*

The Hoare calculus proves the correctness of programs with a syntax-driven axiomatic system. The Hoare System consists of the following axioms and rules:

Skip statement:

$$\{P\}\ \texttt{skip}\ \{P\}$$

Assignment statement:

$$\{P\left[{t \atop v}\right]\}\ v \leftarrow t\ \{P\}$$

where $P\left[{t \atop v}\right]$ describes the state P except that v has the value of t.

Composition rule:

$$\frac{\{P\}\ S_1\ \{R\} \qquad \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1;\ S_2\ \{Q\}}$$

Conditional rule:

$$\frac{\{P \wedge B\}\ S_1\ \{Q\} \qquad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{Q\}}$$

Loop rule:

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \texttt{while}\ B\ \texttt{do}\ S\ \{P \wedge \neg B\}}$$

Consequence rule:

$$\frac{P \supset P_1 \qquad \{P_1\}\ S\ \{Q_1\} \qquad Q_1 \supset Q}{\{P\}\ S\ \{Q\}}$$

The rules presented so far are not enough to prove the termination of any program. This system is only able to prove partial correctness. In order to prove total correctness, it is necessary to adapt the loop rule:

Loop rule 2:

$$\frac{\begin{array}{c}\{P \wedge B\}\ S\ \{P\} \\ \{P \wedge B \wedge (t = x)\}\ S\ \{t < x\} \\ P \supset t \geq 0\end{array}}{\{P\}\ \texttt{while}\ B\ \texttt{do}\ S\ \{P \wedge \neg B\}}$$

Notice that t is an integer expression and x is an integer variable that is not part of P, B, t or S.

Proposition 2.18 *The Hoare calculus for the partial correctness of programs is sound.*

Proposition 2.19 *The Hoare calculus for the total correctness of programs is sound.*

Proposition 2.20 *The Hoare calculus for the partial correctness of programs is complete.*

Further information can be found in Apt and Olderog [1994] or Salzer [2002].

2.5 Weakest Preconditions

Let S be a program statement and let Q be a predicate.

Definition 2.21 *The* weakest precondition *$wp(S, Q)$ is the set of initial states, described by a predicate, for which S terminates and Q is true on termination.*

In contrast to the axiomatic Hoare logic the termination is inherent in the definition of pre- and postconditions.

Proposition 2.22 *A program S is correct with respect to the predicates P and Q if $P \supset wp(S, Q)$.*

Weakest preconditions satisfy the following properties:

Axioms:

$$wp(S, \mathit{false}) = \mathit{false}$$

$$\frac{P \supset Q}{wp(S, P) \supset wp(S, Q)}$$

$$\frac{wp(S, P \vee Q)}{wp(S, P) \vee wp(S, Q)}$$

$$\frac{wp(S, P \wedge Q)}{wp(S, P) \wedge wp(S, Q)}$$

The weakest preconditions for typical program statements can be computed as follows:

Skip rule:

$$wp(\texttt{skip}, Q) = Q$$

Assignment rule:

$$wp(v \leftarrow t, Q) = Q\left[{t \atop v}\right]$$

Composition rule:

$$wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))$$

Conditional rule:

$$\begin{aligned} wp(\texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2, Q) &= (B \supset wp(S_1, Q)) \wedge (\neg B \supset wp(S_2, Q)) \\ &= (B \wedge wp(S_1, Q)) \vee (\neg B \wedge wp(S_2, Q)) \end{aligned}$$

Loop rule:

$$\begin{aligned} wp(\texttt{while } B \texttt{ do } S, Q) &= H(Q) \\ &= H_0(Q) \wedge H_1(Q) \wedge H_2(Q) \wedge H_3(Q) \wedge \ldots \end{aligned}$$

where

$$\begin{aligned} H_0(Q) &= \neg B \supset Q \\ H_{k+1}(Q) &= B \supset wp(S, H_k(q)) \end{aligned}$$

For additional material consult Gannon et al. [1993], Gries [1989] or Dijkstra and Scholten [1990]. For the advanced reader, Winkler [1995] discusses the different views and implications of weakest precondition as a predicate transformer versus the idea of weakest precondition as a state set transformer.

3 Selected Tools

This section introduces the reader to the tested tools in a theoretical fashion.

3.1 Overview

We evaluate the verification tools with respect to the following intended users:

Software engineers with a good knowledge of computer science but without specific training in formal methods

Students of computer science and software engineering in the middle of their studies, being confronted with formal verification tools for the first time.

The four tools selected for this thesis are:

Frege Program Prover (FPP) FPP was chosen because of its academic nature. It was explicitly designed for teaching formal methods. It is also a candidate for being used in some lectures at the TU Vienna.

The tested version is available as web service, which was last updated on May 22, 2001.

KeY System (KeY) KeY is the successor of the Karlsruhe integrated Verifier (KiV). KiV has a long tradition and is well known in academia. The KeY system is, at least concerning its intentions, one of the few systems comparable to FPP and PD.

The tested version is KeY-0.1342, the most recent internal version provided by the KeY team.

Perfect Developer (PD) PD was already used for internships and courses at TU Vienna and claims to be one of the few existing commercial tools that can be used by almost any person with just a little knowledge of formal methods.

The tested version is Perfect Developer 2.00.

Prototype Verification System (PVS) PVS is famous and widely cited, and has served as a reference for many years. Therefore it is included in this comparison even though it aims mainly at verification of algorithms, not programs.

The tested version is PVS 3.2.

Further tools We list some other tools, which are of interest in the context of software verification, but which will not be discussed in detail.

Isabelle Isabelle can be obtained from `www.cl.cam.ac.uk/Research/HVG/Isabelle/`. It provides a generic theorem prover, supports higher-order logic, ZF set theory and a lot of other features. It is developed at the Cambridge University and TU Munich.

ACL2 ACL is both a programming language in which one can model computer systems and a tool for proving properties of those models. It can be found at `http://www.cs.utexas.edu/users/moore/acl2/`.

B-Method The B-Method includes the B-Tool and B-Toolkit and is in detail explained at `http://vl.fmnet.info/b/`.

Model Checking Tools Model checking tools have been very successful in real world applications during the last years. Examples are
SPIN (`http://spinroot.com`),
SMV (`http://www-2.cs.cmu.edu/~modelcheck/smv.html`) and
SLAM (`http://research.microsoft.com/slam/`) used for driver verification.

A more extensive overview and description on various tools can be found at the formal methods virtual library at `http://vl.fmnet.info/#notations`.

Related work In general there do not exist many comparisons in the style of this thesis. Special impact on this work had Griffioen and Huisman [1998]. They compare PVS and Isabelle by implementing some examples and investigating the logics behind these tools. They come to the result that both tools are very powerful, but also have some weak points. They give advice on how to combine both tools to obtain even better proofs. Zolda [2004] compares Isabelle and ACL2 and points out the different approaches. He comes to the conclusion that both tools have a lot of functionality but need a good background in logic. Freining et al. [2002] compare FPP with NPPV and SPARK. NPPV and SPARK have advantages in special tasks, but FPP is the winner in the specified test environment.

3.2 Frege Program Prover

The Frege Program Prover, from now on called FPP, was developed at the University of Jena, Department of Mathematics and Computer Science, Programming Languages and Compilers.

Find an introduction to FPP at `http://psc.informatik.uni-jena.de/Fpp/fpp-intr.htm`, whereas the system itself can be found at `http://psc.informatik.uni-jena.de/FPP/FPP-main.htm`.

FPP is an experimental system, implemented as a web application, for analysing the semantics of programs and for performing correctness proofs of annotated programs. FPP supports a subset of Ada — the concrete FPP syntax can be found at `psc.informatik.uni-jena.de/Fpp/fpp-synt.htm`. Identifiers are assumed to be integer or boolean variables.

FPP offers various capabilities:

Computation of the weakest precondition FPP computes the weakest precondition for a given program statement S in combination with a given postcondition Q. The internal mechanism is mainly based on the calculations of weakest preconditions as presented in section 2.5.

Check for the correctness of a program Given a precondition P, a program S and a postcondition Q, FPP checks whether the Hoare triple $\{P\}\ S\ \{Q\}$ is consistent. Such a triple is called consistent, if the program S satisfies the conditions stated by the precondition P and the postcondition Q. Hence FPP checks consistency by testing

$$P \supset wp(S, Q)$$

Theorem prover FPP can be used as a theorem prover by setting the precondition P to "true", the program S to "NULL" and the postcondition Q to the theorem that needs to be proved. The typical structure of such an FPP instance looks like:

```
--!Pre: true;
NULL;
--!Post: <theorem to be proved>;
```

It should be mentioned that this functionality is a logical consequence of the two previous capabilities. It is not considered to be competitive with other pure theorem provers.

For checking the correctness of a program, FPP needs to perform mathematical proofs. This is done with an extended version of Analytica (confer Clarke and Zhao [1993], Bauer et al. [1998]). FPP does the translation between the input program written in a subset of Ada into a representation that Analytica can handle. In the next step Analytica tests the Hoare triple on validity and returns the result. As Analytica is a main part of FPP for checking the correctness, it will be shortly presented.

Analytica Analytica is an automated theorem prover, originally intended for elementary analysis. It is written in the Mathematica programming language, which is based on term rewriting. In fact each step executed by Analytica is the application of a rewriting rule.

Mathematica provides a rule-based programming language. Mathematica rules are of the form `Pattern op Body` with `op` being one of `=`, `:=`, `->` or `:>`. The `Pattern` part describes a class of expressions, where a rule is applicable. The `Body` defines the expression to which the left side should be rewritten. Rules are either eager or lazy rules (depending on the `op` operator), specifying the details of the rewriting process. For details on this process look up a good book on term rewriting (e.g. Baader and Nipkow [1998]).

Analytica works in four phases: Skolemisation, simplification, inference and rewriting.

Skolemisation $\exists$ quantifiers are replaced by $\forall$ quantifiers. Within this translation it is necessary to consider free and bound variables and to introduce new function symbols. This is necessary to guarantee that the original formula is only satisfiable iff the skolemised formula is satisfiable.

Simplification Simplification is considered the most important step in Analytica. Simplification of a formula is executed in a so-called proof context. The proof context describes those formulas that may be assumed true during the simplification process. Analytica features various powerful rules to reduce the complexity of formulas.

Inference The inference phase is based on a sequent calculus as already presented in section 2.3. To Analytica rules have been added to provide easier handling, but the theoretic concept is the same.

Rewriting Rewriting implements those concepts presented above for Mathematica. Various tactics are available.

For details see Bauer et al. [1998], Winkler [1997] and Freining et al. [2002].

3.3 KeY System

The KeY system is developed and maintained by the Chalmers University of Technology, the University of Koblenz-Landau and the University of Karlsruhe. The online address for this project is `http://www.key-project.org/`.

According to Ahrendt et al. [2004], KeY allows to write formal specifications and to verify those specifications in the context of UML based software development. In detail the KeY tool consists of and uses:

Basis CASE tool KeY is an extension to UML CASE (Computer Aided Software Engineering) tools, mainly the commercial Together Control Center from the Borland Software Corporation. An integration in the Eclipse Java development environment is under construction.

OCL constraints OCL is short for Object Constraint Language (cf. OMG [2003a]). It is used to specify constraints on objects in the Unified Modelling Language (UML). UML is in the meantime an accredited and recognised standard in object-oriented software development processes (cf. OMG [2003b]). KeY uses OCL to define the formal specification. KeY supports the user in creating and analysing OCL constraints and offers especially:

Creation of OCL constraints KeY is able to generate automatically constraints by using design pattern instantiations. For standard formulations predefined instantiations of design pattern exist and can be easily used. On the other hand the user is free to formulate any valid OCL statement without assistance of KeY.

