A Hoare-Style Calculus with Explicit State Updates

Reiner Hähnle and Richard Bubel[1]

Department of Computer Science and Engineering,
Chalmers University of Technology and Göteborg University

Abstract

We present a verification system for a variant of Hoare-logic that supports proving program correctness by forward symbolic execution. No explicit weakening rules are needed and first-order reasoning is automated. The system is suitable for teaching program verification, because the student can concentrate on reasoning about programs following their natural control flow and proofs are machine-checked.

Keywords: Hoare-logic, program verification, symbolic execution, partial correctness, total correctness, worst-case execution time

1 Introduction

An introduction to formal program verification is part of many courses and text books about Formal Methods (for example, [23,15]). Most of these use a variant of Hoare logic [13] or weakest precondition calculus [7] for a small imperative programming language. Teaching formal program verification on this basis comes with a number of challenges:

- Because of the assignment rule, one needs to compute explicitly weakest preconditions and, therefore, reasons backward through the target program. This is unnatural. The composition rule leads also often to backward reasoning in order to compute intermediate states.

- Even small proofs are tedious to do by hand and one tends to forget "trivial" assumptions such as upper/lower bounds. We found several by-hand proofs in lecture notes on program verification that could not be machine-checked, because of too weak preconditions or invariants.

- Checking first-order conditions is a distraction and requires to introduce first-order inference rules.

---

[1] Email: {reiner,bubel}@chalmers.se

The last two points could be easily addressed by a verification tool containing a sufficiently powerful first-order reasoner as an "oracle" to be invoked whenever program-free verification conditions are reached. Surprisingly, there seems to be no easy-to-use Hoare-style verification tool on the market serving that purpose. There are a number of verification systems for imperative languages used in research [24,18,17,22,2,5,4], but none of them is suitable for teaching purposes.

In this paper we present a verification system for a version of Hoare-calculus that addresses the problems described above: it is usable with minimal effort, it contains a clean separation between program and first-order rules, and it features a first-order reasoner tailored to verification tasks that can be presented as an oracle. We also address the first point: our program logic enables *forward* symbolic execution while still being based on a weakest precondition calculus. The technical device used to achieve this is an explicit notion of symbolic program states. We show that this introduces only minimal overhead, but has substantial advantages from a pedagogical view. The system is freely available and easy to install.[2] Our implementation is based on the KeY tool [4], one of the most powerful verification systems for Java.

Finally, as an extension of the Hoare-calculus for partial correctness, we implemented two more variants: one for total correctness and one for a limited form of worst-case execution time WCET reasoning. These are briefly presented in the appendix, Sect. A.8.

2 Background

2.1 Target Programming Language

We use a simple imperative **while** programming language:

$Program$ ::= $\big(Statement\big)$?
$Statement$ ::= $EmptyStatement$ | $AssignmentStatement$ |
  $CompoundStatement$ | $ConditionalStatement$|
  $LoopStatement$
$EmptyStatement$ ::= ';'
$AssignmentStatement$ ::= $Location$ = $Expression$';'
$CompoundStatement$ ::= $Statement\,Statement$
$ConditionalStatement$ ::= **if** '('$BooleanExp$')'
  '{'$Statement$'}' **else** '{'$Statement$'}'
$LoopStatement$ ::= **while**'('$BooleanExp$')' '{'$Statement$'}'
$Expression$ ::= $BooleanExp$ | $IntExp$
$BooleanExp$ ::= $IntExp\ ComparisonOp\ IntExp$ | $IntExp$ == $IntExp$ |
  $BooleanExp\ BooleanOp\ BooleanExp$ | $!BooleanExp$ |
  $Location$ | **true** | **false**
$IntExp$ ::= $IntExp\ IntOp\ IntExp$ | $\mathbb{Z}$ | $Location$
$ComparisonOp$ ::= < | <= | >= | >
$BooleanOp$ ::= & | | | == $IntOp$ ::= * | / | % | + | -

Locations are simply program variables. In general, they could be more complex structures, such as array or field accesses, but we will not discuss this here.[3] Locations and expressions are *typed*. There are two incomparable types called `boolean` and `int`. The type `int` denotes the mathematical integers $\mathbb{Z}$, not a finite integer type like in most real world languages like Java. Note that equality is overloaded. The grammar above is simplified in the sense that the real grammar uses common precedence rules for the different operators and allows of course parenthesised expressions. Obviously, the programming language defined here is a syntactic subset of the imperative fragment of Java [10].

2.2 First-Order Logic

In order to specify programs we use typed first-order logic. The only types allowed are `boolean` and `int`. Terms and formulas of first-order logic are defined as usual, with one notable exception: expressions of the programming language are also permitted as terms. This is ok, because expressions are side-effect free. Atomic formulas are either user-defined predicates or the special rigid *equality* symbol $\doteq$ which takes arbitrary terms as arguments. See the appendix for concrete formula syntax.

Program variables are (despite their name) not modelled as first-order variables but as constants (0-ary functions). Therefore, it is not possible to quantify over program variables. Further, we distinguish between *rigid* and *non-rigid* (or *flexible*) symbols. The difference is that rigid symbols are evaluated by a classical interpretation function and variable assignment. Their value is fixed and cannot be changed by a program. Uninterpreted rigid constants are often used to specify initial and final values of program variables. The availability of rigid functions and constants makes it easy to capture and refer to earlier program states and initial values.

In contrast, the value of *non-rigid* symbols depends on the current state in which they are evaluated. *Non-rigid* symbols can be changed by programs. In the presented logic the only *non-rigid* symbols are program variables.

Some useful *conventions*: program variables are typeset in typewriter font, logical variables in italic. When we specify a program π we assume that all program variables of π are contained in the first-order signature with their correct type.

