Ralph-Johan Back | Johannes Eriksson | Magnus Myreen

Testing and Verifying Invariant Based Programs in the SOCOS Environment

TURKU CENTRE *for* COMPUTER SCIENCE

TUCS Technical Report
No 797, December 2006

Testing and Verifying Invariant Based Programs in the SOCOS Environment

Ralph-Johan Back
Åbo Akademi University, Department of Computer Science,
Lemminkäisenkatu 14, FIN-20520 Turku, Finland
backrj@abo.fi
Johannes Eriksson
joheriks@abo.fi
Magnus Myreen
magnus.myreen@cl.cam.ac.uk

TUCS Technical Report
No 797, December 2006

Abstract

SOCOS is a prototype tool for constructing programs and reasoning about their correctness. It supports the invariants-first programming methodology by providing a diagrammatic environment for specification, implementation and execution of procedural programs. Invariants, pre- and postconditions can be evaluated at runtime, following the Design by Contract paradigm. SOCOS can also generate correctness conditions for static program verification based on the weakest precondition semantics of statements. To verify the program 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 PVS.

Keywords: Invariant based programming, verification, SOCOS

TUCS Laboratory
Software Construction Laboratory

1 Introduction

We present here tool support for an approach to program construction, which we refer to as *invariant based programming* [6, 3]. This approach is different from other programming paradigms in that it lifts specifications and invariants to the role of first-class citizens. The programmer starts by formulating the specifications and the internal loop invariants before writing the program code itself. Expressing the invariants first has two main advantages: they are immediately available for evaluation during execution to identify invalid assumptions about the program state. Furthermore, if strong enough, invariants can be used to prove the correctness of the program. To mechanize this step, we have already developed a static checker [9], which generates verification conditions for invariant based programs and sends them to an external theorem prover. In this paper we continue on the topic by presenting the SOCOS tool, an effort to extend this checker into a fully diagrammatic programming environment.

The syntax of SOCOS programs is highly visual and based on a precise diagrammatic syntax. We use *invariant diagrams* [6], a graphical notation for describing imperative programs, to model procedures. The notation is intuitive and shares similarities with both Venn diagrams and state charts—invariants are described as nested sets and statements as transitions between sets. As a means for constructing programs, the notation differs from most programming languages in that invariants, rather than control flow blocks, serve as the primary organizing structure.

SOCOS has been developed in the Gaudi Software Factory [7], our experimental software factory for producing research software. The tool is being developed in parallel with the theory for incremental software construction with refinement diagrams [5], and the project has undergone a number of shifts in focus to accommodate the ongoing research. By using an agile development process [8] we have been able to keep the software up to date with the changing requirements.

1.1 Related Work

Invariant based programming is not a new idea. It has been considered before in a number of different forms by Dijkstra [14], Reynolds [19], Back [3] and van Emden [22]. Basic in all these approaches is that loop invariants are formulated before the program code.

Equipping software components with specifications (contracts) and assertions is the central idea of *Design by Contract* [17]. This method is supported either by add-on tools, or is integrated into the language itself (most notably in the case of Eiffel). Most languages and tools which support Design by Contract do not, however, provide static checking.

There exists a number of methods and tools for formal program verification, some with a long standing tradition. Verification techniques typically include a

combined specification and programming language, supported by software tools for verification condition generation and proof assistance. For the construction of realistic software systems, a mechanism that allows reasoning on higher levels of abstraction becomes crucial; some approaches, such as the B Method [1], provide supports for the correct refinement of abstract specifications into executable implementations. This method has had success in safety-critical and industrial applications and is a testament to the feasibility of the correct-by-construction methodology for software systems of realistic scales.

For Java and the JML specification language a host of tools have been developed for both runtime and static checking [12]. In particular, ESC/Java2 [16] enables programmers to catch common errors by sending proof conditions to an automatic theorem prover. However, it is fully automatic and thus not suitable for full formal verification. The LOOP tool [21], on the other hand, translates JML-annotated Java programs into a set of PVS theories, which can be proved interactively using the PVS proof assistant. Another tool called JACK, the Java Applet Correctness Kit [11], allows the use of both automatic and interactive provers, and is even nicely integrated into the Eclipse IDE.

1.2 Contribution

The above mentioned tools work by implementing a wp (weakest precondition) calculus for Java to automate correctness condition generation. The verification conditions generated for invariant-enriched existing languages become quite complex, and it is often difficult to understand from which part of the code a lemma has been generated. The main difference here is that rather than adding specifications and invariants to an existing language, we start with a simple notation based on nested invariants. Our belief is that an intuitive notation that enables the proof conditions to be easily understood from the program decreases the mental gap between programming and verification.

