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 tools in the area of software verification. We have integrated a newly developed software model checker into an interactive verification environment for imperative programming languages. Although the problems in software verification are mostly too hard for full automation, we could increase the level of automated assistance by discharging less interesting side conditions. That allows the verification engineer to focus on the abstract algorithm, safely assuming unbounded arithmetic and unlimited buffers.

1 Introduction

Our work is part of the Verisoft project.[1] 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 deal with such an ambitious topic has been the famous CLI stack [5] back in 1989—even though the principal researcher of the famous CLI stack project, J. S. Moore [8], declared 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.

Like in the CLI-stack project, we have several layers of abstraction. However, for the vast majority of our software, we employ a single verification environment. It was implemented on top of the general-purpose theorem prover Isabelle as an instance of the well-known Hoare calculus. Within this environment, we plan to verify the different software layers, starting from considerable parts of the micro kernel, via the operating system, up to the application level.

An interesting observation is that, by far, most of the problems of today's software are not caused by a malicious algorithm but by overlooked corner cases in

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

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

G. Sutcliffe and A. Voronkov (Eds.): LPAR 2005, LNAI 3835, pp. 381–395, 2005.

the specific implementation: Bounded arithmetic and limited buffers lead to unintended over- or underflows. Hence we conclude that programmers—and quite likely verification engineers—focus primarily on the abstract algorithm when implementing, respectively verifying, software and tend to neglect the machine dependent limitations.

Of course, the verification engineer has to address these issues at some point. Our experience in the Verisoft project is, however, that the corner cases are usually perceived as distraction from the "real"—the functional—verification goal and addressed at last.

Furthermore, a maximum degree of automation is crucial for such an ambitious project as Verisoft. However, our verification environment yet provided an interactive-only user interface. Hence we have integrated a model checker for the automatic pre-verification of side conditions as they arise due to the finiteness of the underlying machine. This integration allows verification engineers to concentrate on the abstract problem with virtually unbounded arithmetic and unlimited buffers.

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 side conditions and illustrates our verification 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 on 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. Moreover, we discuss some enhancements of the model checker to simultaneously test the reachability of multiple locations. In Section 6, we give an estimation of the speed-up to expect from the use of our tool. 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* [4] integrated the MDG tool, which 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 so that the MDG system can complete the proof.

Similarly in the context of hardware verification, Joyce and Seger [7] 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* [10] 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 by Pisini *et al* [4]. 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 [9], 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, higher-order unification and resolution in particular, one already gets a decent deductive system. Moreover, Isabelle follows the well-known *LCF system approach*, which allows us 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 [11] 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. For total correctness, we are about to integrate a termination checker in a similar fashion.

Runtime faults are modeled as explicit *guards* within the program c. Such a guard formulates constraints on the current program state. The semantics 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 $\{\!|\, i\,' \leq N \,|\!\}$ abbreviates the set comprehension $\{s.\ i\ s \leq N\}$, 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 symbol ($'$).

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. However for specifying and reasoning about programs, we want to "think unbounded". Therefore 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 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.

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)

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}	$2^{7}-1$

Figure 1 on the next page illustrates the program representation in our verification environment. It shows 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*.

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 spirit of Magic [1] and Blast [2, 3]. Most data types of the C language are handled by

```
⋀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. An external representation of code with guarded commands within our verification environment. A guarded command consists of a list of guard conditions and the affected command. The conditions are enclosed in braces: {| |}. The guard conditions are separated from the command by ↦. The term $^{\sigma}$*array-size* refers to the old value of *array-size* before the execution of the program fragment.

ACSAR, including integer types, arrays and structs. Furthermore, ACSAR supports all control structures 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. In the following, we explain the basic verification algorithm of ACSAR.

4.1 Translating Programs to Transition Constraints

A transition constraint tc is a tuple (l, g, u, d) where l and d are the values of the program counter before and 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.

