Integration of a Software Model Checker into Isabelle*

Matthias Daum[1], Stefan Maus[2], Norbert Schirmer[3], and M. Nassim Seghir[2]

[1] Universität des Saarlandes, Saarbrücken, Germany
[2] Max-Planck Institut für Informatik, Saarbrücken
[3] Technische Universität München, Germany

Abstract. The paper presents a combination of interactive and automatic methods in the area of software verification. We have integrated a newly developed software model checker into an interactive verification environment for imperative programming languages. The verification environment is implemented on top of the general-purpose theorem prover Isabelle as an instance of the well-known Hoare calculus. With our work, we make inroads into the field of software verification with automatic tools. Though the problems are considered too hard for full automation, we increase the level of automated assistance by discharging less interesting side conditions, allowing the verification engineer to focus on the major proof goals.

1 Introduction

Our work is part of the Verisoft project.[4] This large coordinated project aims at the pervasive formal verification of entire computer systems consisting of hardware, compiler, operating system, communication system, and applications. To the best of our knowledge, the last attempt to cope with such an ambitious topic has been the famous CLI stack [1] back in 1989—even though the principal researcher of the famous CLI stack project, J. S. Moore, declared in [7] that the formal verification of a system 'from transistor to software level' is a grand challenge problem. However, basic research in the area of formal verification has greatly evolved during the last 15 years. A major goal of the Verisoft project is to solve that challenge by integrating and improving the existing technology.

As in the CLI stack project, we have several layers of abstraction. However, we plan to employ the verification environment described in this paper for almost all our software components, starting from considerable parts of the micro kernel and the operating system up to the application level.

The rest of the paper is organized as follows. Section 2 presents related work in this area. Section 3 introduces Isabelle and the Hoare Logic module. Furthermore it gives an idea of the checked runtime errors and illustrates our verification

* Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the Verisoft project under grant 01 IS C38.

[4] More information on the Verisoft project can be found at `http://www.verisoft.de/`

environment by a small example. In Section 4, we present our newly developed software model checker for reachability analysis in C programs. Section 5 reports about the integration of this model checker into our verification environment. We describe certain aspects of the novel *swmc_guards* tactic, which implements the user interface to our model checker. Furthermore we discuss some enhancements of the model checker to check the reachability of multiple locations simultaneously. Our first practical experiences are summarized in Section 6. Finally, Section 7 concludes the paper.

2 Related work

Several works for combining verification techniques have been proposed in the literature, including different ways of integrating automatic approaches into interactive theorem proving. Pisini *et al* [3] integrated the MDG tool, that supports model checking and equivalence checking, into the HOL system, a theorem prover for higher order logic, for the verification of hardware. They introduced two tactics, MDG_COMB_TAC and MDG_SEQ_TAC to generate the adequate input files for the MDG system to complete the proof.

As well in the context of hardware verification, Joyce and Seger [6] proposed a link between symbolic trajectory evaluation and interactive theorem proving. Their technique consists in introducing a new proof procedure, called VOSS_TAC, through which the Voss system is invoked for checking the validity of assertions, and returning the result to the theorem prover.

Rajan *et al* [9] described an approach where a BDD-based model checker for the propositional mu-calculus was used as a decision procedure within the framework of the PVS proof checker.

Our integration approach is quite similar to the one proposed in [3]. However, while all systems mentioned above aim at hardware verification, our integration approach concerns software verification. Software is more complex than hardware in the sense that it includes more language constructs and a larger variety of data types.

3 Verification Environment

Isabelle is a generic proof assistant. It provides a framework to declare deductive systems, rather than to implement them from scratch. Currently the best developed object logic is HOL [8], higher order logic, including an extensive library of (concrete) mathematics, as well as various packages for advanced definitional concepts like (co-)inductive sets and types, primitive and well-founded recursion etc. To define an object logic, the user has to declare the syntax and the inference rules of the object logic. By employing the built-in mechanisms of Isabelle/Pure, in particular higher-order unification and resolution, one already gets a decent deductive system. Moreover, Isabelle follows the well-known *LCF system approach*, which allows to write arbitrary proof procedures in ML without breaking system soundness since all those procedures are expanded into primitive inferences of the logical kernel. To integrate trusted external programs, the mechanism of *oracles* can be employed. An oracle produces a theorem without breaking its proof down to primitive inferences. The software model checker is integrated as such an oracle into a Hoare Logic module of Isabelle/HOL.

