Verifying Invariant Based Programs in the SOCOS Environment

Ralph-Johan Back, Johannes Eriksson and Magnus Myreen
Turku Centre for Computer Science
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, FIN-20520, Turku, Finland
{backrj,joheriks}@abo.fi, magnus.myreen@cl.cam.ac.uk

Abstract

An invariant based program is a state transition diagram consisting of nested situations (predicates over program variables) and transitions between situations (predicate transformers). Reasoning about correctness is performed in a local fashion by examining each situation at a time and proving that the situation is satisfied for all possible executions. Since the invariants are in place from the beginning and the verification conditions are easily extracted from the diagram there is no need for complicated proof rules, making invariant diagrams a suitable notation for introducing formal verification to students and programmers. Our preliminary experience from using invariant diagrams in the classroom has prompted the need for a tool to support the method: we introduce here SOCOS, an environment for invariant based programming. SOCOS generates correctness conditions based on weakest precondition semantics, and the user can attempt to automatically discharge these conditions using the Simplify theorem prover; conditions which were not automatically discharged can be proved interactively in the PVS theorem prover.

Keywords: Invariant based programming, verification, SOCOS

1. INTRODUCTION

In *invariant based programming* [1, 3] the programmer starts by formulating the specifications and the internal loop invariants before the program code itself. If strong enough, invariants can be used to prove the correctness of the program. To automate this step, we have previously developed a static checker [5], which generates verification conditions for procedural invariant based programs and sends them to an external theorem prover. In this paper we continue on the same topic by presenting the SOCOS tool, an effort to extend this checker and embed it into a diagrammatic programming environment.

A SOCOS procedure is described by an *invariant diagram* rather than a textual specification. Such a diagram consists of *situations* and *transitions* between situations. Situations are state predicates and describe invariants, while transitions are predicate transformers and describe statements. Situations can be nested to indicate strengthening of the invariant—the nested situation represents a smaller set of states than the outer situation. Invariant diagrams are superficially similar to state charts, but serve a different purpose: rather than modeling control flow, invariant diagrams describe the structure of invariants and can be used to prove the correctness of the program.

We have already used invariant based programming and invariant diagrams in teaching formal reasoning to students, albeit on a limited scale.[1] Our preliminary experience indicates that the notation is intuitive and easy to pick up, even for subjects who have had little or no training in formal methods. It is also quite straightforward for them to extract the proof conditions from the diagram and prove these manually. However, even small programs generate several lemmas to

[1]One graduate level course, not part of the standard curriculum. We have also held a number of three-hour sessions with selected students, where they are asked to use invariant diagrams to implement and reason about an algorithm which they are not familiar with.

be proved, the majority of which are rather trivial. Since students quickly become frustrated with mundane “busy-work” and lose interest, our motivation is to provide a programming environment which allows focusing on the difficult proof obligations.

A SOCOS program can be developed incrementally until total correctness is achieved. Reasoning is carried out locally and the user is not burdened with a large proof task up front. SOCOS provides three main advantages over manual checking of invariant diagrams: firstly, it removes the tedium of checking trivial verification conditions; secondly, it automates the run-time checking of contracts and invariants; and thirdly, it provides an intuitive visual feedback when something goes wrong.

Run-time checking of pre- and postconditions and invariants in order to find errors is the central idea of Design by Contract [10]. Most tools which support Design by Contract do not, however, provide static checking. Our approach is probably closest to the ESC/Java2 [9] and Spec# [6] static checkers, and uses the same underlying theorem prover, Simplify [7]. The main difference is that rather than adding specifications and invariants to an existing language, we start with a simple notation based on nested invariants.

The remainder of this paper is structured as follows. Section 2 briefly describes the SOCOS user interface. In Section 3 we discuss the semantics of SOCOS programs and the generation of proof conditions. Section 4 provides a use case of SOCOS as we demonstrate the implementation of a simple sorting program. Section 5 concludes with some general observations and a summary of on-going research.