The *semantics* of first-order formulas is interpreted over fixed domain models. Specifically, all boolean terms are interpreted over $\{\mathtt{true}, \mathtt{false}\}$ and all integer terms over $\mathbb{Z}$. Built-in function symbols are listed in the appendix. Apart from that, all semantic notions such as satisfiability, model, validity, etc., are completely standard, see, for example, [8].

2.3 Hoare Calculus

Before we define our own version we present a standard version of Hoare calculus [13]. As usual, the behaviour of programs is specified with *Hoare triples*:

$$\{P\}\,\pi\,\{Q\} \tag{1}$$

---

[3] The definitions given here are unsound in the presence of aliasing. General definitions of the concepts involved are found in [4].

Here, P and Q are closed first-order formulas and π is a program over locations $L = \{l_1, \ldots, l_m\}$. The meaning of a Hoare triple is as follows: *for each model $\mathcal{M}$ of P, if π is started with initial values $i_k = \mathcal{M}(l_k)$ $(1 \le k \le m)$ and if π terminates with final values f_k, then $\mathcal{M}^{f_1,\ldots,f_m}_{l_1,\ldots,l_m}$ is a model of Q.*

We can paraphrase this in a slightly more informal, but more intuitive, manner: for a given program π over locations $\{l_1, \ldots, l_m\}$, let us call an assignment of values $l_k = v_k$ $(1 \le k \le m)$ the *state s* of π. What the Hoare triple then says is that if we start π in any state satisfying the *precondition P*, if π terminates, then we end up in a final state that satisfies *postcondition Q*.

The standard Hoare rules are displayed in Fig. 1. We employ the following conventions for schematic variables occurring in the rules: e is an expression, b is a boolean expression, x is a program variable, s,s1,s2 are statements. P, Q, R, I are closed first-order formulas.

$$\text{assignment} \;\frac{}{\{P\{\texttt{x/e}\}\}\,\texttt{x=e;}\,\{P\}}$$

$$\text{composition} \;\frac{\{P\}\,\texttt{s1}\,\{R\} \qquad \{R\}\,\texttt{s2}\,\{Q\}}{\{P\}\,\texttt{s1}\ \texttt{s2}\,\{Q\}} \qquad\qquad \text{skip} \;\frac{}{\{P\}\,\texttt{;}\,\{P\}}$$

$$\text{conditional} \;\frac{\{P\,\&\,\texttt{b} \doteq \texttt{true}\}\,\texttt{s1}\,\{Q\} \qquad \{P\,\&\,\texttt{b} \doteq \texttt{false}\}\,\texttt{s2}\,\{Q\}}{\{P\}\,\texttt{if(b)\{s1\}else\{s2\}}\,\{Q\}}$$

$$\text{loop} \;\frac{\{I\,\&\,\texttt{b} \doteq \texttt{true}\}\,\texttt{s}\,\{I\}}{\{I\}\,\texttt{while(b)\{s\}}\,\{I\,\&\,\texttt{b} \doteq \texttt{false}\}}$$

$$\text{weakeningLeft} \;\frac{P \rightarrow Q \qquad \{Q\}\,\texttt{s}\,\{R\}}{\{P\}\,\texttt{s}\,\{R\}} \qquad\qquad \text{weakeningRight} \;\frac{\{P\}\,\texttt{s}\,\{Q\} \qquad Q \rightarrow R}{\{P\}\,\texttt{s}\,\{R\}}$$

$$\text{oracle} \;\frac{}{P} \quad (P \text{ any valid first-order formula})$$

Fig. 1. Rules of standard Hoare calculus.

3 Hoare Logic with Updates

The standard formulation of Hoare logic in Fig. 1 has a number of *drawbacks* in usability that are particularly problematic when used for teaching purposes:

- Because of the assignment rule, one needs to compute explicit weakest preconditions and, therefore, reasons backward through the target program.

- The compositional rule splits the proof and requires to have the intermediate state available.

- Weakening must be used before applying the rules for conditionals/loops. It would be better to delay weakening until first-order verification conditions are reached and let it be dealt with by an automated theorem prover.

- It is not easy to associate a node in a Hoare proof tree with a computation state of the target program.

We overcome these problems by introducing an explicit notation that describes finite parts of symbolic program states. This allows us to recast Hoare logic as forward symbolic execution.

3.1 State Updates

A (state) *update* is an expression of the form *Location* := *FOLTerm*. Actually, this is only the most simple form of an update, called *atomic update*. Complex updates are defined inductively: if $\mathcal{U}$ and $\mathcal{V}$ are updates, then so are $\mathcal{U}$, $\mathcal{V}$ (*sequential update*), and $\mathcal{U} \,||\, \mathcal{V}$ (*parallel update*). [4]

The more important of these is the parallel update. Consider a parallel update of the form $\mathcal{U} = l_1 := t_1 ||\cdots|| l_m := t_m$. Assume that we are in a computation state s. Then the update takes us into a state $s_{\mathcal{U}}$ such that:

$$s_{\mathcal{U}}(l) \;=\; \begin{cases} s(l) & \text{if } l \notin \{l_1, \ldots, l_m\} \\[2mm] t_k & \text{if } l = l_k \text{ and } l \notin \{l_{k+1}, \ldots, l_m\} \end{cases} \tag{2}$$

In words: the value of the locations occurring in $\mathcal{U}$ are overwritten with the right-hand side of the respective update. The second condition in the second clause ensures that the right-most update in $\mathcal{U}$ "wins" if the same location occurs more than once on the left-hand side in $\mathcal{U}$. Apart from that, all updates are executed in parallel. Updates are similar to a preamble or fixture as used in unit testing [19]: a piece of code that gets you into a certain state. There is, however, a difference between updates and code: the right-hand side of an update may contain any first-order term, not merely program expressions. This feature is often used to initialise a program with "arbitrary, but fixed" values.