In this paper we show how proper tool support can make invariant based programming useful in practice. We have implemented the SOCOS environment for construction, testing, verification and visualization of invariant programs. Both run-time checking and static verification are available to achieve higher assurance. 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.

Since the notation requires the programmer to carefully describe the intermediate states, invariant based programs serve as documentation of design decisions and are thus more easily inspected than ordinary programs. Our preliminary experience indicates that the tool is quite useful for constructing small programs and reasoning about them: 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.

The remainder of this paper is structured as follows. In Section 2 we describe the diagrammatic notations used to implement SOCOS programs and give an overview of the SOCOS invariant diagram editor component. In Section 3 we describe how programs are compiled, executed and debugged. In Section 4 we discuss the formal semantics of SOCOS programs and the generation of proof conditions. Section 5 provides a use case of SOCOS as we demonstrate the implementation of a simple sorting program. Section 6 concludes with some general observations and a summary of on-going research.

2 Invariant Diagrams

Invariant based programs are constructed using a new diagrammatic programming notation, *nested invariant diagrams* [6], where specifications and invariants provide the main organizing structure. To illustrate the notation we will consider as an example a naive summation program which calculates the sum of the integers 0..n using simple iteration, accumulating the result in the program variable sum. An invariant diagram describing this program is given in Figure 1.

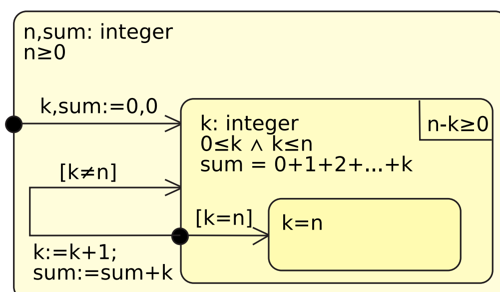


Figure 1: Summation program

Rounded boxes in the diagram represent *situation*s. A situation describes the set of program states that satisfy the invariant inside the box. When multiple predicates are written on consecutive lines they are understood to be combined with conjunction. The middle-sized box in Figure 1 thus constrains the variable k to be an integer between 0 and n, and the variable sum to have the value $0+1+2+\ldots+k$. Furthermore, nested situations inherit the predicates of outer situations, so additionally y n and sum are integers and n is greater than or equal to 0. In the smallest situation, all these predicates hold and in addition $k = n$ holds. A more deeply nested box thus strengthens the invariant.

A transition is a sequence of arrows that start in one situation and end in the same or another situation. Each arrow can be labeled with:

1. A *guard* $[g]$, where g is a Boolean expression - g is assumed when the transition is triggered.

2. A *program statement* S - S is executed when the transition is triggered. S can be a sequence of statements, but loop constructs are not allowed.

To simplify the presentation and logic of transitions, we can add intermediary choice points (forks) to branch the transition. However, joins and cycles between choice points are not allowed. Transitions thus form trees where each branch acts as an if-statement.

It should be noted that the nesting semantics of invariant diagrams that apply to situations do not apply to transitions. The program state does not have to satisfy the situation invariant while executing the transition.

In general, any situation that does not have an incoming transition can be considered an *initial state*. Conversely, we will consider situations without outgoing transitions *terminal states*.

To prove the correctness of a program described by an invariant diagram, we need to prove consistency and completeness of the transitions, and that the program cannot start an infinite loop. A transition from situation I_1 to situation I_2 using program statement S is *consistent* if and only if $I_1 \Rightarrow wp.S.I_2$ where wp is Dijkstra's weakest-precondition predicate transformer [15]. The program is *complete* if there is at least one transition with an enabled guard in each state, with the exception of terminal states. We show that a program terminates by providing a *variant*, a function which is bounded from below and which is decreased by every cycle in the diagram. In the summation example the variant, $\mathsf{n} - \mathsf{k}$, together with its lower bound ,0, is written in the upper right corner of the situation box.

The notion of correctness for invariant diagrams is further discussed in Section 4 when we consider formal verification of SOCOS programs. For a more general treatment of invariant diagrams and invariant based programming we refer to [6].

2.1 Invariant Diagrams in SOCOS

Figure 1 showed an example of a purely conceptual invariant diagram. SOCOS diagrams, which we will use in this paper, are annotated with some additional elements. Some restrictions have also been introduced for implementation and usability reasons. Figure 2 shows the equivalent summation program implemented as a SOCOS procedure.