Formal analysis of OCL constraints Constraints on classes which affect other constraints are automatically recognised and their relations between each other are analysed irrespective of implementation details.

Verification of implementations From the given OCL constraints KeY can prove whether an implementation satisfies obligations. This way KeY can verify programs.

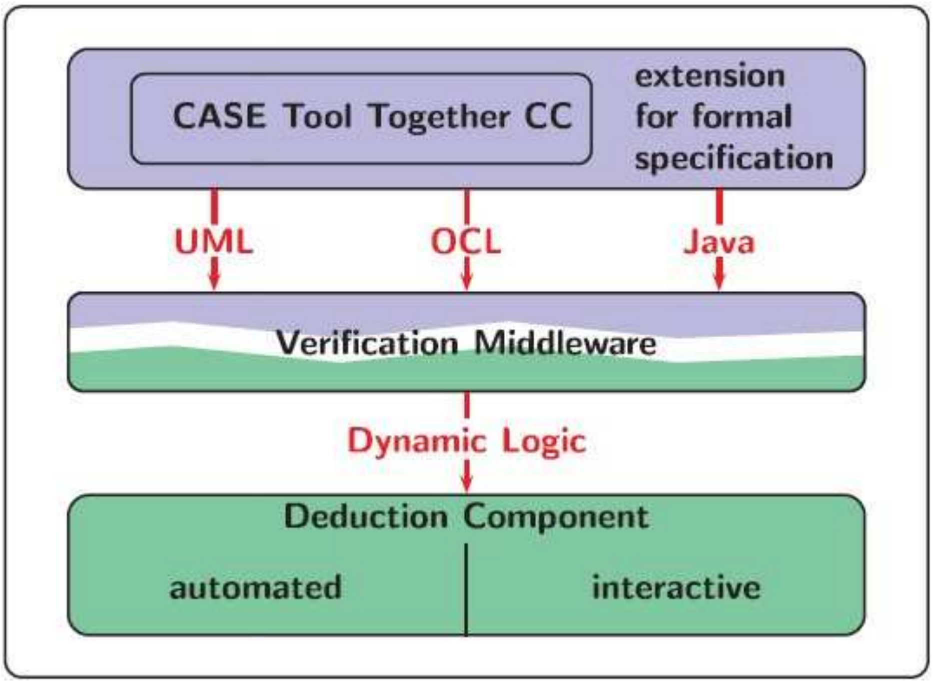

Figure 1: Architecture of the KeY system, Ahrendt et al. [2004, p. 3]

JavaCard The target language of KeY is JavaCard. JavaCard is a subset of the widely used Java programming language — both developed by Sun Microsystems — with some restrictions, tailored for applications with smart cards that need a secure environment. A description of the JavaCard language is described in Sun Microsystems [2003] and is not given in this document as it is very similar to the well-known Java programming language.

Architecture KeY consists of various components (Figure 1):

Modelling component As already mentioned, KeY allows the user to create, process and analyse OCL constraints. This component handles this task. The CASE tool is responsible for modelling and rendering UML elements, whereas KeY uses external tools to manipulate OCL constraints. In addition OCL specification templates are available in this component.

Verification middleware KeY internally uses a dynamic logic for JavaCard, which will be explained below. The verification middleware does now the translation of the OCL constraints, the JavaCard program code and the UML model in this JavaCard dynamic logic. This is then the input to the deduction component and hence connects the two layers modelling component and deduction component. Furthermore the proofs are managed and stored in this component.

Deduction component Based on the proof obligations resulting from the verification middleware, the deduction component tries to find proofs and discharge the proof obligations. The prover is interactive, but is designed to prove as much as possible automatically.

JavaCard Dynamic Logic Within KeY it is possible to specify preconditions, postconditions and class invariants. KeY allows now to check whether those assertions are valid after the execution of an implementation. The logic behind this procedure is a dynamic logic adapted for the JavaCard programming language.

Dynamic logic can be seen as an extension to classical Hoare logic that was already introduced in section 2.4. As presented in Ahrendt et al. [2004, p. 11 ff], deduction in this dynamic logic uses symbolic program execution and program transformations.

The dynamic logic is built from non-dynamic standard logic with some modal logic extensions. For every program statement in JavaCard an equivalent statement in JavaCard dynamic logic can be found — KeY has an 100% JavaCard coverage.

Similarly dynamic logic statements can be found for OCL constraints, as the complexity of OCL is not as high as the complexity of the whole JavaCard programming language.

Once the proof obligations have been collected, and all OCL and JavaCard statements have been transformed into the JavaCard dynamic logic, a deductive calculus is used: This calculus uses techniques like those presented in section 2.3, and is kind of sequent-style calculus. All in all there are about 250 rules as a lot of constructs need to be considered. Some examples are:

Active statement rules As JavaCard allows different scopes and blocks, it is useful to restrict the computation to only those program statements that have some impact on the further program execution. Then these statements are called active.

Assignment and update rules The idea is the same as in classical Hoare logic, but object-oriented programming style adds some difficulties. Typically assignments on objects are not as trivial as some references might need to be considered. Special rules were introduced to manage this problem in an efficient way.

Exception rules JavaCard encourages the use of `try-catch-finally` and exceptions, which are no issues in standard sequent calculus. Again rules that deal with this issue were added.

Taclets KeY introduced the principle of taclets in theorem proving: “A taclet combines the logical content of a sequent calculus rule with pragmatic information that indicates when and for what it should be used.” [Ahrendt et al., 2004, p. 14]. Taclets are considered powerful enough for theorem proving in combination with a relatively simple and convenient way for the user to write them. An excellent excursion to the topic of taclets can be found in Beckert et al. [2004].

3.4 Perfect Developer

The Perfect Developer system, abbreviated PD, was designed and developed at Escher Technologies Ltd. in England. The main online reference is `http://www.eschertech.com/products/index.php`.

PD claims to be a tool for developing software systems that can be verified automatically. As a result a typical development process with PD should incorporate various phases:

- The user starts with a formal definition of functional requirements, where typical definitions deal with safety properties and expected behaviours.

- Based upon this a formal model can be elaborated. Diagrams in the Unified Modelling Language (UML) are supported. This allows the user to specify behaviour in a standardised way, enabling broader use and common understanding. Abstract data and abstract operations are defined, non-determinism is reduced to its minimum. Many times this leads to executable models in a very early development phase of the standard software engineering process.

- The formal model from the previous phase can now be checked against requirements. The requirements can be stated easily in form of preconditions, postconditions, invariants and other assertions within PD.

- Once the model has been built to encompass all necessary requirements, the next step is to refine the model to an implementation. PD can generate code directly from the specification. Another way is to specify a separate implementation that is checked against the model. The implementation includes concrete data structures and executable statements of varying complexity.

- After passing the previous steps the result is a model and an implementation. Verification of the model against the implementation reveals either errors, incomplete and inaccurate specifications or guarantees valid program code.

- As a final step code for target platforms can be generated. There are C++, Java and Ada available. This allows the code to be used on virtually any important operating system or hardware platform.

The PD Language Perfect Developer uses a strongly typed specification and implementation language, offering manifold constructs to the user.

Standard types like bool, byte, char, int, real, string, ...

User defined enumerations An enumeration is defined as a collection of values with an ordering relation. In fact an enumeration is a class in PD, with operators for finding the lowest, the highest values, predecessor and successors.

User defined classes Object-oriented principles are implemented in a rigorous way: From abstract to final classes, single inheritance, polymorphism and dynamic binding are supported.

Class templates PD offers six structures for different well defined uses: sets, bags (multi-sets), sequences, pairs, triples and mappings. Mappings define a relation between elements from the input domain to the output range. Typical applications are lookup tables or relational databases. Numerous operations on these structures are built-in, as well as rich theories for proving assertions about them.

Unions PD allows to combine types to build others. The standard example are strings, which are just character sequences in the PD language. Another example are lists of some type in combination with void representing the end of the list or the null value in other programming languages.

Operator and function overloading As PD supports template mechanisms, PD can easily build distinct signatures for overloading operator or function symbols. This ensures type safety with user-friendly naming conventions.

Partial functions Within the typed logic partial functions with equality or arithmetic behaviour are possible. Functions are either interpreted, possibly-interpreted or uninterpreted.

Expressions Within PD a lot of expressions exist: Quantified expressions, type widening expressions, type enquiry expressions, heap expressions, conversion expressions, scope resolution and `?` as a shorthand for not yet specified behaviour. For details see Crocker [2001, chapter 5.4].

The PD Prover With the ongoing development of Perfect Developer the internal mechanisms were upgraded. At the beginning PD was influenced by approaches of Floyd and Hoare and the calculation of weakest preconditions (see sections 2.4 and 2.5).

Based on this approach a sequent calculus prover system was introduced. A further improvement was the use of a Rasiowa-Sikorsky deduction system. The main inference rules are:

Primary prover inference rules Those rules unify terms, expand functions or create meta-variables. In addition there exist manifold rules for standard connectives of the PD first order logic.

Term creation rules like transitivity rules are subject to this topic.

Hard-coded rewrite rules

Other rewrite rules

The next version of PD already used a resolution procedure and paramodulation with intense help of the built-in term rewriter component. The term rewriter is made up of two sets of rules:

Hard-coded rules Rules that are frequently used belong to this group or rules that simply cannot be parameterised.

Parameterised rewrite rules The larger set of rules is part of this group. The spectrum varies from rules with different operands to rules with preconditions.

The language of PD has the power of first order predicate calculus with some additional higher-order constructs. The prover is basically able to prove classical two-valued logic, what implies that higher-order logic statements are transformed first. Additional rules are necessary for object-oriented features, like polymorphism or dynamic binding.

Perfect Developer is also strongly influenced by the Verified Design-By-Contract principle:

Verified Design-By-Contract The Verified Design-By-Contract idea is build upon the principle of Design-By-Contract. Design-By-Contract can be characterised as a system of preconditions and postconditions that have to hold for a given program. The implementations differ regarding the degree of formalisation:

Comments Comments state conditions in the source code. The problem is that these conditions cannot be automatically checked as they are just comments to the compiler.

Annotations with run-time checks Special statements in the programming language generate code that does not influence the effect of the program but check invariant conditions during run-time. This allows the user to find errors in the testing phase.

Annotations with static analysis It would be optimal if the compiler (or prover) could check the conditions before the program is even executed. But for common programming languages this is almost impossible due to:

- Heavy use of pointers and complex data structures for relatively simple data
- Most languages were simply not designed for verification
- Standard programming languages lack powerful statements to express useful verification conditions

Verified Design-By-Contract addresses these problems: In Perfect Developer it is possible to construct specifications that consist of preconditions, postconditions, invariants and further annotations. The Perfect language is powerful enough to write expressive conditions ($\forall$ and $\exists$ constructs exist within Perfect) such that the behaviour of a program can be exactly defined.

As a result code should just serve as an implementation to the specification. Often it is not even necessary to provide code because PD can construct executable code from the single specification. This ensures maximum consistency between code and its behaviour described in the specification.

A more detailed explanation of (Verified) Design-By-Contract can be looked up in Crocker [2004b], Crocker [2004a], Crocker [2003b] and Crocker [2003a]. They discuss the topics presented here in a far more detailed way and should be consulted for deeper insight.

3.5 PVS Specification and Verification System

The Prototype Verification System PVS was developed at the Stanford Research Institute. The main project address is `http://pvs.csl.sri.com/`.

PVS is a prototype system for writing specifications and constructing proofs. PVS offers an expressive high-order logic in combination with semi-automatic proving. Hence user interaction is often necessary. PVS appears to the user as an extension to the common "Emacs" editor. The main components are:

Type-checker The high-order logic in PVS is strongly typed. The type-checker has to enforce those restrictions on variables and other program structures. The conditions generated for this job are called type correctness conditions (TCCs) in PVS.