2. THE SOCOS DIAGRAM EDITOR

Programs are constructed in the SOCOS invariant diagram editor (Figure 1). The editor is very similar in style to UML statechart editors. Any number of diagrams can be open simultaneously. To the left of the diagram is an outline editor for browsing model elements hierarchically, and the bottom pane holds property editors and various communication windows.

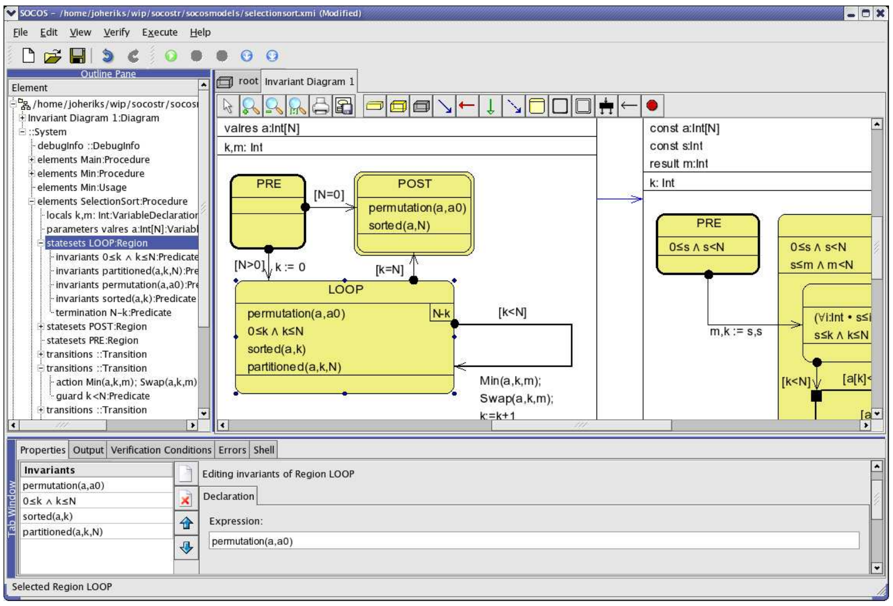


FIGURE 1: Invariant diagram editor of SOCOS

Diagrams can be compiled, executed and debugged in the editor. Execution is animated and the program state can be inspected. If run-time checking is enabled, invariants and assertions are evaluated during execution of a situation: if an invariant evaluates to false, the execution halts. SOCOS can evaluate only a pre-defined subset of all expressible invariants, namely arithmetic expressions and Boolean expressions containing only bounded quantification. We provide an overview of the debugging functionality of SOCOS in our technical report [4].

3. PROVING CORRECTNESS OF INVARIANT DIAGRAMS

SOCOS supports interactive and non-interactive verification of invariant diagrams. It generates the verification conditions and sends them to proof tools. At the time of writing two proof tools are supported: Simplify [7] and PVS [11]. Simplify is a validity checker that suffices to automatically discharge simple verification conditions such as conditions on array bounds. PVS is an interactive proof environment in which the user may verify the correctness of parts that Simplify is unable to check.

3.1. Verification Condition Generation

SOCOS generates verification conditions using MathEdit [5]. Three types of verification conditions are generated: consistency, completeness and termination conditions. All of these use the weakest precondition semantics as their basis [8]. The consistency conditions ensure that the invariants are preserved; completeness conditions that the program is live; and termination conditions that the program does not diverge.

