VERIGE: Verification with Invariant Generation Engine

Nicolas Latorre, Francesco Alberti, Natasha Sharygina
University of Lugano
via G. Buffi, 13
Lugano, Switzerland

ABSTRACT

Program verification systems fail in verifying programs if appropriate loop invariants are not suggested. Generation of loop invariants in general is an art and providing them manually is a highly complex task (if possible at all). In this paper we present VERIGE, a tool that integrates a verifier with an invariant generator engine. VERIGE implements a novel generic algorithm that alleviates the load on the invariant generator and consequently achieves a general speed-up of program verification.

1. INTRODUCTION

Deductively verifying that a program meets its specification is a hard task: the programmer has to annotate the program with loop invariants and other formulæ that will be exploited by the underlying program verification system in order to check the (partial[1]) correctness of the input code. In this paper we take into consideration programs with arrays. If defining good loop invariants is generally an art and definitely out of reach for the vast majority of the programmers, defining loop invariants for programs with arrays is a much harder task, as quantification comes into play. Indeed interesting properties over arrays, like "being sorted", "being initialized", etc., are expressible only by exploiting quantifiers.

The tool we present in this paper, VERIGE, alleviates the task of manually writing loop invariants and additional annotations necessary for a deductive proof of a program by coupling a program verification system, BOOGIE [5], with an invariant generator able to infer quantified invariants, SAFARI [2]. The novelty of VERIGE, differentiating it with respect to standard invariant generation techniques, lies in the way the program verification system and the invariant generator are integrated. BOOGIE takes as input an annotated program with pre-conditions, post-conditions, invariants, assertions, etc. and produces a set of first-order formulæ, called *verification conditions*, which validity implies the partial correctness of the input annotated code. The validity of this formulæ is checked by feeding them into an Satisfiability Modulo Theory (SMT) solver. The lack of some required annotation causes the failure of the SMT-solver in proving the validity of some verification conditions. In this case BOOGIE displays some *execution traces*, i.e., the portions of the input code, corresponding to the verification condition which validity has not been proven by the SMT-solver. At this point the user would have to analyze the execution traces, provide further annotations and re-run BOOGIE on the program. With VERIGE, instead, such execution traces are automatically passed to SAFARI. SAFARI will analyze the traces and either find a real error of the input code, that will be reported to the user, or *automatically* provide the additional annotations required to prove the verification conditions associated to the execution traces.

The fact that VERIGE exploits an invariant generator for *quantified properties* like SAFARI and *how* SAFARI is exploited in the verification process differentiates VERIGE from the related literature on invariant generation. Abstract interpretation approaches (e.g., [10,16]) do not ensure that the generated invariants are precise enough to prove a given postcondition. The closest work to VERIGE are, to the best of our knowledge, those presented in [14,15]. In [14] the portions of the program requiring an analysis are determined with the help of a SAT solver, while in our case such paths are determined by the failure of the SMT-solver behind BOOGIE. The approach presented in [15] exploits the same driving engine for the analysis, that is BOOGIE, but the invariant inference is performed by exploiting different abstract domains (with the risk, as discussed before, of not being able to compute sufficiently precise invariants). Our refinement technique can be viewed as an extension of the CEGAR paradigm [9], where counterexamples are not concrete executions of the program under analysis but fractions of it. The approach implemented in VERIGE differs as well from the one presented in [7] as VERIGE exploits an invariant generator which is not based on templates.

For experimentation, we tested VERIGE on various programs with arrays. BOOGIE by itself fails in verifying such programs as they lack loop invariants. Notably, VERIGE succeeds in verifying such programs. Furthermore, we com-

[1] A program S is *partially correct* iff, given a precondition P and a postcondition Q, every terminating execution of S starting in a state satisfying P, terminates in a state satisfying Q.