The significance of parallel updates lies in the following property, formally proven in Lemma 3.2 below. Let us call two updates $\mathcal{U}$ and $\mathcal{V}$ *equivalent* if $s_{\mathcal{U}} = s_{\mathcal{V}}$ for any state s.

Lemma 3.1 *For each update $\mathcal{U}$ there exists a parallel update $\mathcal{V}$ that has the form $l_1 := t_1 \,||\, \cdots \,||\, l_m := t_m$ such $\mathcal{U}$ and $\mathcal{V}$ are equivalent.*

3.2 Hoare Triples with Update

We allow to write an update $\mathcal{U}$ in front of any program like this: $[\mathcal{U}]\pi$. If we are in state s the meaning is that the program is started in state $s_{\mathcal{U}}$. Within Hoare logic we use updates as follows:

$$\{P\}\,[\mathcal{U}]\,\pi\,\{Q\} \tag{3}$$

where, P, Q, and π are as above, and $\mathcal{U}$ is an update over the signature of P and π. We enclose updates in square brackets to increase readability. Either one of $\mathcal{U}$ and π can be empty. The meaning of this *Hoare triple with update* is as follows: if s

---

[4] There are further kinds of updates [21,4], but we do not need these here.

is any state satisfying the *precondition P* and we start π in $s_{\mathcal{U}}$, then, if π terminates, we end up in a final state that satisfies *postcondition Q*.

3.3   Hoare-Style Calculus with Updates

In Fig. 2 we state the rules of a Hoare calculus with updates that has some new features compared to standard Hoare calculus of Fig. 1:

- Composition is turned into left-to-right symbolic execution.
- Weakening is pushed below application of program rules and becomes part of first-order verification condition checking.
- We employ updates for handling assignments.

One advantage of weakest precondition calculation [7] as well as backward-execution style Hoare calculus is that an assignment can be computed by simple substitution and no renaming of old variables is necessary. The price to be paid for that is the not very intuitive backward-execution of programs. The KeY program logic uses updates to achieve weakest precondition computation with *forward* symbolic execution. In our eyes, this is a major pedagogical advantage: not only follows program rule application the natural execution flow in imperative programs, but the whole prove process is also compatible with established paradigms such as symbolic debugging.

In the KeY logic as well as in the present version of Hoare logic the rules have a "local" flavour in the sense that each judgement (node) in the proof tree relates to a symbolic state during program execution.

We use the same conventions for schematic variables as above, but in addition, let $\mathcal{U}$ be an update and s is either a statement or the empty string. The rules are depicted in Fig. 2. Let us briefly discuss each of them.

The *assignment* rule becomes easy: assignments are directly turned into updates. In our simple language, expressions have no side effects, so we do not need to introduce temporary variables to capture expression evaluation: we can directly turn e into the right-hand side of an update and later evaluate the semantic denotation. The same holds for guards. Because we moved composition of substitutions into updates, we can now evaluate programs left-to-right. The weakest precondition calculation is hidden in the update rules (see Fig. 3 below).

There is one new rule called exit that is applied when a program is fully symbolically executed. At this point, the update is applied which computes the weakest precondition of the symbolic program state $\mathcal{U}$ with respect to the postcondition Q. Then it is checked whether the given precondition implies the weakest precondition. The premise of the exit rule (as well as the left-most premise of the loop rule) are first-order verification conditions. This is indicated by a turnstile in order to make clear that we left the language of Hoare triples.

The conditional rule simply adds the guard expression as branch condition to the precondition. Of course, we must evaluate the guard in the current state $\mathcal{U}$. As said above, this formulation requires expressions to have no side effects. It has the advantage that path conditions can easily be read off each proof node.

$$\text{assignment} \;\; \frac{\{P\}\,[\mathcal{U},\, \mathtt{x} := \mathtt{e}]\,\mathtt{s}\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{x=e;s}\,\{Q\}}$$

$$\text{exit} \;\; \frac{\vdash\; P \mathrel{-\!\!>} \mathcal{U}(Q)}{\{P\}\,[\mathcal{U}]\,\{Q\}} \qquad\qquad \text{skip} \;\; \frac{\{P\}\,[\mathcal{U}]\,\mathtt{s}\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{;s}\,\{Q\}}$$

$$\text{conditional} \;\; \frac{\{P\,\&\,\mathcal{U}(\mathtt{b} \doteq \mathtt{true})\}\,[\mathcal{U}]\,\mathtt{s1;s}\,\{Q\} \qquad \{P\,\&\,\mathcal{U}(\mathtt{b} \doteq \mathtt{false})\}\,[\mathcal{U}]\,\mathtt{s2;s}\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{if(b)\{s1\}else\{s2\}s}\,\{Q\}}$$

$$\text{loop} \;\; \frac{\vdash\; P \mathrel{-\!\!>} \mathcal{U}(I) \qquad \{I\,\&\,\mathtt{b} \doteq \mathtt{true}\}\,[\,]\,\mathtt{s1}\,\{I\} \qquad \{I\,\&\,\mathtt{b} \doteq \mathtt{false}\}\,[\,]\,\mathtt{s}\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{while(b)\{s1\}s}\,\{Q\}}$$

Fig. 2. Rules of Hoare calculus with updates.

The loop rule is a standard invariant rule. We exploit again that expressions have no side effects, but also that we have no reference types. The chosen formulation stresses the analogies to the conditional rule. The first premise says that the precondition must be strong enough to ensure that the invariant holds after reaching the state at the beginning of the loop. In the second premise we are not allowed to use P, because P might have been affected by executing $\mathcal{U}$. In addition, we must reset the update to the empty one. In other words, started in *any* state where the loop invariant and condition hold the invariant must hold again after execution of the loop body. In practise, one uses as a starting point for the invariant those parts of P that are unaffected by $\mathcal{U}$. In those parts that *are* modified, one typically generalises a suitable term and adds that to the invariant.