```
1  void three_times(int n)
2  {
3    int s = 0, i=0, result;
4    while (i != n) {
5      s = s + 3;
6      i = i + 1;
7    }
8    result = s;
9  }
```

Function *three_times* can be represented by the following system of transition constraints:

$$
\begin{array}{ll}
(1,\ \mathit{True},\ [s \leftarrow 0,\ i \leftarrow 0],\ 4) & (1)\\
(4,\ i \neq n,\ [i \leftarrow i+1,\ s \leftarrow s+3],\ 4) & (2)\\
(4,\ i = n,\ [\mathit{result} \leftarrow s],\ 10) & (3)
\end{array}
$$

Upon translation, lines 1-3 are merged into transition constraint (1). Transition constraint (2) models the case where the control enters the loop (lines 4-8) and transition (3) represents the case where the control proceeds with the instruction after the loop (line 9) because the loop condition does not hold.

4.2 Abstraction

ACSAR uses the predicate abstraction technique [6] 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. ACSAR uses a backward search to explore the set of abstract states. Formally, we introduce:

- The set of program states S, the set of error states S_{err}, the set of predicates P (initially empty) and the set of transition constraints Tc. A state s is provided as a logical formula $s = (x_1 = v_1 \wedge x_2 = v_2 \wedge \cdots \wedge x_n = v_n)$, where x_i are program variables and v_i their values ($i \in [1, n]$).
- the abstraction function $\alpha : L \rightarrow L$ with $\alpha(s) = \bigwedge p$ such that $(p \in P \wedge s \Rightarrow p)$, where L is the set of quantifier-free first-order logic formulas restricted to program variables.
- the operator $Pre^{\#}$ that returns the previous abstract state: $Pre^{\#}(a, tc) = \alpha(wp(a, tc))$, where a is an abstract state provided as a logical formula, $tc \in Tc$ is some transition constraint, and $wp(a, tc)$ is the exact weakest precondition of a with respect to tc. Intuitively, the $Pre^{\#}$ operator provides the abstract state that reaches the state a after performing the transition tc.

Now, we can build the abstract system. We start with the abstract error state $\alpha(S_{err})$ and try to compute the least fixpoint of $Pre^{\#}$. Either we find the

least fixpoint or a counter example. If we find a counter example, we have to test its validity using a SAT solver. In case of a spurious counter example, our abstraction has been too coarse, and we have to refine it.

4.3 Refinement

When an abstraction is too coarse, i. e. we have 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 by the work of Henzinger *et al* [2]. If the backward search reaches the initial state S_I, the path leading from S_{err} to S_I is analyzed by 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 384, 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 385, 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. We have integrated the model checker via a new tactic, called *swmc-guards*. To deal with multiple guards, we have extended the verification procedure of ACSAR.

When the verification engineer applies *swmc-guards*, the current proof goal is converted into a reachability problem and presented to ACSAR. The model checker generates a reachability check report. The tactic evaluates this report and forms a new proof goal from the old one and the new results from the model checker. In the next sections, we describe this process in detail.

5.1 Conversion of the Original Proof Goal

In Isabelle, the proof goal is basically a Hoare triple with a code fragment that contains guards. The model checker, however, expects a C program with labelled error locations. Hence we have to convert the original problem. For the check against runtime faults, only the precondition and the actual code fragment with the guard conditions are of interest. Quantifier-free conditions can easily be formulated as C expressions, and the conversion of the basic commands like *WHILE* or *IF* is straightforward.

The conversion of the code fragment is primarily a syntax transformation. However, the internal representation in Isabelle is quite opulent. Hence we decided to introduce an intermediate language and implemented the transformation

in two stages. This intermediate language is much more compact and was tailormade to represent usual imperative programming languages. We expect that this approach will simplify the integration of similar tools.

The conceptual core of the conversion is the representation of precondition and guards. The precondition is an assumption, hence the guard conditions have only to be checked if the precondition holds. We represent this fact by enclosing the whole code fragment in an **if** command that has the precondition as branch condition. For each guard condition g, we introduce a distinct error location r and generate a conditional jump to this error location for the case that the condition does not hold. Consequently, r is reachable if and only if g is satisfiable.

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 its translation into a reachability problem and adding the necessary error locations.

Verification Approach. In the verification phase, we check the validity of each guard in isolation. With this approach, the verification engineer might be able to find several bugs at a time. In order to avoid the influence between guards, we have to disable all but the currently processed guard. Consider the following example:

```
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*.

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.

Let us now consider a program in terms of a set of transition constraints Tc. We introduce the subset Tc_{err} containing 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.

We disable all currently not concerned guards g_j in the following way: We build a transition constraint tc''_j from tc'_j by removing the guard condition g_j and keeping all other fields unchanged: $tc''_j = (l_j, \mathit{True}, u_j, k_j)$. Now, we remove the transition constraints tc_j and tc'_j from the set of transition constraints Tc, and add tc''_j to Tc.

Introducing Assumptions. In the previously described approach, each guard is checked in isolation from the other guards. This is equivalent to having 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 an 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 on the facing page 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 substancially shorter if we keep proven guards as assumptions.

5.3 Returning the Results to Isabelle

Our interface is implemented as a so called *oracle*. In this way, we can introduce a theorem into the verification process without giving a proof for it. In our case,

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
$Tc'_{err} = \bigcup\{tc'_i$ such that $tc_i \in Tc_{err}\}$;
$Tc''_{err} = \bigcup\{tc''_i$ such that $tc_i \in Tc_{err}\}$;
$Tc = (Tc - (Tc_{err} \cup Tc'_{err})) \cup Tc''_{err}$;
for each guard g_i such that $g_i \in G$ **do**
  $Tc = (Tc - \{tc''_i\}) \cup \{tc_i, tc'_i\}$;
  $res = check\_reach(err\_loc(g_i), Tc)$;
  switch (res)
    case *unreachable:*
      $store\_in\_report(rep, g_i, valid)$; $Tc = Tc - tc'_i$;
    case *reachable:*
      $store\_in\_report(rep, g_i, invalid)$; $Tc = (Tc - \{tc_i, tc'_i\}) \cup tc''_i$;
    otherwise:
      $store\_in\_report(rep, g_i, unknown)$; $Tc = (Tc - \{tc_i, tc'_i\}) \cup tc''_i$;
  $init()$;
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

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

Program	number of guards	verification time (in seconds)	
		without assumptions	with assumptions
selection-sort	14	5.918	2.729
bubble-sort	12	140.189	29.622

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 us 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.