The Hoare Logic module described in [10] is built on top of Isabelle/HOL. An imperative programming language is defined as HOL data-type together with an operational semantics and a Hoare calculus for partial and total correctness. Programs are specified as Hoare triples and verified using a verification condition generator. A Hoare triple has the format $\Gamma \vdash\ P\ c\ Q$ where Γ is the procedure environment that maps procedure names to their bodies, c is a code fragment and P and Q are assertions. Intuitively, the formula states that if P holds before the execution of c then Q will hold afterwards. In this paper we only focus on partial correctness. Runtime faults are modeled as explicit guards within the program c. The semantic of the Hoare Logic ensures that every such guard must hold under the precondition P. Formally assertions and guards are sets of program states. The states are represented as records in HOL. As example, the assertion $\{\!|\, ´i \le N |\!\}$ abbreviates the set comprehension $\{s.\ i\ s \le N\}$, where i is a record selector. The abstraction on state s is hidden in assertions, and the application to s is abbreviated by the prefixed acute $´$.

Many runtime errors can occur during the execution of a program due to the violation of some constraints imposed by the definition of the data types used in the program. Examples of errors are overflow and underflow exceptions and array out of bound access. The programming language used in the Verisoft project is called C0. It is a type-safe subset of C with an exact specified semantics, which is also formalized in Isabelle/HOL. Numeric expressions in C0 are evaluated using bounded modulo arithmetic with silent over- and underflows. But for specifying and reasoning about programs we want to "think unbounded". Therefor we regard over and underflows as runtime errors on the level of the Hoare Logic and use ordinary unbounded arithmetic.

To prove the absence of such runtime errors, we have to identify which expressions can potentially cause which errors. We have formalized the error conditions. Table 1 on the following page shows a non-exhaustive list of expressions that might cause runtime errors. For each of these expressions, the table lists a set of guards. The evaluation of an expression causes a runtime error if and only if the conjunction of its guards evaluates to false. The guards are automatically generated by Isabelle through the parsing process of the program code.

Figure 1 on the next page illustrates the program representation in our verification environment. Shown is the proof goal for the correctness theorem of a bubble-sort implementation. The code fragment sorts the first *´array-size* values contained in an array variable named *´array*.

expression e	guards	runtime error
$e_1 + e_2$ $e_1 - e_2$ $- e_1$ $e_1 * e_2$	$e \leq (max_{type(e)})$ $e \geq (min_{type(e)})$	overflow underflow
e_1 / e_2	$e_2 \neq 0$ $e \leq (max_{type(e)})$ $e \geq (min_{type(e)})$	division by zero overflow underflow
$e_1 [e_2]$	$e_2 < size(e_1)$ $e_2 \geq 0$	above bounds of e_1 below bounds of e_1

type T	min_T	max_T
`int`	-2^{31}	$2^{31}-1$
`unsigned`	0	$2^{32}-1$
`char`	-2^7	2^7-1

Table 1. The table shows some expressions causing runtime errors together with their respective guards (top) and the ranges for the basic integer types (bottom).

```
⋀σ. Γ⊢ {|σ. 0 < ´array-size ∧ ´array-size ≤ length ´array|}
        {|1 ≤ ´array-size|}↦ ´i := ´array-size − 1;
        WHILE 0 < ´i
        DO ´j := 0;
            WHILE ´j < ´i
            DO {|´j + 1 ≤ max-nat ∧
                 ´j + 1 < length ´array ∧ ´j < length ´array|}
               ↦ IF ´array[´j + 1] < ´array[´j]
                  THEN {|´j < length ´array|}↦ ´temp := ´array[´j];
                    {|´j < length ´array ∧
                      ´j + 1 ≤ max-nat ∧ ´j + 1 < length ´array|}
                    ↦ ´array[´j] := ´array[´j + 1];
                    {|´j + 1 ≤ max-nat ∧ ´j + 1 < length ´array|}
                    ↦ ´array[´j + 1] := ´temp
                  FI;
               {|´j + 1 ≤ max-nat|}↦ ´j := ´j + 1
            OD;
            {|1 ≤ ´i|}↦ ´i := ´i − 1
        OD;
        ´res := 0
        {|∀j < σarray-size. ∀i < j. ´array[i] ≤ ´array[j]|}
```

Fig. 1. External representation of code with guarded commands within our verification environment. A guarded command consists of a list of guard conditions and the affecting command. The conditions are enclosed in braces: $\{\!|\ |\!\}$. Guard conditions and command are separated by $\mapsto$. The term $^{\sigma}$*array-size* refers to the old value of *array-size* before the execution of the program fragment.