3.4 Rules for Updates

We still need rules that handle our explicit state updates. Specifically, we need to (i) turn sequential into parallel updates (Lemma 3.1) and (ii) apply updates to terms, formulas, and other updates. For the first task we use a Lemma from [20] (in specialised form):

Lemma 3.2 *For any updates $\mathcal{U}$ and $\boldsymbol{x} := t$ the updates $\mathcal{U},\, \boldsymbol{x} := t$ and $\mathcal{U} \,||\, \boldsymbol{x} := \mathcal{U}(t)$ are equivalent.*

The resulting rule is depicted with the various update application rules in Fig. 3. These are rewrite rules that can be applied whenever they match. We use the same schematic variables as before and, in addition, t is a first-order term, $\mathcal{P}$ is a parallel update of the form $l_1 := t_1 || \cdots || l_m := t_m$, y is a logical variable, F is an n-ary function or predicate symbol, $\square$ is a propositional connective, and λ is a quantifier.

On top left is the rule that turns sequential into parallel updates. The second row contains rules for applying updates to program and logical variables. Note the similarity between the rule for program variables and (2) on p. 5. Logical variables are rigid and never changed by the updates. The third and fourth row contain rules for complex terms and for formulas. These are merely homomorphism rules. In quantified formulas, again, logical variables cannot be affected, but as they may

$$\mathcal{U}, \mathtt{x} := t \implies \mathcal{U} \, || \, \mathtt{x} := \mathcal{U}(t)$$

$$\mathcal{P}(\mathtt{x}) \implies \begin{cases} \mathtt{x} & \text{if } l \notin \{l_1, \dots, l_m\} \\ t_k & \text{if } \mathtt{x} = l_k \text{ and } l \notin \{l_{k+1}, \dots, l_m\} \end{cases} \qquad \mathcal{U}(y) \implies y$$

$$\mathcal{U}(F(t_1, \dots, t_n)) \implies F(\mathcal{U}(t_1), \dots, \mathcal{U}(t_n)) \qquad\qquad \mathcal{U}(P \square Q) \implies \mathcal{U}(P) \square \mathcal{U}(Q)$$

$$\mathcal{U}(\lambda y.\, P) \implies \lambda y.\, \mathcal{U}(P), \; y \notin \text{free}(\mathcal{U})$$

Fig. 3. Rewrite rules for update computation.

occur in updates one has to ensure that no name clashes occur ($free(\mathcal{U})$ returns the set of logical variables not bound in $\mathcal{U}$). On the whole it becomes clear that update application is basically substitution of program variables with their new values.

In fact, if we define standard substitution formally as rewrite rules, we need only two rules less! One of the additional rules is closely related to composition of substitutions. In the end we only have a very slight overhead due to the distinction between logical and program variables. Note that there is no rule to apply updates to programs. They accumulate until symbolic execution of the underlying program terminates.

4 Using KeY-Hoare

We illustrate how the system is used by proving correctness of a small program `countdown` that decreases a counter to 0:

```
while (timer>0) {
  timer = timer -1;
}
```

All variables are integers. Provided that the starting value of `timer` is non-negative, the program always terminates with the value of `timer` being 0. Let `startValue` be a rigid constant that captures an arbitrary initial value of `timer`. A suitable precondition is `startValue >= 0`. The postcondition can be stated simply as `timer = 0`. (Concrete formula syntax is defined in the appendix.) The initial Hoare triple with updates reads as follows:

$$\{\mathtt{startValue} >= 0\} \, [\mathtt{timer} := \mathtt{startValue}] \, \mathtt{countdown} \, \{\mathtt{timer} = 0\}$$

A file with an initial Hoare triple as proof obligation (in a simple format described in the appendix) is loaded to the KeY-Hoare system. Then the user can select a rule from Fig. 2 offered in a popup-menu after moving the mouse pointer over a Hoare triple and clicking (see screenshot below). There is exactly one applicable rule for each program construct and the system offers exactly this rule: the user experiences statement-wise symbolic execution of the target program. The only non-trivial interaction is to supply an invariant in a dialogue box that opens when the loop rule is applied. The invariant `timer >= 0` is sufficient.

8
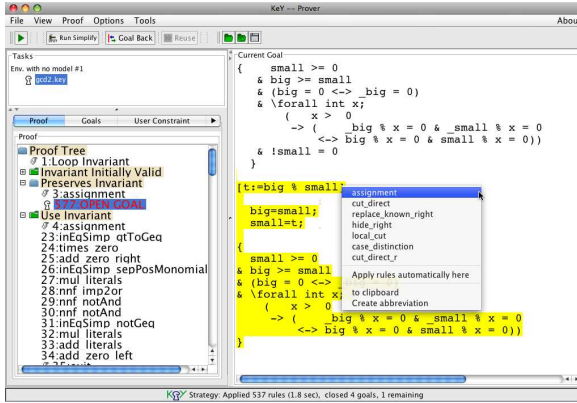
Whenever first-order verification conditions are reached, the system offers a rule Update Simplification that applies the update rules from Fig. 3 automatically. At this point, the user can opt to push the green Go button ▶. Then the built-in first-order theorem prover tries to establish validity automatically. For simple problems discussed in the introductory courses, such as `countdown`, this works quite well. If no proof is found, typically, the invariant or the specification (or the code!) is too weak or simply wrong. Inspecting the open goals usually gives a good hint. The system allows the student to follow symbolic execution of the program and to concentrate on getting invariants and specification right. First-order reasoning is left to the system. It is possible to inspect and undo previous proof steps as well as to save and load proofs.