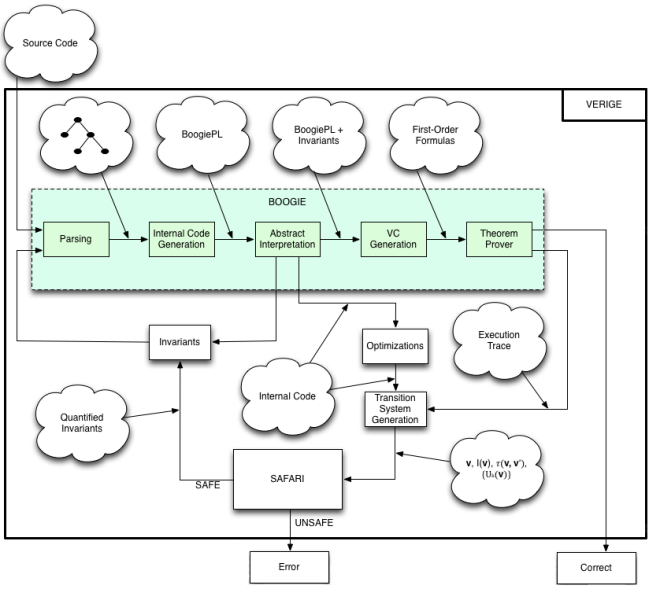

Figure 1: VERIGE **architecture.**

Algorithm 1: Lazy integration algorithm

Data: An annotated BoogiePL program $\mathcal{P}$
Result: BOOGIE verification result of the BoogiePL program with quantified invariants generated by SAFARI

```
1  begin
2      PastCEs ← ∅;
3      ⟨ ce, numErrors ⟩ ← Boogie( P );
4      while ( numErrors > 0 ) do
5          while ( ce ∈ PastCEs ) do
6              ce ← getSupersetCE(ce);
7          ir ← optimize(ce);
8          ⟨ v, I(v), τ(v, v'), U(v) ⟩ ← ce2ts( ir );
9          ⟨ result , invariant ⟩ ← SAFARI(⟨v, I(v), τ(v, v'), U(v)⟩);
10         if ( result = unsafe ) then
11             return UNSAFE, ce;
12         invariantsMap ← storeInvariants( invariant );
13         P ← toBoogie( invariantsMap, P );
14         ⟨ ce, numErrors ⟩ ← Boogie( P );
15     return VERIFIED;
```

pared the integration strategy of VERIGE with the one where the entire program is given directly to SAFARI. The experiments show that the "lazy" integration implemented in VERIGE allows to achieve a speed-up with respect to an "eager" integration between BOOGIE and SAFARI, where the former acts only as a pre-processor and the entire load of the verification is put on SAFARI. This is justified by the fact that VERIGE executes SAFARI only on those parts of the program where invariants are really needed, thus reducing the "noise" that SAFARI would have to process if executed on the entire input program.

2. THE TOOL

Figure 1 sketches VERIGE architecture, while the pseudocode in Algorithm 1 offers a description of the integration algorithm implemented in our tool. VERIGE starts by executing BOOGIE on the input program $\mathcal{P}$ (line 3). BOOGIE might return a set of execution traces to inform the user about a failure of the underlying theorem prover in checking the generated verification conditions. Such execution traces are caught by VERIGE and analyzed with the help of SAFARI. Notably, BOOGIE might return the same execution trace multiple times[2]. This might generate an infinite loop where BOOGIE will return the same execution trace for which SAFARI will provide always the same invariants. To overcome such problem, VERIGE keeps track of all executions traces analyzed in the past. If the same execution trace is detected, VERIGE generalizes the execution trace (getSupersetCE procedure at line 6). The intuition behind this step is that by analyzing a bigger portion of the program, SAFARI will generate more general invariants. The execution trace is then analyzed and optimized (line 7) to successfully exploit different heuristics and capabilities of SAFARI. From the internal representation of the execution trace VERIGE generates the transition system for SAFARI and executes the invariant generator engine. The termination of SAFARI gives two outcomes: either a true counterexample or an invariant for the transition system representing the portion of the program analyzed. The invariant is finally added to the BoogiePL program, which is again analyzed by BOOGIE.

