A Hoare-Style Calculus with Explicit State Updates

Reiner Hähnle and Richard Bubel

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 by forward symbolic execution. In addition, 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

1 Introduction

An introduction to formal program verification is part of many courses and textbooks about Formal Methods (for example, [23,14]). 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.
- 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.

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

---

1 Email: \{reiner,bubel\}@chalmers.se
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, 17, 16, 22, 2, 4, 3], 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 clear 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. While the calculus is based on weakest precondition, the chosen presentation form are Hoare triples as in our experience the latter one seems to be used more frequently in education. 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. Our implementation is based on the KeY tool [3], one of the most powerful verification systems for Java.

2 Background

2.1 Target Programming Language

We use a simple imperative while programming language with only four kinds of statements:

\[
\begin{align*}
\text{Program} & ::= (\text{Statement})^? \\
\text{Statement} & ::= \text{AssignmentStatement} \mid \text{CompoundStatement} \mid \text{ConditionalStatement} \mid \text{LoopStatement} \\
\text{AssignmentStatement} & ::= \text{Location} = \text{Expression}'; \\
\text{CompoundStatement} & ::= \text{Statement}; \\
\text{ConditionalStatement} & ::= \text{if } ('\text{BooleanExp}') \{ '\text{Statement}' \} \{ '\text{Statement}' \} \text{ else } '\text{Statement}' \\
\text{LoopStatement} & ::= \text{while } ('\text{BooleanExp}') \{ '\text{Statement}' \} \\
\text{Expression} & ::= \text{BooleanExp} \mid \text{IntExp} \\
\text{BooleanExp} & ::= \text{IntExp} \text{ ComparisonOp} \text{ IntExp} \mid \text{IntExp} = \text{IntExp} \mid \text{IntExp} \text{ BooleanOp} \text{ BooleanExp} \mid \text{IntExp} \text{ IntOp} \text{ IntExp} \mid \text{true} \mid \text{false} \\
\text{IntExp} & ::= \text{IntExp} \text{ IntOp} \text{ IntExp} \mid \text{Z} \mid \text{Location} \\
\text{ComparisonOp} & ::= < \mid <= \mid >= \mid > \\
\text{BooleanOp} & ::= \& \mid \mid \mid \text{==} \\
\text{IntOp} & ::= \* \mid / \mid \% \mid + \mid -
\end{align*}
\]

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. Locations and expressions are typed. There are two incomparable types called boolean and int. The type int denotes the mathematical integers Z, not a finite integer.

\footnote{From \url{http://www.key-project.org/download/hoare/}, including all examples discussed in this article.}

\footnote{The definitions given here are unsound in the presence of aliasing. General definitions of the concepts involved are found in [3].}

50
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 either have the form

• \( P(t_1, \ldots, t_n) \) with \( P \) standing for an arbitrary user-defined predicate symbol of arity \( n \) and terms \( t_1, \ldots, t_n \) of appropriate type or
• \( s \equiv t \) with the reserved equality symbol \( \equiv \) taking arbitrary terms as arguments.

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 addition, built-in symbols with a fixed semantics, such as equality \( \equiv \) and the operators occurring in expressions of the programming language, are rigid.

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.

We decided against modelling program variables as logical variables, mainly for usability reasons: in order to get by with logical variables alone one needs introduce primed variables as in [6]. Each state change during symbolic execution necessitates introduction of fresh primed variables. These increases the number of symbols required to specify and to prove a problem which in turn compromises readability of proofs. Readability, however, is an important issue when user interaction is required—not only for students on the beginner level. A further point in favour of the introduction of non-rigid symbols is to avoid confusion for students who may just have gotten used to the static viewpoint of first-order logic. In addition, the provided tool is built-up on an inference engine tailored to dynamic logic where non-rigid modelling is natural.

Some useful conventions: program variables are typeset in typewriter font, logical variables in italic. When we specify a program \( \pi \) we assume that all program variables of \( \pi \) 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 \{true, false\} and all integer terms over \( \mathbb{Z} \). There are built-in function symbols for arithmetic including +, -, \*, / and \%.
and integer comparison operators $\leq$, $<$, $>$ and $\geq$ with their obvious meaning. See the reference manual [11] for concrete formula syntax. 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:

$$\{P\} \pi \{Q\}$$

Here, $P$ and $Q$ are closed first-order formulas and $\pi$ 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 $\pi$ is started with initial values $i_k = \mathcal{M}(l_k)$ $(1 \leq k \leq m)$ and if $\pi$ terminates with final values $f_k$, then $\mathcal{M}_{l_1, \ldots, l_m}^{i_1, \ldots, i_m}$ is a model of $Q$.

We can paraphrase this in a slightly more informal, but more intuitive, manner: for a given program $\pi$ over locations $\{l_1, \ldots, l_m\}$, let us call an assignment of values $l_k = v_k$ $(1 \leq k \leq m)$ the state $s$ of $\pi$. What the Hoare triple then says is that if we start $\pi$ in any state satisfying the precondition $P$, if $\pi$ 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, $s, s_1, s_2$ are statements. $P, Q, R, I$ are closed first-order formulas.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Hoare Triple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>${P} x = e; {P}$</td>
</tr>
<tr>
<td>Composition</td>
<td>${P} s_1 {R} \quad {R} s_2 {Q}$</td>
</tr>
<tr>
<td>Conditional</td>
<td>${P &amp; b \doteq \text{true}} s_1 {Q} \quad {P &amp; b \doteq \text{false}} s_2 {Q}$</td>
</tr>
<tr>
<td>Loop</td>
<td>${I} \text{while}(b){s} {I &amp; b \doteq \text{false}}$</td>
</tr>
<tr>
<td>Weakening Left</td>
<td>$P \Rightarrow Q \quad {Q} s {R}$</td>
</tr>
<tr>
<td>Weakening Right</td>
<td>${P} s {R} \quad Q \Rightarrow R \quad {P} s {R}$</td>
</tr>
<tr>
<td>Oracle</td>
<td>$\mathcal{P}$ (any valid first-order formula)</td>
</tr>
</tbody>
</table>