4 The Model Checker ACSAR

ACSAR (Automatic Checker of Safety properties based on Abstraction Refinement) is a software model checker for C programs that we developed in the same spirit of Magic[5] [2] and Blast[6]. Except for recursive functions and expressions, which cause side effects, ACSAR can handle most constructs of the C language. Function calls are handled by inlining the body of each function into the corresponding call site. Local variables are renamed to avoid name conflicts. Thus, after inlining all the functions, we obtain a unique global control-flow graph. The obtained control flow graph contains only two types of nodes: branches and updates. The verification algorithm of ACSAR is based on the following key notions:

4.1 Translating programs to transition constraints

A transition constraint tc is a tuple (l, g, u, d) where l and d are values of the program counter before respectively after performing the transition, g is the transition condition and u is the variable update. Consecutive assignments are considered as one simultaneous update. We illustrate the translation procedure considering the function *three_times* as example:

Example 1.

```
void three_times(int n)
{
  int s = 0, i=0, result;
  while (i != n)
  {
   s = s + 3;
   i = i + 1;
  }
  result = s;
}
```

The logical representation of this function in terms of the transition constraints would look like this:

$$\begin{array}{l} pc = 1 \wedge s' = 0 \wedge i' = 0 \wedge pc' = 4 \\ pc = 4 \wedge i \neq n \wedge i' = i + 1 \wedge s' = s + 3 \wedge pc' = 4 \\ pc = 4 \wedge i = n \wedge res' = s \wedge pc' = 10 \end{array}$$

where the variable pc represents the program counter and variables with a prime represent new values of the original variables after an update.

[5] See also the Magic Home page for more information: `http://www-2.cs.cmu.edu/~chaki/magic/`

[6] More information on Blast can be found at `http://www-cad.eecs.berkeley.edu/~rupak/blast/`

4.2 Abstraction

ACSAR uses the predicate abstraction technique [4] to automatically abstract an infinite system by a finite one. The idea of predicate abstraction is to represent a set of states by a logical formula built from a set of predicates. This logical formula represents an abstract state. Unlike Blast which implements a forward algorithm, ACSAR uses a backward search. Formally, we introduce:

- the set of program states S,
- the set of error states S_{err},
- the set of predicates P (initially empty).
- the set of transition constraints Tc.
- the abstraction function $\alpha : 2^S \rightarrow 2^P$ with $\alpha(s) = \bigcup\{p\}$ such that $(p \in P \wedge \forall x \in s.\ p$ is true in $x)$. α maps a set of state to a set of predicates. Intuitively, the set of predicates represent an abstract state obtained by the conjunction of the predicates contained in the set.
- the operator $Pre^{\#}(s, tc) = \alpha(wp(s, tc))$, that returns the previous abstract state where s is an abstract state provided as a logical formula, $tc \in Tc$ is some transition constraint, and $wp(s, tc)$ is the exact weakest precondition of s with respect to tc. Intuitively, the $Pre^{\#}$ operator provides the necessary precondition to obtain s after performing the transition tc.

Now, we can build the the abstract system. We start with the abstract error state $\alpha(S_{err})$ and to try to compute the least fixpoint of $Pre^{\#}$. Either we find the least fixpoint or a counter example. If we have found a counter example, We have to test its validity. In case of a spurious counter example, our abstraction was too coarse, and we have to refine our abstraction.

4.3 Refinement

When an abstraction is too coarse (we found a spurious counter example), ACSAR rebuilds a more precise abstraction by inferring new predicates. It increases only the relevant part of the abstract system. This concept of *laziness* is inspired from the work of Henzinger *et al* [5]. If the backward search reaches the initial state S_I, the path leading from S_{err} to S_I is analyzed using the exact weakest precondition wp to check the validity of transitions that constitute the path. If the analysis indicates that this path is a real counter example, the path is returned as witness to the user. Otherwise, we obtain a formula showing the invalidity of the path. Predicates appearing in this formula are used to refine the system.