Notably, while VERIGE exploits BOOGIE and SAFARI, the integration framework can instantiated with different program verification systems and invariant generation engines. Next we describe in more details the implementation details of VERIGE.

2.1 Code optimization

VERIGE works on the code printed by BOOGIE when executed with the option `-printInstrumented`. The BoogiePL code obtained with this option is the result of different optimizations performed on the input source code [6]. It is a set of *basic blocks* , where each block has a label, a possibly empty set of assumptions, a set of instructions, a possibly empty set of assertions and a possibly empty set of `goto` labels describing successor blocks.

Execution traces are translated by VERIGE into transition systems accepted by SAFARI. The translation process implemented in VERIGE optimizes the code in order to better exploit SAFARI capabilities. The main optimizations performed by VERIGE are the following.

OPT I. The first optimization targets the reduction of redundant or "useless" variables. These can be temporary variables used for e.g., expressing array swaps (swaps can be encoded in the language accepted by SAFARI without introducing a temporary variable since array updates are expressed as case-defined functions) or variables of the original program that are not involved in the computation of the subprograms analyzed with the execution traces.

OPT II. The second optimization separates blocks that contain *dependent instructions*. For each pair of instructions of a block, i_1 and i_2, i_1 and i_2 are dependent if i_1 is executed before i_2 and i_1 defines a variable used by i_2. The optimization separates two dependent instructions in two different, but related, blocks.

[2]This can be due to several reasons, analyzed in detail in Section 2.4.

OPT III. The third optimization is required to match syntactic restrictions of SAFARI input language (e.g., guards flattening[3], maximum number of array indexes per transition, etc.).

2.2 Transition system generation

Given an optimized internal representation of the BoogiePL program, VERIGE can generate the corresponding transition system $S = (\mathbf{v}, I(\mathbf{v}), \tau(\mathbf{v}, \mathbf{v}'))$. The optimized representation is a sequence of blocks of the kind

$$\begin{aligned} l_{i_0} : \\ &\mathsf{assume}\ \phi(\mathbf{v}) \\ &Upd(\mathbf{v}, \mathbf{v}') \\ &\mathsf{goto}\ l_{i_1}, \ldots, l_{i_n} \end{aligned} \tag{1}$$

where $Upd(\mathbf{v}, \mathbf{v}')$ is a conjunction of statements of the kind

$$v'_j := E(\mathbf{v})$$

for all $v'_j \in \mathbf{v}'$. E is a general expression over program variables. For each block we generate n transitions of the kind

$$pc = l_{i_0} \wedge \phi(\mathbf{v}) \wedge Upd(\mathbf{v}, \mathbf{v}') \wedge pc' = l_{i_j} \tag{2}$$

for $1 \leq j \leq n$. The formula $U(\mathbf{v})$, representing the unsafe states for which we want to test the reachability from $I(\mathbf{v})$ by a repeated application of $\tau(\mathbf{v}, \mathbf{v}')$, is generated from the assertion that, according to last BOOGIE execution, "might not hold". SAFARI is then executed. SAFARI implements a backward reachability analysis enhanced with abstraction-refinement features [1]. In the context of this work, SAFARI is exploited in order to generate safe inductive invariants, which are provided by enabling the option `display_safe_invariant`.

2.3 Annotating the BoogiePL source code

If SAFARI proves the safety of its input, it returns an invariant of the kind

$$\bigwedge_{l_i \in L} pc = l_i \rightarrow \psi_{l_i}(\mathbf{v})$$

where pc is a fresh variable with respect to $\mathbf{v}$ and has been introduced by VERIGE for translating the execution traces into transition systems. VERIGE will add each conjunct in the corresponding block of the control-flow graph defined by BOOGIE by adding new assert statements of the kind

$$\mathsf{assert}(\psi_{l_i}(\mathbf{v}));$$

Notably, by adding new assertions to the code, VERIGE is *mimicking the human interactive process of examining the execution traces provided by the program verification systems* and adding new "facts" that, if confirmed, may help in proving the safety of the code. We remark that the formulæ generated by SAFARI are added as new assertions in the code. That is, BOOGIE will take advantage of these formulæ by re-verify that they are indeed inductive annotations. Another strategy would have been adding the formulæ as assumptions. In this case, however, the problem would have been weakened significantly, with the risk of becoming unsound.