Compared to the conceptual notation, the main differences are:

- The outer situation is a *procedure box*, which represents a procedure declaration with a procedure name, parameters and local variables. This directly reflects a design decision—since we are working with procedures, we refine the outer box into a format which includes procedure-specific attributes. Due to restrictions in our implementation local variables have procedure scope and it is presently not possible to introduce new variables in nested situations, as was done in Figure 1.

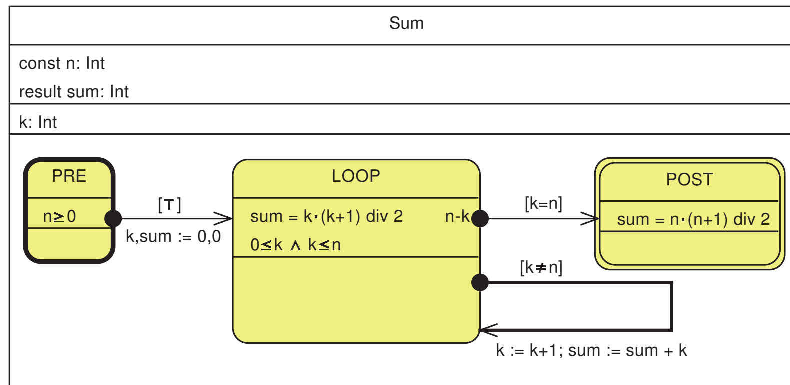


Figure 2: Summation program, SOCOS syntax

- Each situation is labeled with a descriptive name, such as LOOP for a recurring situation. The label is used as a general identifier so that the situation can be referred to in error reports and generated proof conditions.

- For simplicity the variant is assumed to be a function to natural numbers and to be bounded from below by zero, so in SOCOS we write just $\mathsf{n}-\mathsf{k}$ since the bound is implicit.

- In each cycle of transitions at least one transition has to be marked as decreasing the variant. Such transitions are rendered in the diagram as thicker arrows, indicating that the variant *must decrease* when this transition is executed. At other transitions in the cycle, the requirement is only that the variant *must not increase*. This is further discussed in Section 4.

- We provide an initial and a terminal situation representing the entry and exit point of the procedure, respectively. These situations constitute the contract (pre- and postcondition) of the procedure. The precondition situation is called PRE and is additionally marked with a thick outline, while the postcondition is called POST and is marked with a double outline. Note that in the example we do not nest POST within LOOP, but instead we repeat part of the invariant in the postcondition. Since the contract constitutes the *external interface* to other procedures, it should constrain local variables (such as k in this case).

- SOCOS does not support a general summation operator, in this specific example we have used the direct formula $\frac{k(k+1)}{2}$ for expressing the sum $0+1+2+...+k$.

SOCOS implements a minimal set of executable and non-executable program statements according to the syntax:

$$
\begin{aligned}
S ::= \;& \textsf{magic} \mid \textsf{abort} \mid \\
& x_1,...,x_m := v_1,...,v_m \mid \\
& S_0;S_1 \mid \\
& [b] \mid \{b\} \mid \\
& P(a_1,...,a_n)
\end{aligned}
$$

Here magic is the miraculous statement—it satisfies every postcondition. abort represents the aborting program, which never terminates. The assignment statement assigns a list of values $v_1,...,v_m$ to a list of variables $x_1,...,x_m$. The ; operator represents sequential composition of two statements S_0 and S_1. An assume statement $[b]$ means that we can assume the predicate b at that point in the transition, while an assert statement $\{b\}$ tells us that we have to show that b holds at a in the transition. A procedure call $P(a_1,...,a_n)$ stands for a call to procedure P with the actual parameters $a_1,...,a_n$. The type of an actual parameter a_i depends on how the parameter type is qualified: for unqualified and const parameters, an expression is accepted. For result and valres parameters, the actual must be a simple variable. The formal weakest precondition semantics of these statements are the standard ones [10].

2.2 Diagram Editor

Procedures are constructed in the SOCOS invariant diagram editor. Any number of diagrams can be open simultaneously, and there are no sequential constraints imposed on editing operations: any part of the program can be changed at any time. A screen shot of the SOCOS invariant diagram editor can be seen in Figure 3. The highlighted tab below the main toolbar indicates that an invariant diagram is currently being edited. To the left of the diagram is an outline editor for browsing model elements, and the bottom pane holds the property editors and various communication windows.