5 Integrating ACSAR into the Verification Environment

As shown in Table 1 on page 4, the guards are expressed as assertions in quantifier-free first-order logic. Software model checkers deal efficiently with such properties. In order to take advantage of the efficiency and automation of model checking, we integrated our tool ACSAR into Isabelle.

In Figure 1 on page 4, we have already seen the external representation of a bubble-sort implementation. At that point, we would like to discharge the guards automatically with ACSAR. For the integration of our software model checker, we have implemented a new tactic called *swmc_guards*. This tactic plays the role of a user interface.

The current proof goal is taken from the theorem prover, translated into a C file, and presented as reachability problem to ACSAR. The model checker generates a reachability check report specifying for each error location whether it is *reachable*, *unreachable* or ACSAR could not decide on reachability. Finally, the reachability report from the model checker is converted into a new proof goal and feeded back into Isabelle. In the next sections, we describe the translation processes in detail.

5.1 Translating a guard condition check into a reachability problem

The semantic of guards expresses that a guard should always hold to guarantee the correct execution of the guarded command. If a guard is not valid, an unexpected result may occur. Correctness of the program implies that a guard violation should never happen, which expresses a *safety* property. Checking safety properties can be reduced to reachability check problems.

Suppose that the system is provided in terms of transition constraints as introduced in Section 4. Now we can transform the validity check of a guard condition g at location l into a reachability problem as follows: We introduce an error location r and a transition constraint $tc = (l, \neg g, -, r)$ into our system. If the newly introduced error location is distinct from all other locations, and consequently, no other transition constraint than tc is leading to r, we can state:

$$(r \text{ is reachable}) \longleftrightarrow (\neg g \text{ can hold})$$

A graph search has to be performed to check the reachability of the error state.

5.2 Checking Reachability of Multiple Error Locations

Initially, ACSAR was designed to check the reachability of only one error location at a time. To deal with multiple error locations, one has the choice between two options. The first option is to invoke ACSAR several times from Isabelle. This approach is simple in the sense that no major changes are needed. Its drawback is the time consumed by communications between the theorem prover and the model checker. The second option, that we adopted, is to extend the checking algorithm of ACSAR to deal with multiple error locations. All guards are transmitted at once to the model checker rather than transmitting one guard at a time.

Therefore we have to extend the translation algorithm above. We assume a finite set G of guards and a finite set L of control locations. Furthermore there should be a control location $l_i \in L$ associated to each guard $g_i \in G$. Now we introduce a new error location r_i for each guard such that the set of error locations R will be distinct from the control locations and from each other, i. e.

$$L \cap R = \emptyset \quad \wedge \quad \forall g_i, g_j.\ r_i = r_j \longrightarrow g_i = g_j$$

Finally, we have to introduce a transition constraint $tc_i = (l_i, \neg g_i, -, r_i)$ for each guard $g_i \in G$, and can state:

$$\forall g \in G.\ \exists! r \in R \text{ such that } (r \text{ is reachable}) \longleftrightarrow (\neg g \text{ can hold})$$

Figure 2 on the facing page illustrates the resulting C code of the previous bubble-sort example after the translation into a reachability problem and adding the necessary error locations.

Verification approach. Let us consider a program in terms of a set of transition constraints Tc. The subset Tc_{err} contains the transition constraints tc_i that lead to error locations. For each transition constraint $tc_i = (l_i, \neg g_i, -, r_i) \in Tc_{err}$, there exists a transition constraint $tc'_i = (l_i, g_i, u_i, k_i) \in Tc$ corresponding to the case that guard g_i holds. Now we build a transition constraint tc''_i from tc'_i by removing the guard g_i and keeping all other fields unchanged: $tc''_i = (l_i, \mathit{True}, u_i, k_i)$.

In the verification phase, we remove for all guards g_j apart from the currently processed guard g_i ($j \neq i$), the transition constraints tc_j and tc'_j, and add tc''_j to the set of transition constraints Tc. That means, all other guards are disabled, which is important to avoid influence between guards. Consider the following example:

Example 2.

```
   int a [6], b [4], i ,j ,k;
   i = 5;
   j = i − 1;
4  if (j+2 > 5) goto ERROR_1;
   a[j+2] = 3;
   k = j + 1;
   if (k > 4) goto ERROR_2;
8  b[k] = 1;
   goto end;
   ERROR_1:   goto end;
   ERROR_2:   goto end;
12 end:       ;
```