[3]A formula is said to be *flat* if the array variables are indexed only by existentially quantified variables.

2.4 Execution traces generalization

Verification conditions generated by BOOGIE are (quantified) formulæ over some theory of interest T specifying the semantics of the program instructions. In our experimental setting, T will be the theory of arrays $T_A^{\mathbb{Z}}$, i.e., the theory of arrays having Presburger arithmetic for interpreting array indices [8]. Deciding the validity of quantified formulæ over $T_A^{\mathbb{Z}}$ is, in general, undecidable (only some quantified fragments of $T_A^{\mathbb{Z}}$ admits decision procedures [3, 8, 13]). In order to deal with this theoretical limitation, BOOGIE relies on the *matching modulo equalities* (E-matching) heuristic [12] implemented in the Z3 SMT solver [11]. Such heuristic exploits a given pattern to find suitable instances for the universally quantified variables. If the instantiated quantifier-free formula is inconsistent, the solver can correctly say that the input quantified formulæ are unsatisfiable, implying the validity of the given verification conditions. Otherwise it would simply return an `unknown` message for those not revealing inconsistent cores[4]. In the case of BOOGIE, the patterns for driving the E-matching procedure are automatically generated according to some internal heuristics. Given the complexity of the verification conditions generated when verifying array programs, it might well happen to receive `unknown` outcomes from the SMT-Solver, even for proof obligations representing paths already refined by SAFARI. This can happen because SAFARI has its own internal procedures for instantiating quantifiers [2], and the E-matching pattern produced by BOOGIE might not be able to replicate the instantiations performed by SAFARI to prove the inductiveness of some annotations. Clearly, this situation leads to an infinite loop if not handled properly.

We devised to overcome this situation by generalizing the execution traces. That is, VERIGE caches all the execution traces returned by BOOGIE. If BOOGIE returns more than once the same execution trace, VERIGE generalizes it (lines 5-6 of Algorithm 1) by returning a portion of the original program including the execution traces returned by BOOGIE. More precisely, let B be the set of basic blocks in the execution trace returned by BOOGIE, and let $b_{in} \in B$ be the entry block of the execution trace and $b_e \in B$ the exit block. VERIGE first considers if the original program admits a path from b_{in} to b_e involving some blocks $\{b'_1, \ldots, b'_n\}$ not belonging to B. If so, $B \cup \{b'_1, \ldots, b'_n\}$ becomes the new execution trace. If it is not possible to enlarge B by considering other paths from b_{in} to b_e, VERIGE searches for find a portion of the original control-flow graph including B either by starting from an ancestor b'_{in} of b_{in} or ending in a block b'_e which is reachable from b_e.

3. EXPERIMENTAL EVALUATION

VERIGE[5] was tested on a variety of state-of-the-art benchmarks. We focused our experimentation on programs with arrays since they pose non-trivial problems and a good playground to test capabilities/performance of the verification tools. The programs we selected perform various operations over arrays, like copying one array into another, sorting an

[4]BOOGIE files produced to check the validity of verification conditions exploits the incrementality of the SMT-solver in such a way that it is possible to precisely recognize on which verification conditions the solver failed.

[5]Available at `http://atelier.inf.usi.ch/~latorren/verige/verige.html`

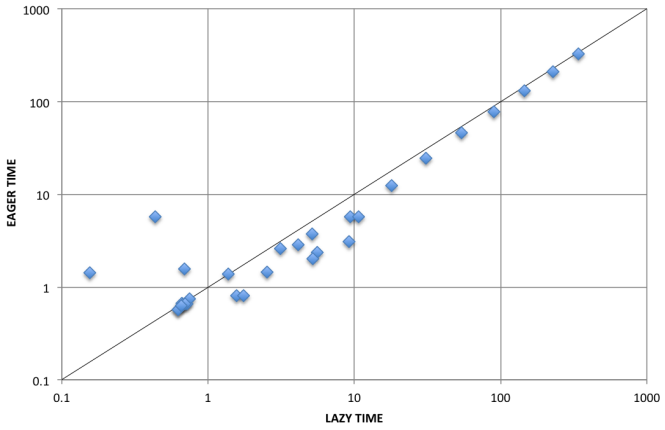

Figure 2: Comparison between the invariant generation time (in seconds) in the eager and lazy mode.