Proof checker This component is the main part of PVS. Specifications are proved under consideration of TCCs and preconditions. The proving phase in PVS is implemented as a life-cycle process (cf. Owre et al. [1992, p. 3]):

- **Exploratory phase** The main objective is to provide meaningful output to the user in order to support the user in finding errors or improving the specification.
- **Development phase** Once the specification is fixed the proof goes into some more details. Very important is the efficiency of the proof development.
- **Presentation phase** The proof is prepared to be presented to the user. This requires the system to produce good and meaningful explanations to the user.
- **Generalisation phase** Once the proof is finished, the next aim is to weaken its precondition. This might strengthen the proof by providing a more general statement.
- **Maintenance phase** Maintenance is an extension to the idea of generalisation. This implies for the proof that its precondition may change. Therefore it might be necessary to redo parts of the proof to adapt to the new situation.

The PVS Language The lexical structure of the PVS language can be found in [Owre et al., 2001a, p. 7]. In addition the PVS Specification Language offers numerous powerful features:

Type declarations Available are uninterpreted, interpreted and enumerated types and subtypes.

Variable declarations PVS interprets variables as logical variables, not as program variables. As a result it is possible to use binding expressions such as `FORALL`, `EXISTS` or `LAMBDA`.

Constant declarations As PVS supports high-order logic the term constant applies to any n-ary function. Normal constants can be seen as 0-ary functions. As in type declarations, constants can be uninterpreted or interpreted.

Recursive definitions Recursive definitions are allowed in PVS, but it is necessary to give PVS information on termination. The user has to define a `MEASURE` function that decreases strictly on recursive calls.

Macros Macros are available for convenience use.

Inductive definitions It is possible to define a function or behaviour (e.g. predicate) in an inductive style. Some restrictions have to be guaranteed — for details refer [Owre et al., 2001a, p. 23 ff].

Formula Declarations Formulas are very important in PVS. Formulas can either be axioms, assumptions, lemmas, theorems or obligations (and many more). With them it is possible to describe the behaviour of programs in the specification.

Conversions Conversions are inserted automatically by the type-checker subcomponent, as soon it appears to be necessary.

Types PVS offers complex types, subtypes, function types, tuple types, record types and dependent types.

Expressions For the PVS Specification language various expressions are defined. The semantics of the structures is similarly to any other functional programming language. For an exact specification look at [Owre et al., 2001a, p. 43 ff] and Owre and Shankar [1999].

- Boolean Expressions
- `IF-THEN-ELSE` Expressions
- Numeric Expressions
- Applications

  Function applications as defined in mathematics.

- Binding Expressions
  The main binding expressions are λ and quantifiers.
- `LET` and `WHERE` Expressions
- Set Expressions
- Tuple Expressions
- Projection Expressions
- Record Expressions
- Record Accessors
- Override Expressions
- Coercion Expressions

Furthermore a lot of expressions are possible (especially tables and abstract data types). Again the interested reader may refer Owre et al. [2001a].

Theories Specifications in PVS are built from theories, that may be parameterised. The reason for theories was to provide maximal modularity and code-reusability.

The grammar in an extended Backus-Naur form for the PVS Language is defined in [Owre et al., 2001a, p. 83 ff, Appendix A].

The Logic of PVS The rules presented here are the theoretical background for the prover. Those rules are implemented in an efficient way but the idea works in the same way as presented here:

PVS internals heavily use sequent calculus. For propositional logic a short outline was given in section 2.3. The notation introduced in section 2.3 conforms with the notation used for PVS high-order logic sequent calculus.

The main connectives that PVS provides and hence need to be considered are:

$\neg$	`NOT`	
$\wedge$	`AND`	`&`
$\vee$	`OR`	
$\supset$	`IMPLIES`	`=>`
$\Longleftrightarrow$	`IFF`	`<=>`
$\forall$	`FORALL`	
$\exists$	`EXISTS`	
λ	`LAMBDA`	

All rules presented in section 2.3 are basis for the sequent calculus in PVS. Furthermore there are:

Equality Rules:

$$\text{if } a \equiv b \quad \frac{}{\Gamma \vdash a = b, \Delta}$$

$$\frac{a = b, \Gamma[b] \vdash \Delta[b]}{a = b, \Gamma[a] \vdash \Delta[a]}$$

with $\Gamma[e]$ denoting one or more occurrences of e in Γ.

Quantifier Rules:

$$(\forall\text{-l}) \quad \frac{\Gamma, A\left[{t \atop x}\right] \vdash \Delta}{\Gamma, (\forall x : A) \vdash \Delta}$$

$$(\forall\text{-r}) \quad \frac{\Gamma \vdash A\left[{a \atop x}\right], \Delta}{\Gamma \vdash (\forall x : A), \Delta}$$

$$(\exists\text{-l}) \quad \frac{\Gamma, A\left[{a \atop x}\right] \vdash \Delta}{\Gamma, (\exists x : A) \vdash \Delta}$$

$$(\exists\text{-r}) \quad \frac{\Gamma \vdash A\left[{t \atop x}\right], \Delta}{\Gamma \vdash (\exists x : A), \Delta}$$

with $A\left[{t \atop x}\right]$ means that in A all free occurrences of x are substituted by a term t (with possible renaming of bound variables). This way it is guaranteed that no free variables in t are captured, what is necessary for a correct high-order calculus. a has to be a new constant.

Conditional Rules:

$$\frac{\Gamma, IF(A, B[b], B[c]) \vdash \Delta}{\Gamma, B[IF(A, b, c)] \vdash \Delta}$$

$$\frac{\Gamma \vdash IF(A, B[b], B[c]), \Delta}{\Gamma \vdash B[IF(A, b, c)], \Delta}$$

$$\frac{\Gamma, A, B \vdash \Delta \qquad \Gamma, \neg A, C \vdash \Delta}{\Gamma, IF(A, B, C) \vdash \Delta}$$

$$\frac{\Gamma, A \vdash B, \Delta \qquad \Gamma, \neg A \vdash C\Delta}{\Gamma \vdash IF(A, B, C), \Delta}$$

with $IF(A, B, IF(C, D, E))$ as an abbreviation for `IF` A `THEN` B `ELSIF` C `THEN` D `ELSE` E `ENDIF`. The transformation of an $A[e]$ to $A[a]$ means that all occurrences of e in A are replaced by a.

More exhaustive descriptions about the logic behind PVS and its prover components can be looked up in Owre et al. [2001b]. Owre et al. [2001c] and Owre et al. [1992] give a good introduction into PVS, whereas some applications of PVS may be found in Owre et al. [1998].

4 Examples

In this section the four chosen tools FPP, KeY, PD and PVS are tested with examples that need to be verified. At first the criteria are defined to build a common evaluation platform. Second, the systems are evaluated according to this criteria. Third, the examples are presented and, finally, the result of the verification process of each tool is discussed.

4.1 Criteria

Criteria define the specific context and environment for practical tests. Thus it is relevant to specify them in a detailed way in order to provide a reproducible test scenario.

Tests of verification tools often end up in a single final result. All invested work is aggregated to the completion of a selected case study. During the evaluation of the criteria for this work, the decision to incorporate the whole process — from the problem formulation to the final proof — was taken. In fact this is similar to the idea mentioned in Bundy [2004].

As already mentioned in section 3.1 on page 17 our target groups are:

Software engineers with a good knowledge of computer science but without specific training in formal methods

Students of computer science and software engineering in the middle of their studies, being confronted with formal verification tools for the first time.

Criteria were chosen to test the capabilities in context for this target groups. The criteria are divided up into two categories:

Program related criteria This set of criteria is applicable for the whole verification tool and not restricted to specific examples. The defined criteria are as follows:

> **Commercial or academic nature** Gives some background on the tool.
>
> **Supported platforms and portability**
>
> **Installation**
>
> **General support**
>
> **Code generation** Assuming a verified specification, how easy is it to generate code? Automatic generation avoids introducing errors due to an error-prone manual translation.

  - **Learning curve** How long does it take to work with this tool in an efficient way? Does the system support the user in learning a successful process of verification?

- **Problem related criteria** These criteria make more sense to be looked at in a context for a specific problem.

  - **Ease of problem formulation** Here is discussed, whether the problem can be formulated in a natural and short way. Complex problem formulations are the first step for the introduction of errors and need to be avoided as far as possible.
  - **Complexity of user interaction** Does the system prove programs automatically? How often is it necessary to give hints to the prover or adapt the specification? How difficult is it to adapt the specification?
  - **Degree of coverage** Is it possible to prove a correct specification under all conditions?
  - **Support in finding errors** During the development process no user starts with a perfect specification. To which extent the system can help the user in finding problems or invalid specifications is relevant at this point.

It should be clear that some criteria are interchangeable within both categories under different test scenarios. In fact, the result of this test should encompass all criteria, and should lead to more insight in the specific advantages and disadvantages of each tool.

The criteria are deliberately neither rated nor weighted, as the impact of the various criteria differ too much under different scenarios. The plan is to give a comprehensive idea of how these tools work and which purpose they fulfil, not a ranking with winners and underdogs.

4.2 Methodology

The selected tools are tested with relatively easy and short standard computer science problems. This way it should be possible to implement different algorithms for most systems without having too much difficulties. This enables the reader to compare the systems in a practical environment, as the reader can follow various implementations for the same problem.

The problems handle topics of the following fields of interest:

Elementary number theory Factorial, Fibonacci numbers, sum, prime numbers, iterative multiplication and division

Array problems Inversions, Quicksort, List maximum

Weakest preconditions

It should be especially mentioned that some test scenarios or examples were taken in the style of already published papers: Some FPP examples are very similar to the examples presented in Freining et al. [2002], whereas the quicksort example for PVS was taken from Griffioen and Huisman [1998], as it shows the capabilities for PVS very well.

4.3 Frege Program Prover

An introduction to FPP and some theoretical background was given in section 3.2. As supposed in the foreword on the criteria in this section, program related criteria are discussed:

Commercial or academic nature FPP is an academic tool tailored for teaching purposes. FPP is still under development and future versions are announced to support more features and to offer more powerful provers.

Supported platforms and portability FPP is a web service. So portability and platforms are no point, virtually any device with a web browser is supported by FPP.

Installation Installation is not needed, making it very convenient for the user.

General support The team around FPP offered a lot of help. For any problems qualified solutions were found in short time via email, and the FPP team provided help in obtaining specific literature and academic articles on FPP.

Code generation As FPP supports a subset of Ada natively, no translation is necessary. Assertions are just comments, so a Ada compiler can generate code without further modifications.

Learning curve As FPP is considered as an academic teaching tool, the learning process should be clear and fast. And in fact FPP is quite easy to learn. With the examples on the FPP homepage and the simple syntax, the user is able to understand and write programs after a very short time.

4.3.1 Cubic sum

Compute the sum of cubed numbers.