If the guard leading to label ERROR_1 is not disabled when checking the next guard, we can not find out whether label ERROR_2 is reachable because the expression (j+2 > 5) at line 4 is always true.

```
int array [10];
unsigned int i, j,  array_size ;
int temp, res;

int main () {
  if (((0 <  array_size ) && (array_size <= 10))) {
    if (!(1 <= array_size )) goto ERROR_1;
    i = (array_size  − 1);
    while (0 < i) {
      j = 0;
      while (j < i) {
        if  (!(( j + 1) <= max_nat)) goto ERROR_2;
        if  (!(( j + 1) < 10)) goto ERROR_3;
        if  (!( j < 10)) goto ERROR_4;
        if  ((array[( j + 1)] < array[ j ])) {
          if  (!( j < 10)) goto ERROR_5;
          temp = array[j];
          if  (!( j < 10)) goto ERROR_6;
          if  (!(( j + 1) <= max_nat)) goto ERROR_7;
          if  (!(( j + 1) < 10)) goto ERROR_8;
          array[ j ] = array[(j  + 1)];
          if  (!(( j + 1) <= max_nat)) goto ERROR_9;
          if  (!(( j + 1) < 10)) goto ERROR_10;
          array[( j + 1)] = temp;
        }
        if  (!(( j + 1) <= max_nat)) goto ERROR_11;
        j = (j  + 1);
      }
      if (!(1 <= i )) goto ERROR_12;
      i = (i  − 1);
    }
    res = 0;
  }
  goto end;
ERROR_12:     goto end;
ERROR_11:     goto end;
ERROR_10:     goto end;
    /* ... */
ERROR_1:      goto end;
end:          ;
}
```

Fig. 2. Result of the translation (input to ACSAR). Note: For better readability, we have replaced the numerical upper limit of unsigneds with *max_nat*.

Introducing Assumptions. In the previously described approach, each guard is checked in isolation from the other guards. This is equivalent to have each time a program P_i containing only the guard g_i that we want to prove. This approach can be improved by exploiting results for previous guards in the verification of the actual guard. When a guard is proven to hold, we use it as assumption for the verification of other guards. Formally, we remove tc'_i for each proven guard g_i and keep tc_i as we know that the error location r_i is never reachable. This approach is described in Figure 3 on the next page.

Table 2 presents a performance comparison between the approach when we use valid guards as assumptions and the approach without assumptions. We have measured the execution times of the software model checker for the already presented *bubble-sort* example and an implementation of the more naive *selection-sort* algorithm. In both cases, the execution time is shorter if we keep proven guards as assumptions. The improvement obtained for the *bubble-sort* program is not so important, because the checking time without considering assumptions is already small. However, the time obtained for the *selection-sort* example is remarkably improved by approximately ten times, which shows the effectiveness of our optimization.

Program	number of guards	verification time(s)	
		without assumptions	with assumptions
selection-sort	14	49.381	4.659
bubble-sort	12	1.113	1.063

Table 2. Model checking time of programs with and without using assumptions.

5.3 Returning the results to Isabelle

Our interface is implemented as a so called *oracle*. This way we can introduce a theorem into the verification process without giving a proof for it. In our case, a Hoare triple with annotated guard conditions is returned as theorem. The annotation expresses whether guards will not fail. This Hoare triple is used as premise of an Hoare rule that allows to regard the proven guards as granted for the remaining verification.

In this section we discuss how this rule is formally justified within the Hoare calculus. Within our verification environment, validated guards are decorated with $\surd$ as shown in Figure 4 on page 12.

To integrate the notion of discharging a guard into the Hoare calculus, the guarded command of the programming-language model described in [10] is augmented with a flag: *Guard f g c*, where f is the kind of fault that the guarded command will raise if the guard condition g for command c fails. The syntax in the examples $g \mapsto c$ is an abbreviation for *Guard False g c*, and $g\surd \mapsto c$ for *Guard True g c*. A comma-separated list of guard conditions before the $\mapsto$

Input:

- the set Tc of all transition constraints
- the set Tc_{err} of transition constraints leading to error locations
- the set G of guards

Output:

- a report *rep* specifying for each guard whether it is *valid*, *invalid* or *unknown*.