array, initializing all element of an array (or part of them) to a given value, finding the maximum or minimum element of an array, etc. All these programs are annotated with post-conditions or assertions stating interesting properties about arrays.To the best of our knowledge, highly engineered verification engines generally fail to verify such programs[6]. Given the lack of loop invariants, BOOGIE alone cannot verify any of them either. In addition we point out that inferring such invariants by hand is definitely far from being an easy and trivial process (see Appendix A for a worked out example).

We explicitly focused our experimentation on safe programs in order to evaluate the effectiveness of our new lazy integration. In particular, we wanted to compare our new integrated framework against a more classical one where the invariant generation engine works on the entire program. With respect to the new lazy integration, the latter one we call *eager*. VERIGE can execute both integration modes (the lazy integration is enabled by the command line option `-lazy`).

Experimentation has been performed on a machine equipped with an Intel i5 @ 2.53GHz processor, 4GB of RAM and running OSX 10.6.8. We used BOOGIE 4.2 and z3 4.1. Figure 2 reports a comparison of the verification times of VERIGE executed in lazy and eager mode. This experimental evaluation shows clearly that the lazy approach is not overloading the verification process. This is because the points below the diagonal, representing the examples where the eager approach is faster than the lazy one, are not far from the diagonal. However the points above the diagonal are definitely far from it. This means that, in general, the lazy integration might introduces a small overhead, but also can achieve a substantial speed-up (up to an order of magnitude). To rephrase in less formal words: by adopting the lazy approach the user is not loosing anything with the chance of gaining a lot. For this reason, the lazy integration constitutes a valid alternative to the more standard verification approach where the entire load of the verification is on the invariant generation.

[6]For example, the CodeContracts verification system, which implements invariance-inference algorithms for arrays, as described in [10], generally fails on our benchmarks.

4. CONCLUSION AND FUTURE WORK

We presented a new tool, VERIGE, where the program verification system BOOGIE and the invariant generator SAFARI are successfully integrated. We tested it on various challenging programs with arrays. Experimentally, VERIGE succeeds and is competitive with respect to other state-of-the-art tools. As a future work, it would be interesting to try different refinement strategies since it is a crucial part of the integration and designing new enhanced procedures for this task might lead to further improvements in verification of complex programs.

5. REFERENCES

[1] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Lazy abstraction with interpolants for arrays. In *LPAR*, pages 46–61, 2012.

[2] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. SAFARI: SMT-Based Abstraction for Arrays with Interpolants. In *CAV*, pages 679–685, 2012.

[3] F. Alberti, S. Ghilardi, and N. Sharygina. Decision procedures for flat array properties. In *TACAS*, pages 15–30, 2014.

[4] F. Alberti and N. Sharygina. Invariant generation by infinite-state model checking. In *2nd International Workshop on Intermediate Verification Languages*, 2012.

[5] M. Barnett, B.Y.E. Chang, R. DeLine, B. Jacobs, and K.R.M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[6] M. Barnett and K.R.M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, pages 82–87, 2005.

[7] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, pages 300–309, 2007.

[8] A.R. Bradley, Z. Manna, and H.B. Sipma. What's decidable about arrays? In *VMCAI*, pages 427–442, 2006.

[9] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[10] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118, 2011.

[11] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[12] D.L. Detlefs, G. Nelson, and J.B. Saxe. Simplify: a theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.

[13] Y. Ge and L. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *CAV*, pages 306–320, 2009.