Program code

```
--!pre: n > 0;
i := 0;
result := 0;
--!pre : n > 0 AND i = 0 AND result = 0;
--!post: result = n**2 * (n + 1)**2 / 4;
--!inv : result = i**2 * (i + 1)**2 / 4 AND i <= n;
--!term: n - i;
WHILE i < n LOOP
  i := i + 1;
  result := result + i**3;
END LOOP;
```

Prover output

```
--!pre       : (n >= 1)
--> wp       : (n >= 1)
--> vc       : (True)
--> Result: proved
i := 0;
result := 0;

--!pre       : (n >= 1 AND i = 0 AND result = 0)
--!post      : (result = n**2*(1 + n)**2/4)
--!inv       : (result = i**2*(1 + i)**2/4 AND i <= n)
--!term      : (-i + n)
-->functionality --------------------------
-->initial   : (n >= 1 AND i = 0 AND result = 0)
-->            ==> (result = i**2*(1 + i)**2/4 AND i <= n)
--> Result   : proved
-->induction : (n >= 1 + i AND
-->             result = i**2*(1 + i)**2/4 AND i <= n)
-->            ==> ((1 + i)**3 + result =
-->                (1 + i)**2*(2 + i)**2/4 AND 1 + i <= n)
--> Result   : proved
-->final     : (n <= i AND
-->            result = i**2*(1 + i)**2/4 AND i <= n)
-->             ==> (result = n**2*(1 + n)**2/4)
--> Result   : proved
-->termination ---------------------------
-->initial   : (n >= 1 + i AND
-->            result = i**2*(1 + i)**2/4 AND
-->            i <= n => -i + n >= 1)
--> Result   : proved
-->induction : (n >= 1 + i AND
-->            result = i**2*(1 + i)**2/4 AND i <= n)
-->            ==> (-i + n >= -i + n)
--> Result   : proved
WHILE i < n LOOP
   i := i + 1;
   result := result + i ** 3;
END LOOP;
```

Analysis

Ease of problem formulation The problem can be formulated in a clear and obvious manner. The assertions can be specified in an exact mathematical style, the termination function and invariants were not hard to guess.

Complexity of user interaction During the first attempts not everything could be proved automatically. It was necessary to refine the specification and to restrict the variable ranges. With a rigorous specification we achieved complete automatic verification.

Degree of coverage A total coverage was possible. This FPP program is asserted to stop in finite time and to be correct.

Support in finding errors As one can see from the quite clear prover output, it is traceable to find specification problems. The process of verification with induction is clear and reproducible.

4.3.2 Division

Calculate the quotient and remainder of a division for a given dividend and divisor.

Program code

```
--!pre: divisor > 0 AND dividend >= 0;
remainder := dividend;
quot := 0;
--!pre: divisor > 0 AND remainder = dividend AND quot = 0;
--!post: dividend = remainder + quot * divisor AND
         remainder < divisor;
--!inv: dividend = remainder + quot * divisor AND divisor > 0;
--!term: remainder;
WHILE remainder >= divisor LOOP
  remainder := remainder - divisor;
  quot := quot + 1;
END LOOP;
```

Prover output

```
--!pre       : (divisor >= 1 AND dividend >= 0)
--> wp       : (divisor >= 1)
--> vc       : (divisor >= 1 AND dividend >= 0 => divisor >= 1)
--> Result: proved
remainder := dividend;
quot := 0;

--!pre       : (divisor >= 1 AND remainder = dividend AND
                quot = 0)
--!post      : (dividend = divisor*quot + remainder AND
                divisor >= 1 + remainder)
--!inv       : (dividend = divisor*quot + remainder AND
                divisor >= 1)
--!term      : (remainder)
-->functionality ----------------------------
-->initial   : (divisor >= 1 AND remainder = dividend AND
                quot = 0)
-->          ==> (dividend = divisor*quot + remainder AND
                  divisor >= 1)
--> Result    : proved
-->induction :         (remainder >= divisor)
-->                AND (dividend = divisor*quot + remainder)
-->                AND (divisor >= 1)
-->            ==>     (dividend = -divisor +
                        divisor*(1 + quot) + remainder)
-->                AND (divisor >= 1)
--> Result    : proved
-->final     :          (remainder < divisor)
-->                AND (dividend = divisor*quot + remainder)
-->                AND (divisor >= 1)
-->           ==> (dividend = divisor*quot + remainder AND
                   divisor >= 1 + remainder)
--> Result    : proved
-->termination ---------------------------
-->initial   :          (remainder >= divisor)
-->                AND (dividend = divisor*quot + remainder)
-->                AND (divisor >= 1)
-->           ==> (remainder >= 1)
--> Result    : proved
-->induction :          (remainder >= divisor)
-->                AND (dividend = divisor*quot + remainder)
-->                AND (divisor >= 1)
-->           ==> (remainder >= 1 - divisor + remainder)
--> Result    : proved
WHILE remainder >= divisor LOOP
   remainder := remainder - divisor;
   quot := quot + 1;
END LOOP;
```

Analysis

Ease of problem formulation The problem can be formulated in a sophisticated way, but it is inevitable to specify the variable ranges (e.g. that `dividend >= 0, ...`) exactly. The only difficulty is finding a termination function that helps FPP in the verification process.

Complexity of user interaction The proof is done automatically, no adjustments were necessary in advance except slight modifications concerning nonnegative variable ranges.

Degree of coverage FPP can prove that this program fragment is correct.

Support in finding errors The first attempt for this program lacked the condition, that `divisor > 0`. It was not obvious from the output of FPP what was the exact problem. It took some time to identify the problematic part.

4.3.3 Factorial

Compute the factorial for a given number.

Program code

```
--!pre: n >= 0;
product := 1;
--!pre: product = 1 AND n >= 0;
--!post: product = factorial(n);
--!inv: product = factorial(i);
FOR i IN 1..n LOOP
     product := product * i;
END LOOP;
```

Prover output

```
--!pre       : (n >= 0)
--> wp       : (n >= 0)
--> vc       : (True)
--> Result: proved
product := 1;

--!pre       : (product = 1 AND n >= 0)
--!post      : (product = Factorial(n))
--!inv       : (product = Factorial(i))
-->functionality -------------------------
-->func      : (initial AND induction AND final AND null loop)
-->initial   :(1 <= n AND product = 1 AND
-->           n >= 0 => product = 1)
--> Result   : proved
-->induction :(1 <= n AND product = Factorial(-1 + i) =>
-->           i*product = Factorial(i))
--> Result   : proved
-->final     :(1 <= n AND product = Factorial(n) =>
-->           product = Factorial(n))
--> Result   : proved
-->null loop :(1 >= 1 + n AND product = 1 AND
-->           n >= 0 => product = Factorial(n))
--> Result   : proved
FOR i IN 1 .. n  LOOP
   product := product * i;
```

```
END LOOP;
```

Analysis

Ease of problem formulation Very natural formulation. For the assertions FPP supports the predefined function `factorial`, making it trivial to write invariants and postconditions.

Complexity of user interaction No user interaction was necessary.

Degree of coverage FPP could prove 100% on the first attempt.

Support in finding errors The prover output is short and obvious. Errors were not found.

4.3.4 Fibonacci numbers

Compute the n-th Fibonacci number.

Program code

```
--!pre: n >= 1;
previous := 0;
current := 1;
count := 1;
--!pre : n >= 1 AND previous = 0 AND current = 1 AND count = 1;
--!post: current = fib(n);
--!inv : count >= 1 AND count <= n AND previous = fib(count-1)
         AND current = fib(count);
--!term: n - count;
WHILE count < n LOOP
    x := current;
    current := current + previous;
    previous := x;
    count := count + 1;
END LOOP;
```

Prover output

```
--!pre       : (n >= 1)
--> wp       : (n >= 1)
--> vc       : (True)
--> Result: proved
```

```
previous := 0;
current := 1;
count := 1;

--!pre       : (n >= 1 AND previous = 0 AND current = 1
               AND count = 1)
--!post      : (current = (-(1 - Sqrt(5))**n +
                          (1 + Sqrt(5))**n)/(2**n*Sqrt(5)))
--!inv       : (count >= 1)
-->            AND (count <= n)
-->            AND    (previous)
-->                 = Fib((-1 + count))
-->            AND    (current)
-->                 = Fib((count))
--!term      : (-count + n)
-->functionality ---------------------------
-->initial   : (n >= 1 AND previous = 0 AND current = 1
               AND count = 1)
-->            ==>      (count >= 1)
-->                 AND (count <= n)
-->                 AND    (previous)
-->                      = Fib((-1 + count))
-->                 AND    (current)
-->                      = Fib((count))
--> Result    : proved
-->induction :          (n >= 1 + count)
-->                 AND (count >= 1)
-->                 AND (count <= n)
-->                 AND    (previous)
-->                      = Fib((-1 + count))
-->                 AND    (current)
-->                      = Fib((count))
-->            ==>      (1 + count >= 1)
-->                 AND (1 + count <= n)
-->                 AND    (current)
-->                      = Fib((count))
-->                 AND    (current + previous)
-->                      = Fib((1 + count))
--> Result    : proved
-->final     :          (n <= count)
-->                 AND (count >= 1)
-->                 AND (count <= n)
-->                 AND    (previous)
-->                      = Fib((-1 + count))
-->                 AND    (current)
-->                      = Fib((count))
-->            ==> (current = (-(1 - Sqrt(5))**n +
-->                           (1 + Sqrt(5))**n)/(2**n*Sqrt(5)))
--> Result    : proved
-->termination --------------------------
-->initial   :          (n >= 1 + count)
-->                 AND (count >= 1)
-->                 AND (count <= n)
-->                 AND    (previous)
-->                      = Fib((-1 + count))
-->                 AND    (current)
-->                      = Fib((count))
-->            ==> (-count + n >= 1)
--> Result    : proved
-->induction :          (n >= 1 + count)
-->                 AND (count >= 1)
-->                 AND (count <= n)
-->                 AND    (previous)
-->                      = Fib((-1 + count))
-->                 AND    (current)
-->                      = Fib((count))
-->            ==> (-count + n >= -count + n)
--> Result    : proved
WHILE count < n LOOP
   x := current;
   current := current + previous;
   previous := x;
   count := count + 1;
```

Analysis

Ease of problem formulation FPP supports the `fib` function. This helps to keep the formulation short and expressive. No tricks were necessary.

Complexity of user interaction It was necessary to refine the specification and to tighten variable ranges. Then FPP could prove the correctness.

Degree of coverage After those slight modifications a total coverage was reached.

Support in finding errors Although the program code is very short, the prover output is already quite long. Nevertheless the output does not offer deep insights into problematic issues for FPP.

4.3.5 Inconsistency test

The idea is to test the prover component with inconsistent input programs and look at its reaction and output.

Program code

```
--!pre: x = 0;
x := x - 1;
--!post: (EXISTS y: y >= 0 AND y <= 10: x = fib(fac(y))
         AND x = fac(fib(y)));
```

Prover output

```
--!pre       : (x = 0)
--> wp       : (exists((y)),
-->                   (y >= 0)
-->               AND (y <= 10)
-->               AND    (-1 + x)
-->                    = Fib((fac(y)))
-->               AND    (-1 + x)
-->           = fac(((-(1 - Sqrt(5))**y +
-->             (1 + Sqrt(5))**y)/(2**y*Sqrt(5)))))
--> vc       :      (x = 0)
-->            ==> (exists((y)),
-->                        (y >= 0)
-->                    AND (y <= 10)
-->                    AND    (-1 + x)
-->                         = Fib((fac(y)))
-->                    AND    (-1 + x)
-->                         = fac(Fib((y))))
--> Result: not proved
--> fc       : Not(     (0 <= Var(V(4)))
-->                    AND (-10 + Var(V(4)) <= 0)
-->                    AND (1/Sqrt(5))
-->                        * (-(1 - Sqrt(5))**(fac(Var(V(4)))))
-->                        + ((1 + Sqrt(5))**(fac(Var(V(4)))))
-->                             = (-2**(fac(Var(V(4)))))
-->                        AND    fac(   (2**(-Var(V(4))))
-->                                   * (1/Sqrt(5))
-->                        * (-(1 - Sqrt(5))**(Var(V(4))))
-->                        + ((1 + Sqrt(5))**(Var(V(4)))))
-->                             = (-1))

x := x - 1;
--!post      : (exists((y)),
-->                     (y >= 0)
-->                 AND (y <= 10)
-->                 AND    (x)
-->                      = Fib((fac(y)))
-->                 AND    (x)
-->           = fac(((-(1 - Sqrt(5))**y +
-->             (1 + Sqrt(5))**y)/(2**y*Sqrt(5)))))
```

Analysis

Ease of problem formulation Not important. The problem just consists of a statement and a postcondition stating a difficult and inconsistent assumption.