begin
$Tc'_{err} = \bigcup\{tc'_i \text{ such that } tc_i \in Tc_{err}\}$;
$Tc''_{err} = \bigcup\{tc''_i \text{ such that } tc_i \in Tc_{err}\}$;
$Tc = (Tc - (Tc_{err} \cup Tc'_{err})) \cup Tc''_{err}$;
for each guard g_i such that $g_i \in Set\_G$ **do**
  $Tc = (Tc - \{tc''_i\}) \cup \{tc_i, tc'_i\}$;
  $res = check\_reach(err\_loc(g_i), Tc)$;
  if $res = unreachable$;
    $store\_in\_report(rep, g_i, valid)$;
    $Tc = Tc - tc'_i$;
  else
    if $res = reachable$;
      $store\_in\_report(rep, g_i, invalid)$;
      $Tc = (Tc - \{tc_i, tc'_i\}) \cup tc''_i$;
    else
      $store\_in\_report(rep, g_i, unknown)$;
      $Tc = (Tc - \{tc_i, tc'_i\}) \cup tc''_i$;
  $init()$;
return(rep);
end

Legend:

- The function *err_loc* takes a guard as argument and returns the corresponding error location.
- The function *check_reach* returns either (a) *unreachable* if the guard is valid, (b) *reachable* if the guard is invalid, or (c) *unknown* if no definite decision on the validity of the guard can be made.
- The function *init* reinitializes the system by erasing states generated so far.
- The function *store_in_report* stores the result of the verification concerning the guard in a report file. The report is returned to the theorem prover.

Fig. 3. Multiple error verification by exploiting assumptions.

$$
\begin{array}{l}
\bigwedge\sigma.\ \Gamma\vdash_{/\{True\}}\ \{\!|\sigma.\ 0 < \acute{}array\text{-}size \wedge \acute{}array\text{-}size \leq length\ \acute{}array|\!\}\\
\quad \{\!|1 \leq \acute{}array\text{-}size|\!\}\surd\mapsto\ \acute{}i := \acute{}array\text{-}size - 1;\\
\quad WHILE\ 0 < \acute{}i\\
\quad DO\ \acute{}j := 0;\\
\qquad WHILE\ \acute{}j < \acute{}i\\
\qquad DO\ \{\!|\acute{}j + 1 \leq max\text{-}nat|\!\}\surd,\ \{\!|\acute{}j + 1 < length\ \acute{}array|\!\},\\
\qquad\quad \{\!|\acute{}j < length\ \acute{}array|\!\}\surd\\
\qquad\quad \mapsto IF\ \acute{}array[\acute{}j + 1] < \acute{}array[\acute{}j]\\
\qquad\qquad THEN\ \{\!|\acute{}j < length\ \acute{}array|\!\}\surd\mapsto\ \acute{}temp := \acute{}array[\acute{}j];\\
\qquad\qquad\ \{\!|\acute{}j < length\ \acute{}array|\!\}\surd,\ \{\!|\acute{}j + 1 \leq max\text{-}nat|\!\}\surd,\\
\qquad\qquad\ \{\!|\acute{}j + 1 < length\ \acute{}array|\!\}\surd\\
\qquad\qquad\ \mapsto\ \acute{}array[\acute{}j] := \acute{}array[\acute{}j + 1];\\
\qquad\qquad\ \{\!|\acute{}j + 1 \leq max\text{-}nat|\!\}\surd,\ \{\!|\acute{}j + 1 < length\ \acute{}array|\!\}\surd\\
\qquad\qquad\ \mapsto\ \acute{}array[\acute{}j + 1] := \acute{}temp\\
\qquad\qquad FI;\\
\qquad\quad \{\!|\acute{}j + 1 \leq max\text{-}nat|\!\}\surd\mapsto\ \acute{}j := \acute{}j + 1\\
\qquad OD;\\
\qquad \{\!|1 \leq \acute{}i|\!\}\surd\mapsto\ \acute{}i := \acute{}i - 1\\
\quad OD;\\
\quad \acute{}res := 0\\
\quad \{\!|\forall j < {}^{\sigma}array\text{-}size.\ \forall i < j.\ \acute{}array[i] \leq \acute{}array[j]|\!\}
\end{array}
$$

Fig. 4. The bubble-sort example after calling the software model checker. All validated guard conditions are decorated with $\surd$.