The SOCOS diagram editor is implemented on top of another project developed in the Gaudi software factory, the Coral *modeling framework* [2]. Coral is a metamodel-independent toolkit which can easily be extended with custom metamodels. A metamodel is a syntax description for Coral *models*: it defines the available model elements and their relationships. Coral features a powerful programmatic interface which makes it possible to build Python programs that use a Coral model as the main data repository, and provides features such as transaction management and serialization into the XMI format, an OMG standard [18].

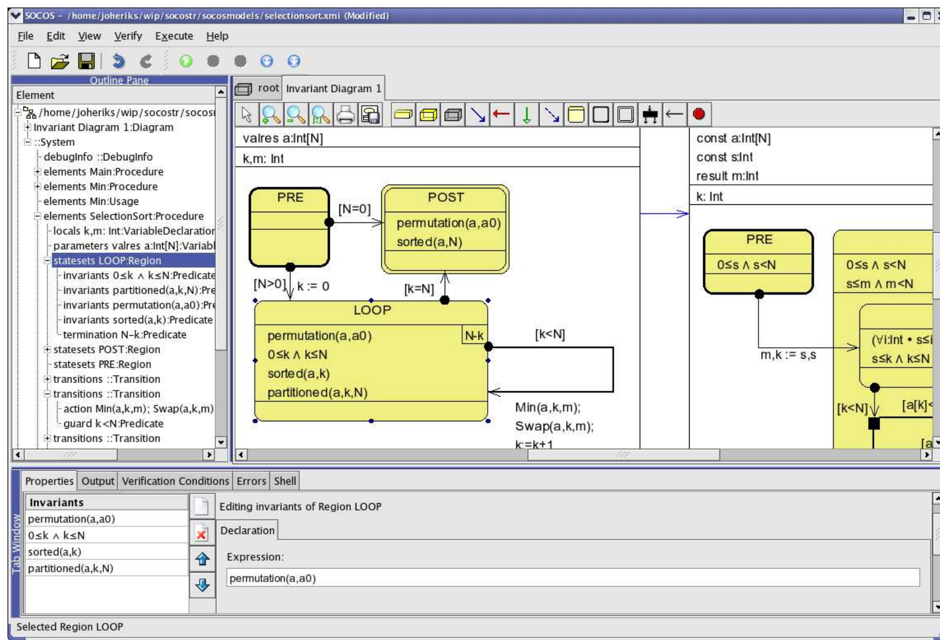


Figure 3: Invariant diagram editor of SOCOS

3 Run-time Checking of Invariant Diagrams

3.1 Compilation

A SOCOS invariant diagram is executed by compiling it into a Python program which is executed by the standard Python interpreter. We selected a very simple approach for code generation; the generated program is effectively a goto-program. Each situation is represented by a method. The body of a situation's method executes the transition statements and returns a reference to the next method to be executed as well as an updated environment (a mapping from variable names to values). The main loop of the program is simply:

```
while s:
    s,env = s(env)
```

where `s` is the currently executing situation and `env` is the environment.

If run-time checking is enabled, invariants and assertions are evaluated during execution of a situation's method. For situations that are part of a cycle, the variant is compared to its lower bound, as well as to its value in the previous cycle to ensure that it is decreasing. In the case that any of these checks evaluate to false, an exception is raised and the execution halts.

While SOCOS automatically evaluates only a pre-defined subset of all expressible invariants (those composed of arithmetic expressions and Boolean expressions containing only bounded quantifiers), it is possible to extend the dynamic evaluation capabilities for special cases by writing a side-effect free Python

script to perform the evaluation. This requires adding a translation rule (explained in the next subsection) which translates the expression to a Python function call; the function is then executed each time SOCOS needs to evaluate an invariant containing the expression.

3.2 Translating Conditions to Python

SOCOS uses a set of translation rules to produce an executable Python program. In order to make the compilation easily extensible we provide the user with the capability to define new translations. The translation of a mathematical expression is done through simple rewrite rules. The user may define new translation rules. Here are a few of the predefined translation rules:

`rule Py00[group=python]` $python(\top) \equiv$ `True`.

`rule Py03[group=python]` $python(a \wedge b) \equiv python(a)$ `and` $python(b)$.

`rule Py13[group=python]` $python(m+n) \equiv python(m) + python(n)$.

All translation rules are similar in shape. They push a translation function (*python* above) through the expression to be translated. The translation of an expression e is performed by repeatedly applying the rewrite rules to the expression $python(e)$ until the function symbol *python* does not occur in the resulting expression. Compilation succeeds if all expressions of the program are translated successfully.