Complexity of user interaction No user interaction. The main point is to look at the prover output when FPP encounters inconsistent program code.

Degree of coverage Obviously this code cannot be verified.

Support in finding errors The prover output is very complex and unmanageable. FPP does not really make clear what the problems is, there is too much output on for the first moment irrelevant conditions. Of course the example was chosen this way to add complexity to the trivial example, but FPP has shown that it cannot produce good output on tricky input. In the examples before the output was often quite reasonable. Especially bigger programs with not so exact specifications appear often to a typical software engineer in his daily development process.

4.3.6 Multiplication

Multiply two integer numbers.

Program code

```
--!pre: x = u AND y = v AND u >= 0 AND v >= 0;
z := 0;
--!pre : x = u AND y = v AND x >= 0 AND y >= 0 AND z = 0;
--!post: z = u * v;
--!inv : x * y + z = u * v AND y >= 0;
--!term: y;
WHILE y > 0 LOOP
    z := z + x;
    y := y - 1;
END LOOP;
```

Prover output

```
--!pre       : (x = u AND y = v AND u >= 0 AND v >= 0)
--> wp       : (x = u AND y = v AND x >= 0 AND y >= 0)
--> vc       : (x = u AND y = v AND u >= 0 AND v >= 0)
-->            ==> (x = u AND y = v AND x >= 0 AND y >= 0)
--> Result: proved
z := 0;

--!pre       : (x = u AND y = v AND x >= 0
               AND y >= 0 AND z = 0)
--!post      : (z = u*v)
--!inv       : (x*y + z = u*v AND y >= 0)
--!term      : (y)
-->functionality --------------------------
-->initial   : (x = u AND y = v AND x >= 0
-->            AND y >= 0 AND z = 0)
-->            ==> (x*y + z = u*v AND y >= 0)
--> Result    : proved
-->induction : (y >= 1 AND x*y + z = u*v AND y >= 0)
-->            ==> (x + x*(-1 + y) + z = u*v AND -1 + y >= 0)
--> Result    : proved
-->final     :(y < 1 AND x*y + z = u*v AND y >= 0 => z = u*v)
--> Result    : proved
-->termination --------------------------
```

```
-->initial   :(y >= 1 AND x*y + z = u*v AND y >= 0 => y >= 1)
--> Result    : proved
-->induction :(y >= 1 AND x*y + z = u*v AND y >= 0 => y >= y)
--> Result    : proved
WHILE y > 0 LOOP
   z := z + x;
   y := y - 1;
END LOOP;
```

Analysis

Ease of problem formulation Very intuitive and straight forward.

Complexity of user interaction The constraints for the variables needed some further work, but could be added in short time.

Degree of coverage Total coverage after slight modifications.

Support in finding errors It was immediately clear that FPP needed constraints on the variables. The output again tends to get longish with those constraints. But all in all this example was no challenge for FPP.

4.3.7 False theorem test

Test FPP's internal prover component with a simple invalid theorem.

Program code

```
--!pre: a => b;
NULL;
--!post: a AND c => c AND (NOT b);
```

Prover output

```
--!pre       : (a => b)
--> wp       : (a AND c => c AND Not b)
--> vc       : ((a ==> b) AND a AND c => c AND Not b)
--> Result: not proved
--> fc       : (b AND c AND a)

NULL;
--!post      : (a AND c => c AND Not b)
```

Analysis

Ease of problem formulation Theorems can be stated very easily in FPP. The statement block is just `NULL` and the theorem to be tested is stated as a postcondition.

Complexity of user interaction Theorems can be proved or discharged completely automatically.

Degree of coverage Obviously 0%, as the input is an invalid theorem.

Support in finding errors The explanation is good and specifies a counter example.

4.3.8 Correct theorem test

Test FPP's internal prover component with a small valid theorem.

Program code

```
--!pre: true;
NULL;
--!post: (a => (b => a)) AND ((a AND (a => b)) => b);
```

Prover output

```
--!pre       : (True)
--> wp       : ((a AND b ==> a) AND (a AND (a ==> b) ==> b))
--> vc       : ((a AND b ==> a) AND (a AND (a ==> b) ==> b))
--> Result: proved
NULL;
--!post      : ((a AND b ==> a) AND (a AND (a ==> b) ==> b))
```

Analysis

Ease of problem formulation The same as in the previous example — very natural.

Complexity of user interaction No user interaction necessary.

Degree of coverage The valid theorem is totally proved by the prover component.

Support in finding errors Specification errors leading to invalid formulas result in instructive counter examples (cf. previous example).

4.3.9 Conditional weakest precondition

Test FPP's ability to compute weakest preconditions.

Program code

```
x := 3;
IF x + y < 19
THEN
  x := x + 19;
ELSE
  x := y + y;
END IF;
--!post: x > y;
```

Prover output

```
--> wp: (19 >= 4 + y AND 22 >= 1 + y OR 19 <= 3 + y AND
-->     2*y >= 1 + y)
x := 3;
IF x + y < 19 THEN
   x := x + 19;
ELSE
   x := y + y;
END IF;
--!post: (x >= 1 + y)
```

Analysis

Ease of problem formulation Not applicable.

Complexity of user interaction Weakest preconditions can be computed fully automatically by FPP, as long as loops are avoided.

Degree of coverage The system finds the weakest precondition. However, its inability to simplify expressions disguises the fact that the formula is equivalent to `TRUE`.

Support in finding errors Not applicable.

4.4 KeY System

KeY was already introduced in section 3.3. At this point KeY is analysed with examples and its practical nature is discussed:

Commercial or academic nature KeY is of academic nature. The tested version of KeY uses a commercial CASE tool, but the KeY prover component can be used separately. Future version will probably use Eclipse which is freely available.

Supported platforms and portability Official supported platforms are Linux, Solaris and Windows. Portability of the written code is given as long as a Java compiler exists on the platform. For most modern operating systems this should not be a problem and hence good portability is attested here.

Installation The installation is quite a complex task: KeY uses Borland Together Control Center as the basic CASE tool, what means extra installation. Together CC is not freely available in general (there exist evaluation and academic versions) and the installation process was tricky. It took some time to get KeY working.

General support The KeY team offered excellent support. As the tested version of KeY was still in alpha stage, help was quite often required, as some subcomponents were not implemented or did not work without modifications. At each point the KeY team gave support and did a great job in helping with verifying the examples. The support was a highlight in the work with KeY.

Code generation As the code is written in JavaCard, a standard Java compiler can build code. This avoids manual error-prone translation, making the whole process more secure.

Learning curve The familiarity with the widely used Java language probably helps many users in working with KeY. OCL constraints are relatively simple and can be learnt in a short time. A problem was the lack of documentation — but this is acceptable if a tool is still officially considered in alpha stage.

4.4.1 Cubic sum

Compute the sum of cubed numbers.

Program code

```
public class CubicSum {
    /**
     * @preconditions n >= 0
     * @postconditions 4 * result = n*n * (n+1)*(n+1)
     */
    public static int cubicSum(int n) {
        int i = 0;
        int result = 0;

        while (i < n) {
            i++;
            result += i*i*i;
        }

        return result;
    }
}
```

Analysis

Ease of problem formulation The problem can be formulated very easily in a procedural style in the Java syntax. The pre- and postconditions can also be stated in a clear manner according to the syntax of OCL constraints. A problem was that OCL does not support real numbers, which led at first to an exception. It was necessary to bring the factor 4 to the left side in the `@postconditions` otherwise the result would be typed as real according to KeY. But in fact this can never happen due to the internal structure of the formula.

Complexity of user interaction The amount of user interaction is unfortunately very high. The induction rules for the `while` construct are complex and tricky. Only with a lot of tricks from the KeY team we could reproduce the whole proof. Without such help it is really hard, what makes the life not easier for a standard software engineer.

Degree of coverage After a lot of tricks and with a lot of knowledge from the user KeY could verify the correctness of this program.

Support in finding errors As good the general support from the KeY team was, as intricate we found the reports and output from KeY. The integration of the KeY prover within the Together CC CASE tool and the fact that the prover has a graphical user interface are for sure good ideas, but the realisation was very confusing. It is hard enough to prove a correct program — finding errors from that is even harder with no special knowledge on the internals of the KeY prover.

4.4.2 Conditional

This example tests KeY on simple conditional statements. Especially the complexity of proofs for very simple programs is here an issue.

Program code

```
public class If {
    /**
     * @preconditions y > 1
     * @postconditions x > y
     */
    public static void ifTest(int x, int y) {
        x = 3;
        if ((x + y) < 19)
            x += 19;
        else
            x = y + y;
    }
}
```

Prover output Only a short excerpt from the complete prover output is presented here. In fact it is intended to show the user the complexity for such a short example. Hence for all other tested KeY examples the prover output is left out here and the interested user is advised to consult the external package with all examples and their solutions.

```
(branch "dummy ID"
(rule "imp_right" (formula "1"))
(rule "method_body_expand" (formula "2"))
(rule "greater" (formula "1"))
(rule "greater" (formula "2") (term "0"))
(rule "assignment_allnormalass" (formula "2"))
```

```
(rule "if_else_eval" (formula "2") (term "2")
      (inst "#boolv=_var8"))
(rule "eliminate_variable_declaration_boolean"
      (formula "2") (term "2"))
(rule "compound_less_than_comparison_1" (formula "2")
      (term "2") (inst "#v0=_var9"))
(rule "variable_declaration_allmodal" (formula "2") (term "2"))
(rule "eliminate_variable_declaration_int"
      (formula "2") (term "2"))
(rule "remove_parentheses_right" (formula "2") (term "2"))
(rule "assignment_addition" (formula "2") (term "2"))
(rule "less_than_comparison" (formula "2") (term "4"))
(rule "and_right" (formula "2"))
(branch "dummy ID"
(rule "imp_right" (formula "2"))
(rule "assignment_allnormalass" (formula "3") (term "2"))
(rule "if_else_split" (formula "3"))
(branch "dummy ID"
(rule "boolean_equal" (formula "1"))
(rule "true_left" (formula "1"))
(rule "compound_assignment_op_plus" (formula "3") (term "2"))
(rule "compound_int_cast_expression" (formula "3")
      (term "2") (inst "#v=_var10"))
(rule "variable_declaration_allmodal" (formula "3") (term "2"))
(rule "eliminate_variable_declaration_int"
      (formula "3") (term "2"))
(rule "remove_parentheses_right" (formula "3") (term "2"))
(rule "compound_addition_2" (formula "3") (term "2")
      (inst "#v1=_var12") (inst "#v0=_var11"))
(rule "variable_declaration_allmodal" (formula "3") (term "2"))
(rule "eliminate_variable_declaration_int"
      (formula "3") (term "2"))
(rule "assignment_allnormalass" (formula "3") (term "2"))
(rule "variable_declaration_allmodal" (formula "3") (term "4"))
(rule "eliminate_variable_declaration_int"
      (formula "3") (term "4"))
(rule "remove_parentheses_right" (formula "3") (term "4"))
(rule "assignment_allnormalass" (formula "3") (term "4"))
(rule "assignment_addition" (formula "3") (term "6"))
(rule "add_literals" (formula "3") (term "1"))
(rule "cast_4" (formula "3") (term "4"))
(rule "assignment_allnormalass" (formula "3") (term "4"))
(rule "method_call_empty" (formula "3") (term "2"))
(rule "empty_modality" (formula "3") (term "2"))
)
```

Analysis

Ease of problem formulation The problem formulation is very natural, almost trivial. Anyone starting with programming should be able to come up with this formulation.

Complexity of user interaction The complexity was considerable high, although the input program is definitely one of the easiest one can imagine. The prover output gives a first hint on the complexity of this job.

Degree of coverage In the end the correctness of this program could be proven.