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 \( U \) and \( V \) are updates, then so are \( U, V \) (sequential update), and \( U \| V \) (parallel update).\(^4\)

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

\[
s_U(l) = \begin{cases} 
  s(l) & \text{if } l \notin \{l_1, \ldots, l_m\} \\
  t_k & \text{if } l = l_k \text{ and } l \notin \{l_{k+1}, \ldots, l_m\}
\end{cases}
\]

(2)

In words: the value of the locations occurring in \( U \) 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 \( U \) “wins” if the same location occurs more than once on the left-hand side in \( U \). Apart from that, all updates are executed in parallel. Updates are similar to a preamble or fixture as used in unit testing \([18]\): 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 stated in Lemma 3.1 below. Let us call two updates \( U \) and \( V \) equivalent if \( s_U = s_V \) for any state \( s \). Then for each update \( U \) exists an equivalent parallel update \( V \) of the form \( l_1 := t_1 \| \cdots \| l_m := t_m \).

3.2 Hoare Triples with Update

We allow to write an update \( U \) in front of any program like this: \([U] \pi\). If we are in state \( s \) the meaning is that the program is started in state \( s_U \). Within Hoare logic

\(^4\) There are further kinds of updates \([20,3]\), but we do not need these here.
we use updates as follows:

$$\{P\} [U] \pi \{Q\}$$

(3)

where, $P$, $Q$, and $\pi$ are as above, and $U$ is an update over the signature of $P$ and $\pi$. We enclose updates in square brackets to increase readability. Either one of $U$ and $\pi$ can be empty. The meaning of this Hoare triple with update is as follows: if $s$ is any state satisfying the precondition $P$ and we start $\pi$ in $s_U$, then, if $\pi$ 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. Thereby a precise formula $R$ is computed which is sufficient to achieve a complete calculus, but it does not subsume the composition rule as a whole as it lacks its implicit weakening.

• 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 $U$ 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 $U$ with respect to the postcondition $Q$. 54
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 $U$. 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.

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 $U$. 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 $U$. 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 (Sect 3.1) and (ii) apply updates to terms, formulas, and other updates. For the first task we use a Lemma from [19] (in specialised form):

**Lemma 3.1** For any updates $U$ and $x := t$ the updates $U$, $x := t$ and $U \parallel x := U(t)$ 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, $P$ is a parallel update of the form $t_1 \parallel \cdots \parallel t_m$, $y$ is a logical variable, $F$ is an $n$-ary function or predicate symbol, $\Box$ is a propositional connective, and $\lambda$ 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. 53. 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 occur in updates one has to ensure that no name clashes occur (free(U) returns the set of logical variables not bound in 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 program computing the greatest common divisor.

\begin{verbatim}
while (!small == 0) {
    tmp = big % small;
    big = small;
    small = tmp;
}
result = big;
\end{verbatim}

All variables are integers. Provided that big is greater or equal than small and both are non-negative, result contains the greatest common divisor of big and small. Let _big and _small be rigid constants that capture arbitrary initial values of big and small. A suitable precondition \( P \) is: \( _\text{small} >= 0 \ & \ _\text{big} >= _\text{small} \). Let \( R \) be the following formula which expresses that any common divisor \( x \) of the integers _big and _small is a divisor of the integer result.

\begin{verbatim}
\forall all int x; 
    ((x > 0 & _\text{big} % x = 0 & _\text{small} % x = 0) \rightarrow result % x = 0))
\end{verbatim}

Then the postcondition \( Q \) can be stated as:
\[ (_\text{big} \neq 0 \to R) \land (_\text{big} = 0 \to \text{result} = 0) \]
(concrete formula syntax see [11]). With the abbreviations introduced above, the initial Hoare triple with updates reads as follows:

\[
\{ P \} \text{big} := \_\text{big} \parallel \text{small} := \_\text{small} | \text{gcd}\{Q\}
\]

A file with an initial Hoare triple as proof obligation (in a simple format described in [11]) 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 on the left). 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. Here the idea for the invariant is that the loop leaves all common divisors of big and small invariant, hence, big and small have exactly the same common divisors than _big and _small. One needs also to state that big never gets smaller than small and what happens when _big==0:

\[
\text{small} \geq 0 \land \text{big} \geq \text{small} \land (\text{big} = 0 \iff _\text{big} = 0) \land \\
\forall \text{int \ x; } (x > 0 \to ( (_\text{big} \% x = 0 \land _\text{small} \% x = 0) \iff (\text{big} \% x = 0 \land \text{small} \% x = 0)))
\]

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 gcd, 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 ed-
ucational tool for Hoare Calculus is J-Algo [15], 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. The RISC Navigator [21] provides an interactive proof assistant with an interface to external decision procedures. A main goal is to provide an easy-to-use tool suitable for educational purposes. It is used in teaching program verification, but requires to generate verification conditions of a Hoare triple by hand. Afterwards these conditions can be loaded and proven within the system. Our state updates are closely related to generalised substitutions used by the B method [1] and to Abstract State Machines [5]. A full discussion is contained in [20]. There are versions of Hoare logic that use the assignment rule from dynamic logic [12] 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. It is based on a state-of-art verification system for Java [3]. The KeY-Hoare tool is currently used in the course Program Verification intended for Bachelors in their final year at Chalmers University. 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

We thank Wolfgang Ahrendt and Philipp Rümmer for numerous helpful comments. We thank also the anonymous referee for valuable comments and suggestions.

References