5 Related Work

The tutoring tool for Hoare Calculus ITS, described and evaluated in [9], does not realise a reasoning system or proof checker. Students can fill out missing Hoare triples in two different notations. ITS checks whether related triples in the different notations have the same denotation and it determines the order in which triples were filled in to see whether students used forward or backward reasoning. Another educational tool for Hoare Calculus is J-Algo [16], a general modular framework that allows to visualise algorithms and comes with a module for the Hoare Calculus. While there is support for stepwise construction of a syntactically valid Hoare proof tableau, the lack of a reasoning system does not allow to obtain machine checked proofs. Our state updates are closely related to generalised substitutions used by the B method [1] and to Abstract State Machines [6]. A full discussion is contained in [21]. There are versions of Hoare logic that use the assignment rule from dynamic logic [11] in which case forward symbolic execution can be realised, however, at the price of introducing existentially quantified variables that hold the result of intermediate states. This is complicated to explain and difficult to use.

6 Conclusion, Future Work

We presented a verification system for a variant of Hoare-logic that supports proving by forward symbolic execution. No explicit weakening rule is needed and first-order reasoning is automated. The system is suitable for teaching program verification, because the student can concentrate on reasoning about programs following their natural control flow and proofs are machine-checked. The KeY-Hoare tool is freely available and can be easily installed (see Appendix). It is based on a state-of-art verification system for Java [4]. The KeY-Hoare tool is currently used in the course

Program Verification intended for Bachelors in their final year at Chalmers University. [5] Course materials including slides, examples, exercises, and exam questions are available from the authors.

At the moment, the GUI of the KeY-Hoare tool contains several elements that are inherited from the full Java version and are not useful in the more specialised context. It should be cleaned up and simplified. The current version of KeY-Hoare does not support arrays as Java arrays are too complicated for an introductory course. It would be easy, however, to implement value-type arrays and we plan to do this soon. In a similar vein, we will also add static method calls. All this is very easy, because it can be derived from simplifying corresponding Java constructs.

Acknowledgements

References

[1] J.-R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, Aug. 1996.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: an overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.

[3] B. Beckert, M. Giese, R. Hähnle, V. Klebanov, P. Rümmer, S. Schlager, and P. H. Schmitt. The KeY System 1.0 (deduction component). In F. Pfenning, editor, *Proc. 21st Conference on Automated Deduction (CADE), Bremen, Germany*, volume 4603 of *LNCS*, pages 379–384. Springer-Verlag, 2007.

[4] B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2006.

[5] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In K. Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer-Verlag, 2005.

[6] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[7] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[8] M. C. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, New York, second edition, 1996.

[9] K. Goshi, P. Wray, and Y. Sun. An intelligent tutoring system for teaching and learning Hoare logic. In *ICCIMA '01: Proceedings of the Fourth International Conference on Computational Intelligence and Multimedia Applications*, page 293. IEEE Computer Society, 2001.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification–Third Edition*. The Java Series. Addison-Wesley,, 2004.

[11] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, Oct. 2000.

[12] T. Harmon and R. Klefstad. A survey of worst-case execution time analysis for real-time Java. In *Proc. Parallel and Distributed Processing Symposium*, pages 1–8. IEEE Press, 2007.

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, Oct. 1969.

[14] J. J. Hunt, F. B. Siebert, P. H. Schmitt, and I. Tonin. Provably correct loops bounds for realtime java programs. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 162–169, New York, NY, USA, 2006. ACM.

---

[5] See http://www.cs.chalmers.se/Cs/Grundutb/Kurser/prove.

[15] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press, second edition, 2004.

[16] J-Algo—The Algorithm Visualization Tool, 2007. http://j-algo.binaervarianz.de/.

[17] C. Marché and C. Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In J. Hurd and T. Melham, editors, *Proc. 18th Intl. Conference on Theorem Proving in Higher Order Logics, Oxford, UK*, Lecture Notes in Computer Science. Springer-Verlag, Aug. 2005.

[18] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[19] G. J. Myers. *Art of Software Testing*. John Wiley & Sons, second edition, 2004.

[20] P. Rümmer. Proving and disproving in dynamic logic for Java. Licentiate Thesis 2006–26L, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, 2006.

[21] P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In M. Hermann and A. Voronkov, editors, *Proc. Logic for Programming, Artificial Intelligence and Reasoning, Phnom Penh, Cambodia*, volume 4246 of *LNCS*, pages 422–436. Springer-Verlag, 2006.

[22] K. Stenzel. *Verification of Java Card Programs*. PhD thesis, Fakultät für angewandte Informatik, University of Augsburg, 2005.

[23] R. D. Tennent. *Specifying Software: a Hands-On Introduction*. Cambridge University Press, 2002.

[24] D. von Oheimb. Hoare logic for java in isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

A   A Brief Reference Manual for KeY-Hoare

A.1   Installation and Running

KeY-Hoare is available from http://www.key-project.org/download/hoare/. Besides compilation from the source code, we offer a pre-compiled bytecode version and installation via *Java Web start* technology. Detailed installation instructions can be found on the website. Here, we briefly describe the Web Start installation.

Java Web Start is included in all recent JDK and JRE distributions of Java. It provides a simple way to run and install Java applications similar to applets.

Most browsers already know how to handle Java Web Start URLs; if not, one needs to associate the file type `jnlp` with the application `javaws`. After that a click on the Web Start link of the KeY website loads and starts KeY-Hoare. [6] One can use Web Start also from the command line:

  javaws http://www.key-project.org/download/hoare/download/webstart/KeY-Hoare.jnlp

After the first start one can start KeY-Hoare offline by simply executing `javaws` (Java 6: `javaws -viewer`) and selecting the KeY-Hoare entry in the list of available applications.

A.2   Formula Syntax

The following predicate symbols are predefined: `>`, `>=`, `<`, `<=`, `=`. The following function symbols are predefined: `+`, `-`, `*`, `/`, `%`. All have their usual signature and meaning and are supported by the automated theorem prover module of KeY-Hoare. For modulo we use the definition `a % b = a-(a/b)*b`. As a consequence, `0 % b = 0` and `a % 0` is undefined. Infix notation and the usual precedence rules are supported.