Support in finding errors With a lot of knowledge it might be possible to find errors from the prover output directly, for most users it is a non trivial job. A support in its proper sense is not available.

4.4.3 Division

Calculate the quotient and remainder of a division for a given dividend and divisor.

Program code

```
public class Division {
    /**
     * @preconditions rem > 0 and divisor > 0
     * @postconditions rem@pre = rem + result * divisor and
                       rem < divisor
     */
     public static int divide(int rem, int divisor) {
        int quot = 0;
        while (rem >= divisor) {
            rem -= divisor;
            quot++;
        }
        return quot;
    }
}
```

Analysis

Ease of problem formulation The executable program part is written in a natural procedural style. The only thing that needed special considerations were the OCL constraints, especially that some variables need a restricted variable range. Nevertheless KeY offers here good help with the graphical OCL constraint builder.

Complexity of user interaction The proof is very complex, especially the induction is presented by KeY in a very technical way. Without the hints and the help of the encouraging KeY team the proof is hard to follow.

Degree of coverage The program could be proven after longish considerations on how to proceed.

Support in finding errors Due to the complexity of the whole verification process an extra support in finding errors is not available.

4.4.4 Factorial

The factorial for a given number shall be computed.

Program code

```
public class Fac {
    /**
     * @preconditions n >= 0
     * @postconditions result > 0
     */
    public static int fac(int n) {
        if (n == 0)
            return 1;
        else
            return (n * fac(n - 1));
    }
}
```

Analysis

Ease of problem formulation The recursive definition of the `fac` function is very intuitive and fully supported by JavaCard. The OCL constraints are very limited as they deny the use of function calls. On the one hand this keeps constraints short and clear, on the other hand powerful constraints are hard to write in this restricted formalism.

Complexity of user interaction The recursive definition makes it hard to get an idea on how to start with the verification process. A disadvantage is that the automatic proving strategies of KeY often do not really simplify the proving process. The required user interaction is relatively high.

Degree of coverage In the end the claim that the result is always positive could be verified.

Support in finding errors Once again, the proving process is quite complex, making it difficult to find errors.

4.4.5 List maximum

Find the maximum from a list of numbers.

Program code

```
public class ListMax {
    /**
     * @preconditions l.length() > 0 and l <> null
     * @postconditions result >= l.get(0)
     */
    public static int listMax(int[] l) {
        int max = l[0];
        for (int i=0; i < l.length; i++) {
            if (l[i] > max)
                max = l[i];
        }

        return max;
    }
}
```

Analysis

Ease of problem formulation Once more KeY shows its big advantage: the natural formulation due to the use of JavaCard. For any Java programmer the program is no challenge. The OCL constraints are not difficult either, but on the first attempt the OCL constraints were unintentionally written in a wrong syntax. The OCL parser accepted OCL constraints, but the prover could not handle them.

Complexity of user interaction Unfortunately the program could not be verified completely automatically, although the preconditions and postconditions are very simple.

Degree of coverage The postcondition could be proved under the assumption of the preconditions and the program statements.

Support in finding errors Due to the complex proving process it is hard to find any errors.

4.4.6 Multiplication

Multiply two integer numbers.

Program code

```
public class Mult {
    static int x;
    static int y;

    /**
     * @preconditions x >= 0 and y >= 0
     * @postconditions result = x@pre * y@pre
     */
    public static int mult() {
        int z = 0;
        while (y > 0) {
            z = z + x;
            y = y - 1;
        }

        return z;
    }
}
```

Analysis

Ease of problem formulation The program itself and the conditions are obvious and easy to formulate. Only for the OCL constraints one has to be careful about the non-negativity conditions and the reference to the variables at program start (with the `@pre` formalism).

Complexity of user interaction User interaction is necessary, again an involved induction proof for non-KeY experts.

Degree of coverage The whole program could be verified.

Support in finding errors None.

4.4.7 Prime

Test whether a nonnegative number is prime.

Program code

```
public class Prime {
    /**
     * @preconditions n >= 0
     * @postconditions n >= 0
     */
    public static boolean isPrime(int n) {
        if (n < 2)
            return false;
        for (int i=2; i < n; i++) {
            if ((n % i) != 0)
                return false;
        }

        return true;
    }
}
```

Analysis

Ease of problem formulation This program is somehow tricky: the program formulation is natural, but there were problems in specifying useful properties for prime numbers in the OCL formalism. Hence in the end the conditions were chosen as easy as possible. But this does not make it easier to prove the obligations, as the conditions do not correlate with the program.

Complexity of user interaction The complexity is very high, making the proving process very complicated.

Degree of coverage In spite of the simple conditions the program could not be fully verified.

Support in finding errors KeY did not help in finding errors in this context.

4.5 Perfect Developer

Some theoretical background on Perfect Developer was given in section 3.4, whereas the focus is here on its practical usage:

Commercial or academic nature PD is a commercial tool with special conditions for academic institutions.

Supported platforms and portability The development environment of PD and the Perfect compiler and verifier kit run under Windows and Linux. The generated code of PD is either Ada, C++ or Java and hence runs on any supported platform for these programming languages.

Installation The installation process is clear and well documented. On the officially supported platforms standard installation mechanisms prepare the system in order to start working immediately with PD. For Windows and certain recent Linux distributions the installation is automatic and is finished within minutes. For other Linux distributions some additional work may be necessary. PD, in particular the graphical user interface, relies on very recent versions of some libraries, which for instance are not available in the stable release of Debian Linux (woody) by default. Backporting and recompiling the libraries solves the problem.

General support Escher Technologies, the producer of PD, runs a mailing list, where questions of any nature were answered often within one or two days. The support is fast and competent.

Code generation The standard target languages of the automatic translation of the Perfect language are C++ and Java. So virtually any deployed operating system has support for at least one of these languages. So portability issues are not worth being further discussed.

Learning curve PD offers a lot of features, that take time to learn. For programmers not familiar with declarative or functional programming it might be a challenge to formulate programs in a non procedural way. Once the user has internalised the language structure the user is able to express specifications and programs in a concise and natural way.

4.5.1 Cubic sum

Compute the sum of cubed numbers.

Program code

```
function cubicSum(n: nat): nat
decrease n
^=  (
        [n = 0]:
            0,
        []:
            n^3 + cubicSum(n - 1)
    )
assert result = (n^2 * (n + 1)^2 / 4);
```

Analysis

Ease of problem formulation The problem can be easily formulated in a recursive style, which is fully supported by PD. The assertions are also clear.

Complexity of user interaction No user interaction during the proof is possible.

Degree of coverage Almost all obligations could be discharged. Type constraint restrictions could not be proven in this formulation — PD could not show that `n^3 + cubicSum(n - 1)` in combination with the assertion is of type `nat`.

Support in finding errors PD gives useful hints what seems to be the problem. Of course the user has to look carefully on this part but at least the user has some clue about PD problems.

4.5.2 Factorial

Compute the factorial for a given number.

Program code

```
function factorial(n: nat): nat
decrease n
^=  (   [n = 0]:
```

```
            1,
        []:
            n * factorial(n - 1)
    )
via
    var tot: nat != 1;
    loop
        var nn: nat != 0;
        change tot
        keep tot' = factorial(nn')
        until nn' = n
        decrease n - nn';
        nn! + 1, tot! * nn'
    end;
    value tot
end
assert result > 0;
```

Analysis

Ease of problem formulation PD supports an implementation part, as used in this example. The declarative formulation describes the behaviour, whereas the implementation part describes, how the computation shall be done. PD ensures that both parts fit together and verifies them against each other.

Complexity of user interaction No user interaction was necessary, the program was verified fully automatically.

Degree of coverage PD could prove the total correctness of this program formulation.

Support in finding errors The prover output for this program is very long (about 20 pages), but PD generates a separate file if it encounters problems. With some hints of PD the user has good chances to find the source of errors.

4.5.3 Intersection

Count how many elements from the first list also occur in the second one. Both lists have to be sorted.

Program code

```
function cardIntSec(A: seq of nat, B: seq of nat): nat
pre A.isndec, B.isndec
satisfy result = #(those x::A :- (exists y::B :- x = y))
via
    var cardCount: nat != 0;
    loop
        var a: nat != 0,
            b: nat != 0;
        change cardCount
        keep a' <= #A, b' <= #B, A.isndec & B.isndec &
            #(those x::A :- (exists y::B :- x = y))
            = cardCount' + #(those x::A.drop(a')
              :- (exists y::B.drop(b') :- x = y))
        until a' = #A | b' = #B
        decrease (#A + #B) - (a' + b');
        if
        [A[a] = B[b]]:
            cardCount! + 1,
            a! + 1;
        [A[a] < B[b]]:
            a! + 1;
        [A[a] > B[b]]:
            b! + 1;
        fi;
    end;
    value cardCount
end;
```

Analysis

Ease of problem formulation The declarative part was easy to formulate and written within minutes. The implementation is longish and was difficult to formulate.

Complexity of user interaction The implementation part was tricky to write and is error prone. Only after a lot of reformulations the number of non verifiable obligations could be reduced. PD still could not prove the invariant of this program.

Degree of coverage Due to the complex implementation part PD could not verify the whole program. Without the separate implementation part PD could verify the whole program and was still able to produce executable code. So the implementation with its improved performance behaviour leads to not verifiable program fragments for PD.

Support in finding errors PD gives good hints on errors or problematic formulations in the declarative part, but fails to give such good reports for the implementation part.

4.5.4 Inversions

Compute the number of inversions within a sequence of numbers.

Program code

```
function countInversions(A: seq of nat): nat
^=  (
        #flatten
        (
            for i::0..<#A
            yield for those j::0..<#A
                :- i < j & A[i] > A[j]
                yield pair of (nat, nat){i, j}
        )
    )
assert result < (#A)^2;
```

Prover output

```
Failed to prove obligation: Assertion valid
In the context of class: Examples
Obligation location: Examples.pd (19,9)
Condition defined at: Examples.pd (30,19)

To prove:
#flatten(
  (for i::0 .. <#A yield
     for those j::0 .. <#A :- (i < j) &
                       (A[j as nat] < A[i as nat])
     yield pair of (nat,nat) {i as nat,j as nat}))
                       < (#A ^ (2 as nat))

Reason: Exhausted rules
```

```
Could not prove:

(+ over for x::0 .. (-1 + #A) yield
  #(those x::(0 .. (-1 + #A)).ranb :-
    (A[j] < A[x]) and (j < x))) < (#A ^ 2)
```

Analysis

Ease of problem formulation The formulation is tricky and needs intricate operators to obtain the correct return type. Nevertheless the formulation is very short and compact.

Complexity of user interaction No modifications were necessary. The proving process by PD is done without any user interaction.

Degree of coverage All obligations can be discharged except the assertion `result < (#A)^2`. The proof requires induction, which is not supported by PD.

Support in finding errors As one can see from the prover output above, PD shows what the problem is in verifying this program, but gives no hint on how to solve it.

4.5.5 List maximum

Find the maximum from a list of numbers.

Program code

```
function listMax(l: seq of nat, maxValue: nat,
                 maxValueIndex: nat, i: nat):
                     pair of (nat, nat)
decrease #l
^=  (
        [#l = 0]:
            pair of (nat, nat){maxValue, maxValueIndex},
        []:
            (
                [l.head > maxValue]:
                    listMax(l.tail, l.head, i, i + 1),
                []:
                    listMax(l.tail, maxValue,
                            maxValueIndex, i + 1)
                )
        )
assert (#l > 0 & maxValue = 0 & maxValueIndex = 0 & i = 0) ==>
            (result.x = l.max);
```

Analysis

Ease of problem formulation The functional recursive definition allows a short formulation. Language constructs like `pair` and `seq` allow sophisticated return types.

Complexity of user interaction No extra user interaction was necessary.