[14] W.R. Harris, S. Sankaranarayanan, F. Ivancic, and A. Gupta. Program analysis via satisfiability modulo path programs. In *POPL*, pages 71–82, 2010.

[15] K.R.M. Leino and F. Logozzo. Loop invariants on demand. In *APLAS*, pages 119–134, 2005.

[16] A. Podelski and T. Wies. Counterexample-guided focus. In *POPL*, pages 249–260, 2010.

APPENDIX
A. A WORKED-OUT EXAMPLE

In this section we show, by following step by step the execution of VERIGE, how the lazy procedure interleaves SAFARI and BOOGIE. The source code we take into account for this demonstration is the well-known bubbleSort procedure (reported in Figure 3). We point out that manually finding suitable loop invariants for this procedure, which has a nested loop, is far from being trivial.

For such procedure, VERIGE generates an Internal Representation retrieved from the output of BOOGIE when executed with the option `-printInstrumented` (see Figure 4). Subsequently, our lazy approach requires a BOOGIE execution on the original source code. In the case of the bubbleSort procedure, given the lack of loop invariants, BOOGIE is not able to check that the implementation obeys to its specifications. The counterexample returned by boogie is the sequence of blocks (b0, b5LH, b5LD, return) of the bubbleSort control-flow graph (Figure 4). At this point, SAFARI enters the scene. VERIGE generates the transition system $S^{(1)} = (\mathbf{v}, I(\mathbf{v})^{(1)}, \tau(\mathbf{v}, \mathbf{v}')^{(1)})$, with $\mathbf{v} = pc, a, i, sw$ and $I^{(1)}(\mathbf{v}), \tau(\mathbf{v}, \mathbf{v}')^{(1)}$ defined as follows:[7]

$$I(\mathbf{v})^{(1)} := pc = \mathsf{b0}$$

$$\tau(\mathbf{v}, \mathbf{v}')^{(1)} := \begin{pmatrix} (pc = \mathsf{b0} \wedge sw' \wedge pc' = \mathsf{b5LH}) \vee \\ (pc = \mathsf{b5LH} \wedge pc' = \mathsf{b5LD}) \vee \\ (pc = \mathsf{b5LD} \wedge \neg sw \wedge pc' = \mathsf{return}) \end{pmatrix}$$

Notice that this transition relation does not admit any loop path. In such cases, SAFARI cannot diverge. VERIGE detects the absence of loops and executes SAFARI without enabling abstraction-refinement features. SAFARI, in turn, shows that the transition system $S^{(1)}$ cannot reach the unsafe formula

$$pc = \mathsf{return} \wedge \exists x, y.(0 \leq x \wedge x < y \wedge y < L \wedge a[x] > a[y])$$

and returns a safe inductive invariant for $S^{(1)}$, made by the conjunction of the following formulæ:

$$\begin{aligned} pc = \mathsf{return} \quad &\rightarrow \forall z_0, z_1.((0 \leq z_0 \wedge z_0 < z_1 \wedge z_1 < L) \rightarrow (a[z_0] \leq a[z_1])) \\ pc = \mathsf{b5LD} \quad &\rightarrow \forall z_0, z_1.((0 \leq z_0 \wedge z_0 < z_1 \wedge z_1 < L \wedge \neg sw) \rightarrow (a[z_0] \leq a[z_1])) \\ pc = \mathsf{b5LH} \quad &\rightarrow \forall z_0, z_1.((0 \leq z_0 \wedge z_0 < z_1 \wedge z_1 < L \wedge \neg sw) \rightarrow (a[z_0] \leq a[z_1])) \end{aligned}$$