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

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

To integrate the notion of discharging a guard into the Hoare calculus, the guarded command of the programming-language model described by Schirmer [11] 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 $\mapsto$ abbreviates nested guarded commands. The state of the programming language is a polymorphic HOL data-type with two constructors:

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

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. Validity of a Hoare triple $\Gamma \models_{/F} P\ c\ Q$ is defined as 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 [12].

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 us 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 marked with a fault 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 goal is to reduce this to a new proof state $\Gamma \vdash_{/\{True\}} P\ c''\ Q$, where c'' contains the same guards as c, but some may be marked as *True*. The software model checker has to prove that those guards that are marked with *True* actually 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' only contains the guards of c'' that are marked with *True*. The guards marked with *False* are missing, since those are the guards 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 following rule implements this strategy:

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

The conclusion is the current Hoare triple that is verified (e.g. Figure 1 on page 385). The first premise is the subgoal that remains for verification after the tactic *swmc-guards* is finished (e.g. Figure 4 on page 392). 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:

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

In the context of total correctness the tactic *swmc-guards* basically works the same. The Hoare triple returned by the oracle is of course still an partial correctness one. Hence the termination proof is left to the user.

6 Evaluation

It is hard to give a general measure of the speed-up that verification engineers gain by the use of the integrated model checker because it depends on the considered problem. For the shown bubble-sort implementation, for example, our verification condition generator could subsume all but two guard conditions. It took just 5 tactics to verify these conditions by hand.

However, our example is very light-weight: The interesting part of the proof consists of just about 40 tactics. Nevertheless, when employing the software model checker, we could use a simpler invariant for the while loop. 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.

7 Conclusion and Future Work

We have developed the new software model checker ACSAR, which is—in our application field—more powerful than competing tools. For example, if the code

of our bubble-sort example is presented to the model checker Blast, it only reports that the first and the last error location are not reachable. Presently we are tackling aliasing problems in order to deal with pointer dereferencing. We expect to essentially improve the treatment of pointers in the near future.

We have integrated the model checker into our verification environment. It turned out to be very helpful that we have developed our own model checker instead of integrating a standard tool. So we were able to adopt ACSAR to check multiple error locations at a time.

The translation mechanism implemented by the *swmc-guards* tactic works in two stages in order to facilitate the integration of similar automatic tools. One such tool is a termination checker, which is already integrated; other tools might follow.

Furthermore, we plan to translate properties with quantifiers. Though the side conditions in guards are always quantifier-free, quantifiers might occur in preconditions. Currently, these preconditions are not transferred to the model checker. Moreover, we examine how the integration can be improved in order to enable the model checker to reason about simple proof goals.

References

1. Sagar Chaki et al. Modular verification of software components in C. In *ICSE*, pages 385–395. IEEE Computer Society, 2003.
2. Thomas A. Henzinger et al. Lazy abstraction. In *POPL*, pages 58–70, 2002.
3. Thomas A. Henzinger et al. Software verification with BLAST. In Thomas Ball and Sriram K. Rajamani, editors, *SPIN*, volume 2648 of *LNCS*, pages 235–239. Springer, 2003.
4. V. K. Pisini et al. Formal hardware verification by integrating HOL and MDG. In M. Sarrafzadeh, P. Banerjee, and K. Roy, editors, *ACM Great Lakes Symposium on VLSI*, pages 23–28. ACM, 2000.
5. William R. Bevier et al. An approach to systems verification. *J. Autom. Reasoning*, 5(4):411–428, 1989.
6. Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.
7. Jeffrey J. Joyce and Carl-Johan H. Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *DAC*, pages 469–474, 1993.
8. J. Strother Moore. A grand challenge proposal for formal methods: A verified stack. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *LNCS*, pages 161–172. Springer, 2002.
9. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
10. S. Rajan, Natarajan Shankar, and Mandayam K. Srivas. An integration of model checking with automated proof checking. In Pierre Wolper, editor, *CAV*, volume 939 of *LNCS*, pages 84–97. Springer, 1995.
11. Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *LPAR*, volume 3452 of *LNCS*, pages 398–414. Springer, 2004.
12. Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, TU-München, 2005. to appear.