Consistency:

A program is consistent whenever each transition is consistent. A transition from I_1 to I_2 realized by program statement S is consistent iff

$$I_1 \Rightarrow wp.S.I_2 .$$

Completeness:

A program is complete whenever each nonterminal situation is complete. A situation I is complete iff

$$I \Rightarrow wp.S^*.\mathsf{false}$$

where S^* is the transition tree from I with each branch being an $\mathsf{if} \ldots \mathsf{fi}$ statement and each leaf being magic.[2]

Termination:

A program does not diverge if the program graph can be divided into subgraphs, such that the transitions in between the subgraphs constitute an acyclic graph and each subgraph is terminating. A subgraph of the program diagram is terminating if (i) it is acyclic or (ii) has a bounded variant that decreases on each cycle within that subgraph.[3]
The cycles considered in case (ii) can consist of any number of transitions that do not increase the subgraph's variant (v below)

$$I_1 \wedge (v_0 = v) \Rightarrow wp.S.(0 \leq v \leq v_0) \tag{1}$$

as long as each cycle contains one transition (indicated by the user) that strictly decreases the subgraph's variant:

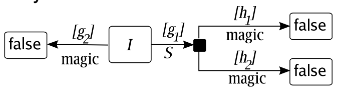
$$I_1 \wedge (v_0 = v) \Rightarrow wp.S.(0 \leq v < v_0) \,. \tag{2}$$

The termination conditions are generated for the transitions that make up cycles in the program graph.[4]

The interested reader is referred to [3] for a more detailed presentation of the notion of correctness of invariant diagrams.

---

[2]In proving completeness, we disregard the statements at the leaves by replacing them with miracles. A simple example may be useful here:



The completeness condition for situation I in this case is:
$I \Rightarrow wp.\mathsf{if}\ g_1 \rightarrow S;\ \mathsf{if}\ h_1 \rightarrow \mathsf{magic}\ []\ h_2 \rightarrow \mathsf{magic}\ \mathsf{fi}\ []\ g_2 \rightarrow \mathsf{magic}\ \mathsf{fi}\ .\mathsf{false}$,
which is equivalent to: $I \Rightarrow (g_1 \Rightarrow (wp.S.(h_1 \vee h_2))) \wedge (g_1 \vee g_2)$

[3]SOCOS will automatically divide the program graph into the smallest possible subgraphs that constitute an acyclic graph and then require that the situations within the subgraph are annotated with identical variants.

[4]Termination and consistency conditions are actually merged together so as to avoid duplication of proof efforts. Their structure allows them to be merged: $I_1 \wedge (v_0 = v) \Rightarrow wp.S.(I_2 \wedge (0 \leq v < v_0))$ and similarly for the case $v \leq v_0$.

3.2. Interaction with External Tools

SOCOS communicates through MathEdit with external proofs tools. Interfaces to PVS and Simplify are currently implemented in MathEdit. The prover to be used can be chosen on the level of single situations and transitions, with Simplify being the default. The interface to Simplify is from the user's point of view non-interactive. Behind the scenes MathEdit runs an interactive session with Simplify. MathEdit sets up the logical context and then checks the validity of each verification in turn, splitting the verification conditions to pinpoint problematic cases. For a more detailed description of the interaction with Simplify see [5]. Interaction with PVS is made simple. By clicking a button in SOCOS, MathEdit produces a theory file containing the verification conditions and starts PVS which opens the generated theory file. By default PVS applies a modified version of the `grind` tactic to all verification conditions.

4. EXAMPLE: SORTING

We illustrate the use of SOCOS by implementing a simple in-place sorting algorithm, selection sort. Selection sort works by partitioning an array into two portions, one sorted followed by one unsorted. Each iteration of the main loop exchanges the smallest element from the unsorted portion with the element immediately after the already sorted portion, thus extending the latter by one. The loop terminates when no elements are left in the unsorted portion.