The concrete syntax of propositional connectives is `!`, `&`, `|`, `->`, `<->` with obvious meaning. First-order quantified formulas are written as follows:

QuantifiedFormula ::= *Quantifier Type LogicalVariable* ; *FOLFormula*
Quantifier ::= `\forall` | `\exists`

Example A.1 The following formula expresses that any common divisor `x` of the integers `a` and `b` is as well a divisor of the integer `r`.

  `\forall int x; ((x > 0 & a % x = 0 & b % x = 0) -> r % x = 0))`

A.3   Input file format

An input file for KeY-Hoare must have either `.key` or `.proof` as file extension. By convention `.key` files contain only the problem specification, i.e., the program together with its specification. In contrast `.proof` files include proofs (or proof attempts) and are created when saving a proof.

A grammar for input files is given in Fig. A.2. An example that illustrates the format is in Fig. A.1. An input file consists of three sections:

---

[6] It is also necessary to accept a certificate issued by the KeY project.

```
\functions {
    int startVal;
}

\programVariables {
  int timer;
}

\hoare {
{ startVal >= 0 }

[timer := startVal]
\[{
      while (timer>0) {
           timer = timer -1;
      }
}\]

 {
   timer = 0
 }
}
```

Fig. A.1. Input file for the `countdown` example.

(i) The section starting with keyword `\functions` declares all required rigid function symbols used, for example, to assign input program variables to an arbitrary but fixed value as described in Section 3.1. In Fig. A.1 this section declares an integer constant `startValue` used as initial value for the program variable `timer`.

(ii) The section starting with keyword `\programVariables` declares *all* program variables used in the program. Local variables declarations within the program are not allowed. Multiple declarations are permitted.

(iii) The section starting with keyword `\hoare` contains the Hoare triple with updates to be proven valid, i.e., it contains the program and its specification. As is illustrated in Fig. A.1, the initial update usually contains an assignment of fixed but arbitrary logical values to the input variables of the program.

A.4 Loading Problems

After starting KeY-Hoare (see Sect. A.1) the prover window becomes visible (the screenshot on p. 8 is displayed in enlarged form in Fig. A.3). The prover window consists of a menu- and toolbar, a status line and a central part split into a left and a right pane. The upper left pane displays a list of all loaded problems. The lower left pane offers different tabs for proof navigation or strategy settings. The right pane displays the currently selected subgoal or an inner proof node.

Before we explain the various subpanes in more detail, the first task is to load

InputFile ::= *Functions? ProgramVariables? HoareTriple?*

Functions ::= **\functions** '{' *FunctionDeclaration**'}'
FunctionDeclaration ::= *Type Name* ('(' *Type*(','*Type*)*')')?';'

ProgramVariables ::= **\programVariables** '{'*ProgramVariableDeclaration**'}'
ProgramVariableDeclaration ::= *Type Name*(',' *Name*)*';'

HoareTriple ::= (**\hoare** | **\hoareTotal** | **\hoareET**) '{'
                   PreCondition Update Program PostCondition
   '}'

PreCondition ::= *FOLFormula*

Update ::= '[' (*AssignmentPair* (||*AssignmentPair*)*)? ']'
AssignmentPair ::= *Name* ':=' *FOLTerm*

Program ::= '\[{' *WhileProgram* '}\]'
PostCondition ::= *FOLFormula*

Type ::= **int** | **boolean**
Name ::= character sequence not starting with a number

Fig. A.2. Input file grammar

a problem file. This can be done either by selecting `Load` in the `File` menu or by clicking on the icon ◼ in the toolbar (◼ reloads the most recently loaded problem). In the file dialogue window that pops up the users can choose one of the examples provided (e.g., `countdown.key`) or their own files.

After the file has been loaded the right pane of the prover window displays the Hoare triple as specified in the input file. The proof tab in the left pane should display the proof tree consisting of a single node. The first time during a KeY-Hoare session when a problem file is loaded the system loads a number of libraries which takes a few seconds.

A.5   Proving

First a few words on the various parts of the prover window. The upper part of the left pane displays all loaded problems. The lower part provides some useful tabs:

The Proof tab shows the constructed proof tree. A left click on a node updates the right pane with the node's content (a Hoare triple with updates). Using a right click offers a number of actions like pruning, searching, etc.

The Goals tab lists all open goals, i.e., the leaves of the proof tree that remain to be justified.

The Proof Search Strategy tab allows to tune automated proof search. The strategy for KeY-Hoare only allows to adjust the maximal number of rule applica-
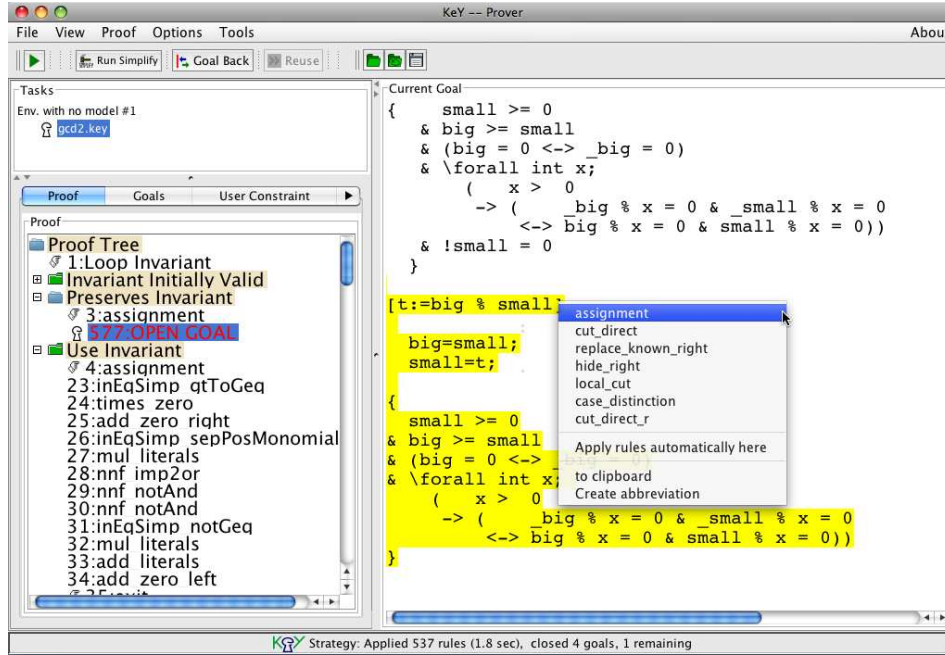
Fig. A.3. Screen shot of KeY-Hoare system.

tions before an interactive step is required, and (de-)activation of the `autoresume` mode.