The invariants are plugged into the Internal Representation of the code as new assertions, and BOOGIE is executed on the code just annotated. Once again, BOOGIE complains that the underlying SMT-Solver (in our case, Z3) cannot verify the program, and it returns a new counterexample. This time, the counterexample traverses the blocks (b0, b5LH, b5LB, b6LH, b6LD) of the control-flow graph depicted in Figure 4. VERIGE analyzes it and produces the transition system $S^{(2)} = (\mathbf{v}, I(\mathbf{v})^{(2)}, \tau(\mathbf{v}, \mathbf{v}')^{(2)})$ made by the following formulæ:

$$I(\mathbf{v})^{(2)} := pc = \mathsf{b0}$$

$$\tau(\mathbf{v}, \mathbf{v}')^{(2)} := \begin{pmatrix} (pc = \mathsf{b0} \wedge sw \wedge pc' = \mathsf{b5LH}) \vee \\ (pc = \mathsf{b5LH} \wedge pc' = \mathsf{b5LB}) \vee \\ (pc = \mathsf{b5LB} \wedge sw \wedge \neg sw' \wedge i' = 1 \wedge pc' = \mathsf{b6LH}) \vee \\ (pc = \mathsf{b6LH} \wedge 1 \leq i \wedge pc' = \mathsf{b6LD}) \vee \\ (pc = \mathsf{b6LD} \wedge L \leq i \wedge pc' = \mathsf{b5LH}) \end{pmatrix}$$

In this case, since the transition system induces a control-flow graph with a loop, SAFARI is executed enabling abstraction-refinement features. As before, SAFARI detects that $S^{(2)}$ cannot reach

$$pc = \mathsf{b5LH} \rightarrow \forall z_0, z_1.((0 \leq z_0 \wedge z_0 < z_1 \wedge z1 < L \wedge \neg sw) \rightarrow (a[z_0] \leq a[z_1]))$$

and generates the following intermediate assertions for BOOGIE:

$$\begin{aligned} pc = \mathsf{b5LH} \quad &\rightarrow \forall z_0, z_1.((0 \leq z_0 \wedge z_0 < z_1 \wedge z_1 < L \wedge \neg sw) \rightarrow (a[z_0] \leq a[z_1])) \\ pc = \mathsf{b5LD} \quad &\rightarrow \forall z_0, z_1.((0 \leq z_0 \wedge z_0 < z_1 \wedge z_1 < i \wedge \neg sw) \rightarrow (a[z_0] \leq a[z_1])) \\ pc = \mathsf{b6LH} \quad &\rightarrow \forall z_0, z_1.\left(\begin{pmatrix} 0 \leq z_0 \wedge z_0 < z_1 \wedge z_1 < i \wedge \\ \wedge\, i \geq 1 \wedge \neg sw \end{pmatrix} \rightarrow (a[z_0] \leq a[z_1])\right) \end{aligned}$$

The invariants generated in this second refinement are added to those generated before. With this bigger set of annotations, BOOGIE verifies successfully the bubbleSort procedure.

[7] For the sake of readability, we are omitting identical updates of the kind $v' = v$ when reporting transition relations.

```
procedure bubbleSort ( a[ ] , N ) {
    sw = true;
    while ( sw ) {
        sw = false; i = 1;
        while ( i < L ) {
            if ( a[i − 1] > a[i] ) {
              t = a[i]; a[i] = a[i − 1]; a[i − 1] = t;
              sw = true;
            }
            i = i + 1;
        }
    }
    assert ( ∀x, y.(0 ≤ x < y < N) → a[x] ≤ a[y] );
}
```

Figure 3: The procedure bubbleSort**.**

```
b0:
  sw := true;
  goto b5LH;

b5LH:
  goto b5LD, b5LB;

b5LB:
  assume sw = true;
  sw := false;
  i := 1;
  goto b6LH;

b6LH:
  assume 1 <= i;
  goto b6LD, b6LB;

b6LB:
  assume i < L;
  goto b7T, b7E;

b7T:
  assume a[i - 1] > a[i];
  t := a[i]; a[i] := a[i - 1];
  a[i - 1] := t; sw := true;
  goto b4;

b7E:
  assume a[i] >= a[i - 1];
  goto b4;

b4:
  i := i + 1;
  goto b6LH;

b6LD:
  assume L <= i;
  goto b5LH;

b5LD:
  assume sw = false;
  return;
```

Figure 4: Internal Representation of the bubbleSort **procedure. The entry block of the procedure is** `b0`**, the exit block is** `return`**.**