4.1. Specification and Implementation

Figure 2 shows the invariant diagram for the $\mathsf{SelectionSort}$ procedure, which sorts an integer array a with elements indexed from 0 to $\mathsf{N}-1$. $\mathsf{SelectionSort}$ uses two helper procedures, Min and Swap. Min finds the index of the minimal element in the subarray $\mathsf{a}[\mathsf{s}..\mathsf{N})$ and returns it in the result parameter m, while Swap exchanges the two elements at indexes m and n in a. Each rectangular box in the diagram represents a procedure. Dependency arrows [2] between procedures connect callers with callees. Rounded boxes with decorated borders represent initial situations (named PRE) and final situations (named POST), while rounded boxes with simple borders represent internal situations. Transitions connect situations and are labeled with guards (enclosed in brackets) and action statements. The termination conditions (variants) for the two recurring situations in $\mathsf{SelectionSort}$ and Min (both called BODY) are written in the upper right corner of the corresponding situation. The lower bound 0 is implicit.

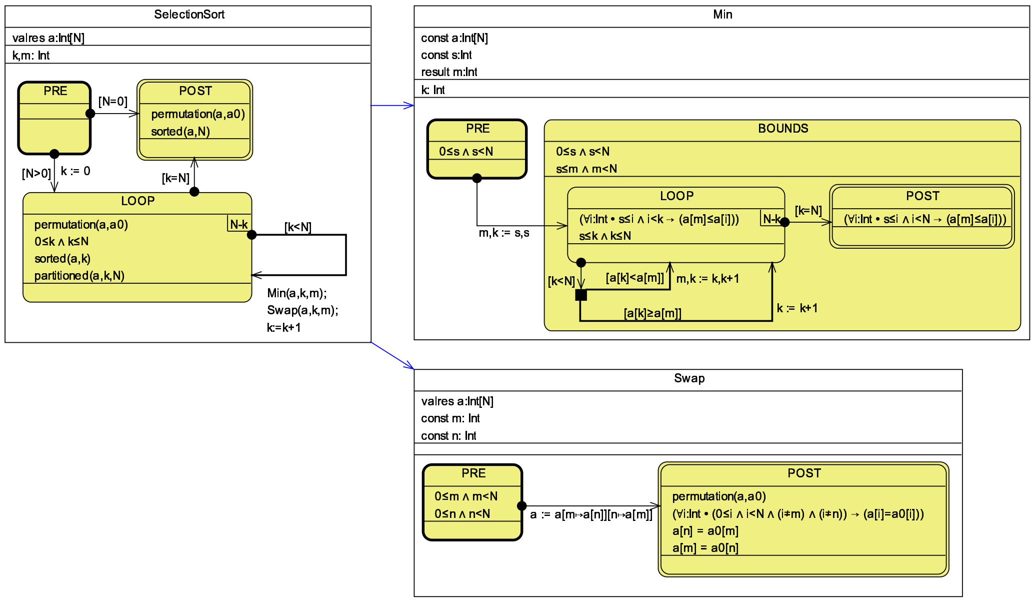


FIGURE 2: Selection sort

4.2. Verification

We will now attempt to verify the program. By default SOCOS sends all verification conditions to Simplify. In this case Simplify was able to discharge all but one of the verification conditions (there are 14 in total). While all conditions for SelectionSort and Min are automatically discharged, problems occur due to the use of permutation in Swap. SOCOS has now pinpointed a specific condition that Simplify was unable to prove and that we need to check (Figure 3).

```
Verification initiated for SelectionSort, Swap and Min.
99.7% of the verifications were proved automatically.
Condition: POST (Swap)
Assumptions:
   0 < N
   0 ≤ m
   m < N
   0 ≤ n
   n < N
   a0 = a
Imply:
   permutation(a0, a[m ↦ a[n]][n ↦ a[m]])
```

FIGURE 3: Remaining condition for Swap

We should prove that swapping two elements maintains a permutation of the original array. However, permutation is a higher-order property, and we can not give a definition of permutation that Simplify can use. In this situation we can proceed in two ways—we can temporarily get rid of the error by adding assumptions: in the case of Swap we would add an assumption statement, $[\text{permutation}(a, a0)]$, following the assignment statement in the transition from PRE to POST if we "believe" that $a[m \mapsto a[n]][n \mapsto a[m]]$ is indeed a permutation of a. During initial development it can be a useful way of postponing proofs until the final structure of invariants has been established. SOCOS will always warn that an assumption is being used. Alternatively, we can start proving the remaining condition interactively in PVS. The PVS language is expressive enough to allow us to provide a higher-order definition of permutation. An array is a permutation of another array if there exists a one-to-one correspondence (a bijective function) between the sets of indexes which, when applied to one array yields the other array. This definition is actually part of the SOCOS background theory library which is automatically loaded when PVS verification is started from SOCOS. It is easy to prove the remaining condition in PVS; however, to conserve space we have not included the actual proof here.