All other tabs are not important for KeY-Hoare and will be removed in future versions.

The right pane displays the content of a proof node in two different modes depending on whether the node is (a) an inner node or a leave justified by an axiom or (b) it represents an open proof goal.

(a) Inner Node View is used for inner nodes of the proof tree. It highlights the formula which had been in focus at time of rule application as well as possible necessary side formulas. The applied rule is listed on the bottom of the view.

(b) Goal View is used when an open goal is selected. This view shows the Hoare triple to be proven and allows the user to apply rules. Moving the mouse cursor over the expressions within the node highlights the smallest enclosing term or formula at the current position. A left click creates a popup window showing all applicable rules for the currently highlighted formula or term.

A.6   Example

The following paragraphs describe in detail how to prove the Countdown example. We assume the problem has been loaded and no other interactions were performed. The maximal allowed number of automatic rule applications in the proof search strategy tab should be high enough. Setting it to 5000 is amply sufficient for this and most other examples.

The right pane displays the Hoare triple as specified in the input file. The first statement in the program section of the triple is a `while` statement. Thus the loop invariant rule is the first one to be applied. We move the mouse cursor on top of

the update such that update, program and postcondition all are highlighted. A left click lists all applicable rules. We select rule `Loop Invariant` which asks us subsequently for the loop invariant to use. In the dialogue box that pops up we are asked to enter a loop invariant, in the example `timer >= 0` is suitable.

After entering the loop invariant and confirming our choice by pressing `OK`, the proof view (left pane) shows the effect of the rule application: the proof has been split up into three branches labelled `Invariant Initially Valid`, `Preserves Invariant` and `Use Case`, respectively. By default, KeY selects the first proof goal `Invariant Initially Valid`. As mentioned in Sect. 2 this goal is purely first-order which is why it is preceded by a turnstile `|-`. First-order reasoning is handled as a black box and we do not manually apply any more rules, but start the automatic proof search strategy. We move the mouse on the turnstile such that the complete sequent is highlighted. A left click and selecting `Apply rules automatically here` closes this branch without further interaction. [7]

The `Preserves Invariant` case of the proof branch comes next. The shown Hoare triple formalises exactly that if the loop invariant and condition hold before executing the loop body then after its execution the invariant must hold again. Note that any context information has been removed: the update in front of the program is empty and the pre- and postconditions consist only of the loop guard and the supplied invariant. It is instructive to study this behaviour in detail by comparing the current proof node with the previous one where the invariant rule was applied. To this end, we navigate upwards in the proof tree and select the closest node above marked with ✍. This icon is used to mark all nodes where the user applied a rule interactively.

Back to the open goal: to prove the preservation property the assignment rule must be applied. We move the mouse to a position such that update, program and postcondition are highlighted and apply the `assignment` rule. After application of this rule only the empty program remains which is removed by applying the `exit` rule. This leaves us again with a pure first-order problem that is proven automatically as described for the `Invariant Initially Valid` branch above.

To close the final open goal it remains to be shown that starting in a state where the loop invariant holds, but not the loop condition, the execution of the remaining program leads to a state where the postcondition of the initial proof obligation holds, provided that the program terminates at all. In the `countdown` example the remaining program is empty. Applying the `exit` rule results in a pure first-order problem that can be solved by invoking the automatic strategies. This finishes the proof.

A.7 Automation

A few remarks on automation. Up to now the required interactive steps consisted of manual application of program rules and invocations of the strategies to simplify/prove pure first-order problems. In order to avoid to start the strategies manually one can activate the `autoresume` mode. This will invoke the strategies on all open

---

[7] In case the maximal number of allowed rule applications was set too low, one can simply restart the strategy as described before.

goals after each manual rule application and simplify them as far as possible. In standard mode they will not apply program rules.

While performing a proof it is possible to save the current state at any time and to load it afterwards. For this one has to select `File | Save` in the file menu and enter a file name ending with `.proof`.

A.8 Total Correctness

In the previous sections we concentrated on partial correctness proofs. Often one is also interested to ensure that a program terminates. Partial correctness plus termination is called *total correctness* and it is supported by KeY-Hoare. In addition, we provide a calculus to reason about simple worst-case execution time (WCET) properties [12]. This section introduces both calculi in brief.

To specify a total correctness problem, the only necessary change to an input file is the problem section tag `\hoareTotal` instead of `\hoare` for partial correctness (and `\hoareET` for WCET reasoning).