abbreviates nested guarded commands. The state of the programming language is a polymorphic HOL data-type with two constructors:

$$\textbf{datatype}\ ('f,'s)\ state = Normal\ 's \mid Fault\ 'f$$

where 's is a type variable for the raw state and 'f for the faults. The operational big-step semantics has the format $\Gamma\vdash \langle c,\ s\rangle \Rightarrow t$, where Γ is the procedure environment that maps procedure names to their bodies. The meaning is, that execution of command c in the initial state s ends in final state t. The big-step semantic rules for guarded commands are the following:

$$\frac{s \in g \qquad \Gamma\vdash \langle c,\ Normal\ s\rangle \Rightarrow t}{\Gamma\vdash \langle Guard\ f\ g\ c,\ Normal\ s\rangle \Rightarrow t} \qquad \frac{s \notin g}{\Gamma\vdash \langle Guard\ f\ g\ c,\ Normal\ s\rangle \Rightarrow Fault\ f}$$

The guard condition g is modeled as a set of states. If it holds, execution is continued otherwise the error is signaled. For the integration of the software model checker, the flag f is Boolean. The flag *True* indicates that the guard is proven. But as the semantic rules above indicate, the value of the flag is not considered to decide whether the guard holds or not. It is only used in the Hoare calculus. The Hoare triples are extended with a set of faults F that are regarded as proven. The validity of the Hoare triples is intended to be partial correctness modulo faults in F:

$$\Gamma \models_{/F} P\ c\ Q \equiv \Gamma \vdash \langle c,\ Normal\ s\rangle \Rightarrow t \longrightarrow s \in P \longrightarrow t \notin Fault\ `\ F \longrightarrow t \in Normal\ `\ Q$$

Here ` denotes the set image operation, e.g. $f\ `\ M$ can be rewritten as set comprehension: $\{f\ s.\ s \in M\}$. Given an execution of command c from an initial state s satisfying the precondition P, provided that the final state t is not a fault in set F, then the final state will satisfy postcondition Q. An empty set of faults F can be omitted, since this is the ordinary case. It ensures that no runtime faults will occur. The Hoare calculus is proven sound with respect to this notion of validity.

Theorem 1. $\Gamma \vdash_{/F} P\ c\ Q \longrightarrow \Gamma \models_{/F} P\ c\ Q$

To integrate the results of the software model checker, we derive a rule that allows to switch from an empty set F to the set $\{True\}$, which means that all guards marked with *True* can be assumed as correct. Here are the Hoare Logic rules for guards:

$$\frac{\Gamma \vdash_{/F} P\ c\ Q}{\Gamma \vdash_{/F} (g \cap P)\ Guard\ f\ g\ c\ Q} \qquad \frac{f \in F \qquad \Gamma \vdash_{/F} P\ c\ Q}{\Gamma \vdash_{/F} \{s.\ s \in g \longrightarrow s \in P\}\ Guard\ f\ g\ c\ Q}$$

The left rule is the ordinary rule for guarded commands. The guard g has to hold in the precondition. On the right hand side, however, the guard can be taken as assumption for the precondition P, since validity of Hoare triples assumes that no fault in F can occur. For interactive verification this means that guards in F can be taken as assumptions whereas the other ones have to be proven. To illustrate the integration of the software model checker, consider the following situation. The current proof goal is a Hoare triple of the form $\Gamma \vdash_{/\{\}} P\ c\ Q$, where all guards in c are marked with *False*. The software model checker proves that some of the guards in c will always hold. Formally the oracle returns a Hoare triple of the form $\Gamma \vdash_{/\{\}} P\ c'\ UNIV$, where UNIV is the universal set that denotes the trivial postcondition, and c' is the same program as c but may have less guards. The missing guards are those, that are not provable by the model checker. Referring to the semantics of Hoare triples, the result of the oracle describes that no guard in c' will fail. This is exactly what the model checker claims. The new proof state then is of the form $\Gamma \vdash_{/\{True\}} P\ c''\ Q$. In c'' the guards that are proven by the model checker are marked as *True* and the others are still marked with *False*. The following rule implements this strategy:

$$\frac{\begin{array}{cc} & \Gamma \vdash_{/\{True\}} P\ c''\ Q \\ \Gamma \vdash_{/\{\}} P\ c'\ UNIV & c' = mark\text{-}guards\ False\ (strip\text{-}guards\ \{False\}\ c'') \\ & c = mark\text{-}guards\ False\ c'' \end{array}}{\Gamma \vdash_{/\{\}} P\ c\ Q}$$