Degree of coverage Everything except the final assertion can be proved. The final assertion requires induction, which is not supported by PD.

Support in finding errors PD signals that it cannot prove the final assertion, but gives no help how to solve this.

4.5.6 Multiplication

Multiply two integer numbers.

Program code

```
function mult(x: nat, y: nat, z: nat): nat
decrease y
^=  (
        [y = 0]:
            z,
        []:
            mult(x, y - 1, z + x)
    )
assert result = (x * y + z);
```

Analysis

Ease of problem formulation The formulation was very easy and clear. No tricks were necessary.

Complexity of user interaction None. Complete automatic verification.

Degree of coverage PD could verify the program completely.

Support in finding errors The output is clear and short for the trivial example.

4.5.7 Prime

Tests whether a nonnegative number is prime.

Program code

```
function factors(n: nat): seq of int
^=  (
        those x::2..(n-1) :- (n % x) = 0
    );

function isPrime(n: nat): bool
^=  (
        [n < 2]:
            false,
        []:
            factors(n) = seq of int{}
    )
assert (result | (n < 2)) = (forall i::2..(n-1) :-
                                 (n % i) ~= 0);
```

Analysis

Ease of problem formulation PD allows to write down the mathematical properties for primes in a very intuitive way. This minimises the probability of introducing intricate errors already in the design phase.

Complexity of user interaction No fine tuning is necessary. The prover works automatically.

Degree of coverage All properties could be verified.

Support in finding errors On the first attempt the assertion was specified in the wrong way. The user could see immediately that PD could not verify the final assertion.

4.5.8 Quicksort

Sorts a sequence of numbers according to the well known Quicksort algorithm developed by Tony Hoare.

Program code

```
function quickSort(numbers: seq of int): seq of int
decrease #numbers
^=
    (
        [#numbers = 0]:
            seq of int{},
        []:
            (
                let pivot ^= numbers.last;
                let pivotindex ^= <#numbers;

                let rest ^= numbers.remove(pivotindex);
                let smallerOrEqual ^= those x::rest :-
                                          x <= pivot;
                let bigger ^= those x::rest :- x > pivot;

                quickSort(smallerOrEqual).append(pivot)
                    ++ quickSort(bigger)
            )
    )
assert result.isndec;
```

Analysis

Ease of problem formulation The functional definition of the Quicksort algorithm is very intuitive and a real advantage. This avoids errors in a non declarative implementation.

Complexity of user interaction No additional help needs to be given to the PD prover.

Degree of coverage PD could not show that this Quicksort formulation yields the same properties as the built-in `isndec` property, since it requires induction.

Support in finding errors No support necessary.

4.6 PVS Specification and Verification System

PVS was already presented in section 3.5. Its characteristics are:

Commercial or academic nature PVS is of academic nature, with some commercial additions. The tool can be freely downloaded and used by any end user.

Supported platforms and portability Supported platforms are only Solaris and Intel Linux platforms. PVS uses heavily Emacs and LISP. Theoretically it is possible to port the application, but so far this has not been done.

Installation The installation is straightforward. It suffices to copy the whole build into a directory and call a small script to relocate relevant links. On a standard Linux system PVS finds immediately Emacs and necessary libraries without problems.

General support The PVS team runs various mailing lists. Competent and fast answers per e-mail make up a good support.

Code generation PVS is not intended to produce executable code. PVS is a specification language integrated with support tools and a theorem prover. So the code has to be written in the specification language, which then can be verified. A translation of such algorithms into real code is still necessary.

Learning curve PVS is hard to learn. PVS offers many possibilities during the proving phase. How and when to use them requires not only intensive training with PVS but also a profound knowledge of logic. PVS is highly interactive, advanced knowledge is definitively necessary.

4.6.1 Cubic sum

Compute the sum of cubed numbers.

Program code

```
cubicSum: THEORY
  BEGIN
    n: VAR nat

    cubicSum(n): RECURSIVE nat =
      IF n = 0 THEN 0 ELSE n^3 + cubicSum(n - 1) ENDIF
    MEASURE n

    cubicSum_test: LEMMA cubicSum(3) = 36
    cubicSum_formula: LEMMA cubicSum(n) = n^2 * (n + 1)^2 / 4

  END cubicSum
```

Proof procedure

```
cubicSum_test: (grind)
cubicSum_formula: (induct-and-simplify "n")
```

Analysis

Ease of problem formulation The recursive specification is natural and constitutes a nice mathematical way to describe this problem.

Complexity of user interaction The `(grind)` meta rule is powerful and discharges most obligations. After careful study of the documentation it is also clear that an induction on the recursively decreasing variable is the way to success.

Degree of coverage With the above commands the whole program could be verified.

Support in finding errors PVS hardly gives hints on structural problems. Often the proving process becomes so interactive and intricate that it is hard to follow PVS problems. The user has already to have a plan to use induction, otherwise he will fail.

4.6.2 Factorial

The factorial for a given number shall be computed.

Program code

```
fac: THEORY
  BEGIN
    n, x, y, z: VAR nat

    fac(n): RECURSIVE nat =
      (IF n = 0 THEN 1 ELSE n*fac(n-1) ENDIF)
    MEASURE (LAMBDA n: n)

    mul_mon: LEMMA FORALL x, y, z:
      (x > 0 AND y > z) IMPLIES x * y > x * z
    fac_non_neg: LEMMA FORALL x: fac(x) > 0
    fac_inc: LEMMA FORALL x: fac(x + 1) < fac(x + 2)
    fac_test: LEMMA fac(0) = 1
    fac_test2: LEMMA fac(5) = 120

  END fac
```

Proof procedure

```
fac_test: (grind)
fac_test2: (grind)
fac_non_neg: (induct "x")
             (grind) (grind)
mul_mon: (induct "x")
         (grind) (grind)
fac_inc: (skolem!)
         (expand "fac" :occurrence 2)
         (lemma mul_mon)
         (inst -1 "fac(1+x!1)" "x!1+2" "1")
         (prop)
         (grind) (lemma fac_non_neg) (grind)
                 (grind)
```

Analysis

Ease of problem formulation The function and the lemmas can be expressed in an intuitive functional style.

Complexity of user interaction The amount of user interaction is quite high, especially for `fac_inc`. The rewriting and instantiating of already proved lemmas is non trivial and needs insight into the structure of the proof.

Degree of coverage With the above mentioned tricks everything could be verified.

Support in finding errors Due to the complex proving procedure, PVS could not give any hints.

4.6.3 Inversions

Prove some properties on inversion pairs of numbers.

Program code

```
inv: THEORY
  BEGIN
    A: VAR ARRAY[nat -> nat]
    lenA: VAR nat

    %% A set is defined as a predicate in PVS
    inv(A, lenA): set[[nat, nat]] =
      { (i: below(lenA), j: below(lenA)) |
          i < j AND A(i) > A(j) }

    %% An array is a (total) function in PVS
    inv_test: LEMMA inv((LAMBDA (x: nat):
      IF x = 0 THEN 3 ELSE 1 ENDIF), 2) =
        add((0,1), emptyset[[nat, nat]])

    inv_null: LEMMA inv((LAMBDA (x: nat): 0), 0) =
      emptyset[[nat, nat]]

  END inv
```

Proof procedure

```
inv_test: (apply-extensionality)
          (grind)
inv_null: (apply-extensionality)
          (grind)
```

Analysis

Ease of problem formulation PVS makes it quite difficult to handle this problem. The first aspect is that arrays are just functions. This leads to difficulties with return types and getting the length or the number of elements of an array.

Complexity of user interaction Sets cause problems with the meta strategy `(grind)`. At first one has to use the mechanism of extensionality. It takes time to extract this from the documentation or the PVS mailing list.

Degree of coverage The properties could be proved, but with a significant amount of interactivity.

Support in finding errors PVS failed to give any support for this problem. In the beginning in `inv_test` the order of numbers in the arrays was unwillingly permuted. PVS gave no hints why it could not prove anything. It took hours to find the error manually.

4.6.4 Multiplication

Multiply two integer numbers.

Program code

```
mult: THEORY
  BEGIN
    c, x, y, z: VAR nat

    mult(x, y, z): RECURSIVE nat =
      IF y = 0 THEN z ELSE mult(x, y - 1, z + x) ENDIF
    MEASURE y

    mult_test: LEMMA mult(3, 5, 0) = 15
    mult_ok: LEMMA FORALL (x, y, z): mult(x, y, z) = x * y + z
    mult_add: LEMMA mult(x, y, z + c) = mult(x, y, c) + z
    mult_ok2: LEMMA FORALL (x, y): mult(x, y, 0) = x * y

  END mult
```

Proof procedure

```
mult_test: (grind)
mult_ok: (induct-and-simplify "y")
mult_add: (induct-and-simplify "y")
mult_ok2: (skosimp*)
          (rewrite "mult_ok" :subst ("z" 0))
```

Analysis

Ease of problem formulation The program itself can be easily formulated in a recursive style. The lemmas for the conditions are also easily stated with the help of quantifiers.

Complexity of user interaction At the first try it is hard to find out that the proof of `mult_ok2` requires the proof of a more general theorem, `mult_ok`. Only the latter can be proved directly by induction. `mult_ok2` is then obtained by instantiation and rewriting.

Degree of coverage With the rewriting complete coverage was possible. The high interactivity of the user shall be mentioned here explicitly.

Support in finding errors PVS gives only passive support in finding errors. By inspecting unprovable sequents one has to deduce which premises are missing or whether errors in the specification occurred.

4.6.5 Quicksort

Sort a sequence of numbers according to the well known Quicksort algorithm developed by Tony Hoare. This example is due to Griffioen and Huisman [1998].

Program code

```
sort[T:TYPE,<=:[T,T->bool]]: THEORY
BEGIN

 ASSUMING
   total: ASSUMPTION total_order?(<=)
 ENDASSUMING

 l,m: VAR list[T]
 e: VAR T
 i: VAR nat
 b: VAR bool
 x,y: VAR T
 p: VAR pred[T]

 IMPORTING list_adt[T]

 % def and lems on sorting.

 sorted_rec(l): RECURSIVE bool =
    null?(l) OR null?(cdr(l)) OR (car(l) <= car(cdr(l))
      AND sorted_rec(cdr(l)))
 MEASURE length(l)

 qsort(l:list[T]): RECURSIVE list[T] =
  IF null?(l) THEN null
  ELSE
     LET piv = car(l)
     IN append(qsort(filter(cdr(l),(LAMBDA e: e <= piv))),
               cons(piv,
               qsort(filter(cdr(l),(LAMBDA e: NOT e <= piv)))))
  ENDIF
  MEASURE length(l)

 qsort_sorted : LEMMA sorted_rec(qsort(l))

END sort

int_sort: THEORY
 BEGIN
  IMPORTING sort[int,<=]

  int_oke: LEMMA FORALL (l:list[int]): sorted_rec(qsort(l))

  qsort_test: LEMMA qsort((: 3, 1, 2 :)) = (: 1, 2, 3 :)

  qsort_test2:
    LEMMA qsort((: 4, 3, 5, 2, 1, 6, 6, 9, 8, 7 :)) =
      (: 1, 2, 3, 4, 5, 6, 6, 7, 8, 9 :)
 END int_sort
```

Proof procedure Due to the length of the proof, the proof was omitted here. It can be found in the source code package accompanying this thesis.

Analysis

Ease of problem formulation This example shows how powerful the PVS specification language is. Generic and modular theories with abstract data types are used here. PVS allows to write expressive formulas with short code.

Complexity of user interaction The proof is very complex with high user interactivity due to the massive use of PVS features. Also lists are somehow tricky, as it is sometimes necessary to give PVS hints on the used types (eg. `::nat`).

Degree of coverage All properties could be verified.

Support in finding errors Similar to the previous example.