The calculus rules for total correctness are identical to those presented in Sect. 3, Fig. 2 except for the loop invariant rule. The new version of the loop invariant rule is given in Fig. A.4. In order to ensure that a while loop terminates one has to provide a term *dec* which decreases strictly monotonic after each execution of the loop body, but stays non-negative. The first branch of the while rule ensures that the given term is initially greater or equal to zero. Whereas the second branch checks that after each loop iteration *dec* is strictly smaller than before, but still non-negative. In order to access the old value of *dec* it introduces an unused rigid function of the *dec*s old value.

$$\vdash\ P \mathrel{-\!\!>} \mathcal{U}(I\ \&\ \mathrm{dec} >= 0)$$

$$\{I\ \&\ \mathtt{b} \doteq \mathtt{true}\ \&\ \mathrm{oldDec} \doteq \mathrm{dec}\}\,[]\,\mathtt{s1}\,\{I\ \&\ \mathrm{dec} >= 0\ \&\ \mathrm{dec} < \mathrm{oldDec}\}$$

$$\{I\ \&\ \mathtt{b} \doteq \mathtt{false}\}\,[]\,\mathtt{s}\,\{Q\}$$

$$\mathsf{loop_T}\ \frac{}{\{P\}\,[\mathcal{U}]\,\mathtt{while}\,(\mathtt{b})\,\{\mathtt{s1}\}\,\mathtt{s}\,\{Q\}}$$

where *old* is a new function symbol of arity $size(fv(\mathrm{dec}))$ ($fv(\mathrm{dec})$ denotes the set of free logical variables in dec)

Fig. A.4. Loop invariant rule for total correctness

The total correctness proof for the countdown example of Sect. A.6 is similar to the partial one, except that the loop invariant rule now requires to enter term dec. Choosing simply `timer` as decreasing term allows to close the proof following the steps described for the partial correctness proof.

The calculus to reason about WCET requires more changes. The complete calculus is presented in Fig. A.5.

The principle idea of the calculus is taken from [14]. When symbolically executing a statement an implicit counter is increased by one. In addition, the branching (guard evaluation) for conditional and loop statements costs also one time unit.

The countdown example serves once more to illustrate this approach. The input file enriched with an WCET specification is given in Fig. A.6.

17

$$\text{assignment}_{\mathsf{ET}} \ \frac{\{P\}\,[\mathcal{U},\ \mathtt{x} := \mathtt{e}, eT := eT + 1]\,\mathtt{s}\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{x=e;s}\,\{Q\}}$$

$$\text{skip}_{\mathsf{ET}} \ \frac{\{P\}\,[\mathcal{U}, eT := eT + 1]\,\mathtt{s}\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{;s}\,\{Q\}} \qquad\qquad \text{exit}_{\mathsf{ET}} \ \frac{\vdash\ P \rightarrow \mathcal{U}(Q)}{\{P\}\,[\mathcal{U}]\,\{Q\}}$$

$$\text{conditional}_{\mathsf{ET}} \ \frac{\begin{array}{c}\{P \ \&\ \mathcal{U}(\mathtt{b} \doteq \mathtt{true})\}\,[\mathcal{U}, eT := eT + 1]\,\mathtt{s1;s}\,\{Q\} \\[4pt] \{P \ \&\ \mathcal{U}(\mathtt{b} \doteq \mathtt{false})\}\,[\mathcal{U}, eT := eT + 1]\,\mathtt{s2;s}\,\{Q\}\end{array}}{\{P\}\,[\mathcal{U}]\,\mathtt{if(b)\{s1\}else\{s2\}s}\,\{Q\}}$$

$$\text{loop}_{\mathsf{ET}} \ \frac{\begin{array}{c}\vdash\ P \rightarrow \mathcal{U}(I \ \&\ \mathrm{dec} >= 0) \\[4pt] \{I \ \&\ \mathtt{b} \doteq \mathtt{true} \ \&\ \mathrm{oldDec} \doteq \mathrm{dec}\} \\ \qquad [eT := eT + 1]\{\mathtt{s1}\}\{I \ \&\ \mathrm{dec} >= 0 \ \&\ \mathrm{dec} < \mathrm{oldDec}\} \\[4pt] \{I \ \&\ \mathtt{b} \doteq \mathtt{false}\}\,[eT := eT + 1]\,\mathtt{s}\,\{Q\}\end{array}}{\{P\}\,[\mathcal{U}]\,\mathtt{while(b)\{s1\}s}\,\{Q\}}$$

where

- old is a new function symbol of arity $size(fv(\mathrm{dec}))$ ($fv(\mathrm{dec})$ denotes the set of free logical variables in dec)

- eT stands for the special program variable `executionTime` which does not occur elsewhere

Fig. A.5. Loop invariant rule for execution time reasoning

The functional part of the specification is left unchanged from the correctness specification. The execution-time related part of the requirements are simply added as a conjunction. In the precondition one has to state which value the execution time counter shall have initially. Usually, one requires that the value of the `executionTime` counter is initially either equal to a fixed non-negative, but unknown value or, as in the example, simply zero.

The postcondition typically specifies the exact number of execution steps (as done here) or an upper bound of the expected execution time. The countdown algorithm is expected to require $2 * \mathtt{startVal} + 1$ time units until completion. This number stems from `startVal` loop iterations where each iteration costs 2 time units (loop condition evaluation and decreasing the `timer` variable) and the final loop condition evaluation which evaluates to false.

The proof itself is similar to the previous countdown proofs. Only the invariant needs to be refined to

```
timer>=0 & executionTime = 2*(startVal-timer)
```

The decrease term can be chosen as before (i.e., simply: `timer`). Afterwards the proof can then be closed in the same way as before.

Note 1 Limitation of the current WCET implementation: *It is not possible to*

─── KeY ──

```
\functions { int startVal; }
\programVariables { int timer; }

\hoareET {

{ startVal >= 0 & executionTime = 0}

[timer := startVal]

\[{
     while (timer>0) {
          timer = timer -1;
     }
}\]

{ timer = 0 & executionTime = 2*startVal + 1}

}
```
─── KeY ───

Fig. A.6. Example countdown with additional execution time specifications.

specify that execution time grows with, for example, a linear factor depending on the program input. This would require to use the concept of meta variables used in the full version of the KeY tool [3]. Automated proof search with meta variables and their theoretical treatment belongs to the advanced concepts of KeY. Apart from that, the implementation symbolic WCET estimates is straightforward.