For brevity we presented a correct implementation of selection sort and then verified it. In practice we will not be able to arrive at a correct program in one step. There will be errors in the invariants, errors in the statements, and missing or too weak invariants. The SOCOS workflow addresses this problem: the checker can be invoked on a single transition or situation, enabling us to work locally and incrementally increase the ratio of proved conditions. PVS proofs are saved and can be replayed at any time to check that correctness is maintained after a change.

5. CONCLUSION AND FUTURE WORK

We have here presented SOCOS, a tool for reasoning about the correctness of invariant diagrams. All but the most trivial of programs generate a large amount of conditions to be proved. However, most of these are rather trivial and are automatically discharged by Simplify or the PVS `grind` tactic. For more difficult conditions, the proofs can be completed interactively in PVS.

We have carried out a number of experiments in invariant based programming to assess the educational merits of the method. Invariant based programming has also been used in one graduate level course, but it is not, however, part of the standard curriculum. Our initial experience has been encouraging, and has also highlighted the need for tool support. We are currently planning to teach invariant based programming to a group of undergraduate students, using

SOCOS as the programming tool. Compared to working with a purely conceptual notation, a tool such as SOCOS greatly simplifies grading because it gives teachers and students a common program interchange platform. Also, students can more quickly advance to bigger examples and trickier problems since they need not be concerned with proving trivial conditions.

The SOCOS user interface needs improvement. Firstly, the incremental workflow requires invoking the correctness checker frequently, but automated verification is computationally taxing and can take a long time. We are considering *background checking* to alleviate this problem—instead of having a separate verification cycle, the proof checker runs continuously in the background and discharges conditions as they are generated while the user is entering the program. Secondly, SOCOS currently supports basic proof management, but does not accommodate continuous change very well. PVS proofs must be managed by hand, and if the program is changed, however slightly, all proofs must be replayed. It would be desirable to keep track of dependencies between program elements, and in the event of a change, only require replaying proofs of possibly invalidated elements.

Finally, the issue of usability and scalability of the available programming constructs must be addressed. SOCOS currently only supports simple procedures, making it unsuitable for larger programs. We are currently adding support for abstract data structures and object-orientation to address this issue.

REFERENCES

[1] R.-J. Back. Invariant based programs and their correctness. In W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, number 223-242. MacMillan Publishing Company, 1983.
[2] R.-J. Back. Incremental software construction with refinement diagrams. In H. Broy, Gunbauer and Hoare, editors, *Engineering Theories of Software Intensive Systems*, NATO Science Series II: Mathematics, Physics and Chemistry, pages 3–46. Springer, Marktoberdorf, Germany, 2005.
[3] R.-J. Back. Invariant based programming. In S. Donatelli and P. S. Thiagarajan, editors, *ICATPN*, volume 4024 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2006.
[4] R.-J. Back, J. Eriksson, and M. Myreen. Testing and verifying invariant based programs in the SOCOS environment. Technical Report 797, TUCS - Turku Centre for Computer Science, Turku, Finland, December 2006.
[5] R.-J. Back and M. Myreen. Tool support for invariant based programming. In *the 12th Asia-Pacific Software Engineering Conference*, Taipei, Taiwan, December 2005.
[6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, 2004.
[7] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
[8] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
[9] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.
[10] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
[11] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 411–414, New Brunswick, NJ, USA, July/August 1996. Springer-Verlag.