5 Summary

The last two sections compared in detail the four selected tools: section 3 presented the tools from a theoretical point of view, whereas section 4 discussed the implementation of several examples to illustrate the differences and capabilities. This section summarises the results with respect to the two target groups, namely software engineers and students of computer science with a limited background in formal logic.

Frege Program Prover FPP supports a small subset of Ada consisting of typical imperative program structures like loop, case- and if-statements. The only data types available are integer and boolean. The language for specifying pre- and post-conditions is rather restricted. E.g., function definitions, recursive specifications and structured data types like arrays are not supported.

FPP is able to verify simple programs and to compute their weakest pre-conditions. The prover, Analytica, acts as a black box signalling either the validity of a formula or returning unprovable sub-formulas; formal proofs are not supplied.

Due to its simplicity and its web interface, FPP is easy to learn and use. It seems to be a valuable tool for illustrating the ideas of formal program verification in basic courses. It is not suitable, however, for advanced courses on the subject or for real world applications, as it is neither able to deal with standard examples from Gries [1989] and Dijkstra and Scholten [1990] involving arrays, nor does it support object-oriented features. Moreover, the terse output in pure ASCII makes it difficult to trace errors.

The KeY system The KeY system supports a subset of Java known as JavaCard, which is increasingly used for mobile and embedded devices. Verification is based on dynamic logic, a generalisation of the Hoare calculus. The system is integrated into a professional CASE tool (Borland's Together Control Center); an integration into the free Eclipse environment is under way. Objects and constraints can be specified using UML and OCL.

Java, UML, OCL, and CASE tools are familiar to software engineers and students alike, which helps in getting started. Nevertheless, KeY cannot be recommended for these target groups at present: the interactive prover and its interaction with the user are in their infancy (compare the example in section 4.4.2) and are inadequate for any serious use. Moreover, OCL is not expressive enough to specify complex program behaviour.

Considering that KeY is still in alpha stage, it seems to be worthwhile to reevaluate the system in a few years in order to see whether it lives up to expectations.

Perfect Developer PD consists of a full-fledged object-oriented programming language, Perfect, of a powerful automated theorem prover and of a code generator translating programs from Perfect to Java, Ada, and C++. A rich collection of built-in data types, classes, functions and theories allows the user to write concise specifications on a fairly abstract level.

PD is a technically mature product that is ready for use in a regular development process. However, software engineers will need some time to become sufficiently acquainted with the many features of Perfect. Moreover, at least a basic knowledge of formal logic is required to be able to interpret the prover output and to use it for detecting errors in the specification or in the program. Perfect Developer is also well suited for teaching advanced courses on formal program verification. Usually there will not be enough time to cover all features of Perfect. Therefore a tutorial is required that concentrates on just those elements of the language that are necessary to implement and verify instructive examples like those in Gries [1989].

PD is the only tool of the four that comes close to the ideal of automatic and easy program verification. But there are also still some shortcomings. One is that the prover currently does not support induction. Consequently certain recursive functions and loops cannot be verified by the system. Another weakness, at least from an academic point of view, is the lack of information concerning the inner workings of the prover. Ideally the logical rules used in correctness proofs should be open for inspection such that independent proof checkers can establish additional trust in the system.

Prototype Verification System PVS is a powerful interactive theorem prover, which has been used for various real world applications. In contrast to the other systems it does not generate verified program code, but proves properties of algorithms. The prover is versatile and offers many possibilities. It is automatic to a certain degree, but usually requires frequent user interactions.

Due to its many basic inference rules and tactics it takes a long learning

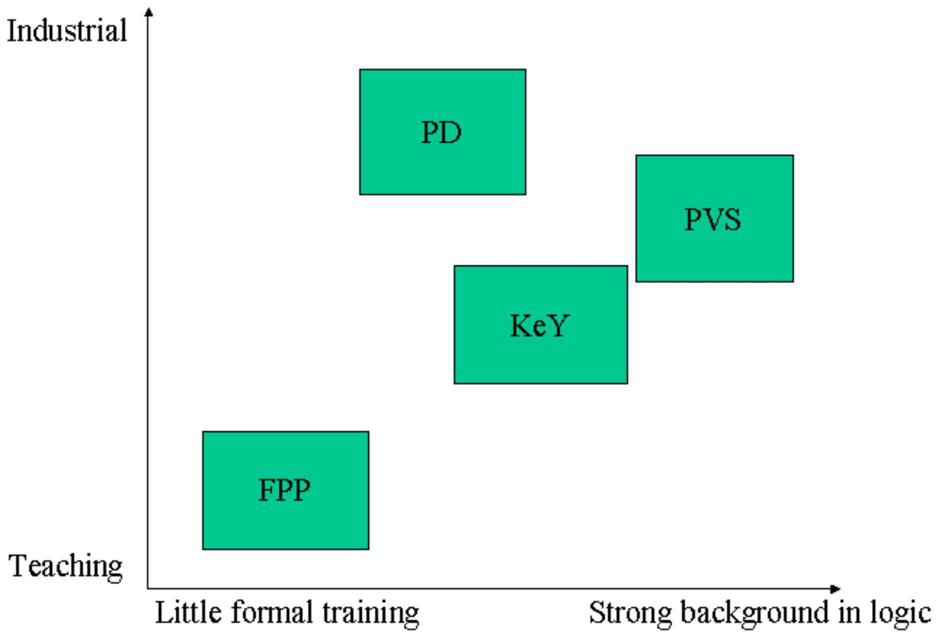

Figure 2: Comparison of FPP, KeY, PD and PVS

> phase to become familiar with the system. Moreover, users of PVS need a firm background in mathematics and formal logic to guide the prover. In our opinion typical software engineers and average students of computer science will have a hard time using PVS. Graduate or Ph.D. students might have a chance, provided they are given enough time. For courses with just a few hours per week in the lab PVS seems to be too complex.

Figure 2 compares the four selected tools FPP, KeY, PD and PVS according to formal background in logic and the field of application.

Tools for formal software verification have made considerable progress in recent years. With the advent of tools that offer formal methods on a level accessible to software engineers the costs for formal software verification will decrease such that it will be used in more and more projects. Universities have to react already today by training students in formal methods, using one or the other system.

Latest announcements have also affirmed that there is ongoing development in the field of software verification tools and the grand challenge towards the verifying compiler is more up-to-date than ever before. Nevertheless a lot needs to be done to achieve a wide acceptance of formal verification:

> Most of the general public, and even many programmers, are unaware of the possibility that computers might check the correctness of their own programs [Hoare, 2003, p. 65].

Resources

Additional material, like the examples and their source code, and this thesis are available online at `http://www.logic.at/staff/feinerer`.

References

Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter Schmitt. The KeY tool. *Software and System Modeling*, 2004. Online First issue, to appear in print.

Krzysztof Apt and Ernst-Rüdiger Olderog. *Programmverifikation*. Springer-Verlag, 1994. ISBN 3-540-57479-4.

Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

Andrej Bauer, Edmund Clarke, and Xudong Zhao. Analytica — An Experiment in Combining Theorem Proving and Symbolic Computation. *Journal of Automated Reasoning*, 21:295–325, 1998.

Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, and Steffen Schlager. Taclets: A new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas (RACSAM)*, 98(1), 2004. Special Issue on Symbolic Computation in Logic and Artificial Intelligence.

Alan Bundy. The Paradox of the Case Study, 2004. URL `http://www-unix.mcs.anl.gov/AAR/issuesept04/index.html#paradox`.

Edmund Clarke and Xudong Zhao. Analytica — A Theorem Prover for Mathematica. *The Mathematica Journal*, 3:56–71, 1993.

David Crocker. *The Perfect Developer Language Reference Manual*, September 2001.

David Crocker. Developing Reliable Software using Object-Oriented Formal Specification and Refinement. Escher Technologies Ltd., 2003a.

David Crocker. Perfect Developer: A tool for Object-Oriented Formal Specification and Refinement. *Tools Exhibition Notes at Formal Methods Europe*, 2003b.

David Crocker. Automated Reasoning in Perfect Developer. Escher Technologies Ltd., 2004a.

David Crocker. Safe Object-Oriented Software: The Verified Design-By-Contract Paradigm. In Redmill and Anderson, editors, *Proceedings of the Twelfth Safety-Critical Systems Symposium*, pages 19–41, London, 2004b. Springer-Verlag. ISBN 1-85233-800-8.

Edsger Dijkstra and Carel Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

Melvin Fitting. *First-order logic and automated theorem proving*. Springer-Verlag, 1990. ISBN 0-387-97233-1.

Carsten Freining, Stefan Kauer, and Jürgen Winkler. Ein Vergleich der Programmbeweiser FPP, NPPV und SPARK. *Ada-Deutschland-Tagung 2002*, pages 127–145, 2002. ISSN 1433-9986. URL `http://psc.informatik.uni-jena.de/Fpp/fpp-intr.htm#references`.

Jean Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Wiley, 2003. URL `http://www.cis.upenn.edu/~jean/gbooks/logic.html`.

John Gannon, James Purtilo, and Marvin Zelkowitz. *Software Specification: A Comparison of Formal Methods*. International Specialized Book Service Inc., September 1993.

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39, 1935.

David Gries. *The Science of Programming*. Springer-Verlag, 1989.

David Griffioen and Marieke Huisman. A comparison of PVS and Isabelle/HOL. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs '98*, volume 1479 of *Lecture Notes in Computer Science*, pages 123–142, Canberra, Australia, September 1998. Springer-Verlag.

Tony Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/363235.363259.

Tony Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/602382.602403.

Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, 2004. ISBN 0 521 54310X.

Cliff Jones. The Early Search for Tractable Ways of Reasoning about Programs. *IEEE Annals of the History of Computing*, 25:26–49, 2003. ISSN 1058-6180.

Bertrand Meyer. The grand challenge of Trusted Components. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 660–667. IEEE Computer Society, 2003. ISBN 0-7695-1877-X.

OMG. *Object Constraint Language Specification*, 2003a. URL `http://www.omg.org/docs/ptc/03-10-14.pdf`.

OMG. *OMG Unified Modeling Language Specification*, March 2003b. URL `http://www.uml.org/#UML2.0`.

Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag. URL `http://www.csl.sri.com/papers/cade92-pvs/`.

Sam Owre, John Rushby, Natarajan Shankar, and David Stringer-Calvert. PVS: an experience report. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullman, editors, *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Boppard, Germany, October 1998. Springer-Verlag. URL `http://www.csl.sri.com/papers/fmtrends98/`.

Sam Owre, John Rushby, Natarajan Shankar, and David Stringer-Calvert. *PVS Language Reference*, November 2001a.

Sam Owre, John Rushby, Natarajan Shankar, and David Stringer-Calvert. *PVS Prover Guide*, November 2001b.

Sam Owre, John Rushby, Natarajan Shankar, and David Stringer-Calvert. *PVS System Guide*, November 2001c.

Sam Owre and Natarajan Shankar. *The Formal Semantics of PVS*, March 1999.

Gernot Salzer. Theoretische Informatik 1. Institute of Computer Languages, Vienna University of Technology, June 2002.

Sun Microsystems. *Java Card 2.2.1 Platform Specification*, October 2003.

Dirk van Dalen. *Logic and Structure*. Springer-Verlag, 4th extended edition, 2004. ISBN 3-540-20879-8.

Jürgen Winkler. wp is Basically a State Set Transformer. Institute of Computer Science, Friedrich-Schiller-University, 1995.

Jürgen Winkler. The Frege Program Prover. *42. Internationales Wissenschaftliches Kolloquium, Technische Universität Ilmenau*, pages 116–121, 1997. ISSN 0943-7207.

Michael Zolda. Isabelle/HOL versus ACL2: Comparing Two Inductive Proof Systems. Institute of Computer Languages, Vienna University of Technology, 2004.