The conclusion is the current Hoare triple that is verified (e.g. Figure 1 on page 4). The first premise is the subgoal that remains for verification after the tactic *swmc_guards* is finished (e.g. Figure 4 on the preceding page). The second assumption denotes the result of the software model checker and is the only part that is provided as an oracle. The effect of the model checker is clearly integrated into the Hoare Logic proof. The side-conditions on c, c' and c'' are solved by Isabelles simplifier. The auxiliary HOL functions *mark-guards* and *strip-guards* are defined recursively on the HOL data-type of commands. The relevant equations for *Guard f g c* are:

$$\begin{array}{lcl} \textit{mark-guards } f' \textit{ (Guard } f\ g\ c\textit{)} & = & \textit{Guard } f'\ g \textit{ (mark-guards } c\textit{)} \\ \textit{strip-guards } F \textit{ (Guard } f\ g\ c\textit{)} & = & \textit{if } f \in F \textit{ then strip-guards } F\ c \\ & & \textit{else Guard } f\ g \textit{ (strip-guards } F\ c\textit{)} \end{array}$$

6 Practical Experience

In our little bubble-sort example, we had to manually discharge—even without the software model checker—only two guards. The other guards could be subsumed by the theorem prover within the proof goals for functional correctness. Hence, the completely manual proof applies basically 5 more tactics than the one that employs the software model checker.

However, our example is very light-weight: The interesting part of the proof consists of just about 40 tactics. Nevertheless, we could simplify the invariants for the while loops with the help of the model checker even in this simple example. Finding a suitable invariant is usually one of the most time consuming tasks during the verification process. This does especially apply to nested while loops.

An interesting observation is that by far the most problems of today's software are not due to a malicious algorithm but due to overlooked corner cases in the specific implementation: Bounded arithmetic and limited buffers lead to unintended over- or underflows. This observation leads to the conclusion that programmers—and possibly verification engineers—focus primarily on the abstract algorithm when implementing respectively verifying software and tend to neglect the machine dependent limitations.

Of course, there is no way around these issues on software verification. These corner cases, however, are often perceived as distraction from the "real"—the functional—verification goal. Indeed, we experienced that most verification engineers in the Verisoft project ignored the guards in the first run and only concentrated on functional correctness. In a second pass, the guards were discharged.

Hence, an automatic discharge of these rather boring premises is of great benefit for the verification engineer since it allows to concentrate on just the interesting part of his work.

7 Conclusion and future work

This work was motivated by the need of a maximum degree of automation in software verification through the use of Isabelle in the Verisoft project. We have integrated our software model checker into the Isabelle theorem prover. This integration was realized through the implementation of a tactic called *swmc_guards*.

It plays the role of an interface between the theorem prover and the model checker.

For the specific purpose of checking runtime errors that could be raised during expression evaluation, it is worth to use a model checker for two reasons: (a) It is efficiently dealing with formulas of quantifier-free first-order logic. (b) The verification engineer can focus on the verification of functional properties, and rely on the software model checker for the rather boring task to ensure the absence of runtime faults. As a result, a big number of guards can be discharged by a single tactic. This saves a lot of user time and effort.

We are planing in the future to take into account more types of guards. For example, dealing with pointer access.

References

1. William R. Bevier, Warren A. Hunt Jr., J. Strother Moore, and William D. Young. An approach to systems verification. *J. Autom. Reasoning*, 5(4):411–428, 1989.
2. Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 385–395, Washington, DC, USA, 2003. IEEE Computer Society.
3. V. Pisini et al. Formal hardware verification by integrating HOL and MDG. In *GLS-VLSI'00, Chicago, USA*, 2000.
4. Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83, London, UK, 1997. Springer-Verlag.
5. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, pages pp. 58–70. ACM Press, 2002.
6. Jeffrey J. Joyce and Carl-Johan H. Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *DAC '93: Proceedings of the 30th international conference on Design automation*, pages 469–474, New York, NY, USA, 1993. ACM Press.
7. J. Strother Moore. A grand challenge proposal for formal methods: A verified stack. In *10th Anniversary Colloquium of UNU/IIST*, pages 161–172, 2002.
8. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283. 2002. Available from `http://www.in.tum.de/~nipkow/LNCS2283/`.
9. S. Rajan, Natarajan Shankar, and Mandayam K. Srivas. An integration of model checking with automated proof checking. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 84–97, London, UK, 1995. Springer-Verlag.
10. Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452, pages 398–414, 2005.