3.3 Debugging

SOCOS provides a graphical debugger for tracking the execution of invariant diagrams. All SOCOS programs define a *main procedure*, which acts as the program entry point. A program can be run continuously or stepped through transition by transition. During execution the current program state, consisting of the procedure call stack, the values of allocated variables and the current situation, can be inspected. It is possible to set *breakpoint*s to halt the execution in specific situations.

Program execution is visualized by highlighting diagram elements in the editor. Active procedures, i.e. procedures on the call stack, as well as the current situation and the currently executing transition are highlighted. The values of local variables for each stack frame are displayed in a *call stack view*.

Invariants are evaluated at run-time and are highlighted in red, green or gray depending on the result: for invariants that evaluate to true the highlight color is green, for invariants that evaluate to false it is red, and if SOCOS is unable to evaluate the invariant it is gray. The termination condition is also compared to its lower bound and highlighted in the same way. The program execution is halted whenever an invariant evaluates to false.

4 Proving Correctness of Invariant Diagrams

The SOCOS environment supports interactive and non-interactive verification of program diagrams. It generates the verification conditions and sends them to proof tools. At the time of writing two proof tools are supported: Simplify [13] and PVS [20]. 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.

4.1 Verification Condition Generation

SOCOS generates verification conditions using MathEdit [9]. Three types of verification conditions are generated: consistency, completeness and termination conditions. All of these use the weakest precondition semantics as their basis [15]. 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.

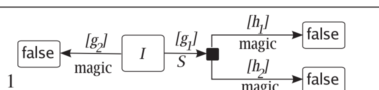
Consistency:

> 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.[1]

---

[1] 

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

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.[2]

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.[3]

The interested reader is referred to [6] for a more detailed presentation of the notion of correctness of invariant diagrams.

4.2 Interaction with External Tools

SOCOS communicates through MathEdit with external proofs tools. Interfaces to PVS and Simplify are currently implemented in MathEdit. The interface to Simplify is from the users 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 [9].

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. A non-interactive mode for using PVS is also supplied. In the non-interactive mode PVS is run in batch mode behind the scenes. PVS applies a modified version of the `grind` tactic to all verification conditions and reports success or failure for each verification condition. The output is shown to the user of SOCOS.

[2]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.

[3]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 $v \leq v_0$.

4.3 Translation of Verification Conditions

The verification conditions are translated using rewrite rules similar to those used for compilation into Python code. The user may define new translation rules for translation into PVS and Simplify.

The verification conditions sent to Simplify and PVS differ in more than just syntax. PVS has a stronger input language, which among other things supports partial functions well. Simplify's input language is untyped, which means that some expressions require side conditions to ensure that they are well defined, for example k div m requires the side condition $m \neq 0$. We cannot guarantee that the generated side conditions are strong enough for user defined operands. Hence we recommend that Simplify is used for spotting bugs early in the design and PVS is used for formal verification of the final components.

Please note that care must be taken while writing new translation rules for the verification conditions. Mistakes in the translation rules can jeopardize the validity of the correctness proof.

5 Example: Sorting

In this section we will demonstrate how a procedure specification, consisting of a procedure interface and given pre- and postconditions, is implemented in SOCOS. We choose a simple sorting algorithm as our case study. The focus will be mainly on the tool and how invariant based programming is supported in practice—for a more detailed treatment of the methodology itself, we refer to [6].

5.1 Specification

We start by introducing a procedure specification consisting of a signature, i.e. the names and types of parameters, and a contract (pre- and postcondition). A standard sorting specification is shown in Figure 4. The procedure accepts one parameter, an integer array a with N elements. Indexes are 0-based; the index of the first element is 0 and the index of the last element is $\mathsf{N}-1$. The valres keyword indicates that a is a value-result parameter. Because a is updated by the sorting routine, but should remain a permutation of the original array, the postcondition relates the old and new values of a to each other by the permutation predicate. We use the convention of appending 0 to the parameter name to refer to the original value of the parameter. The sorted predicate simply means that each element is less than or equal to its successor in the array.

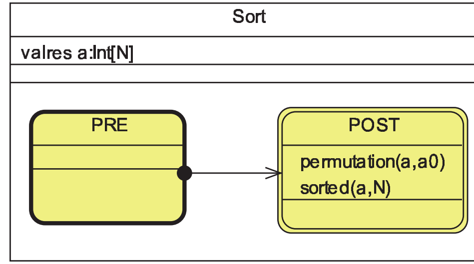


Figure 4: A specification of a sorting procedure

SOCOS (non-empty) integer arrays are modeled as functions from the interval $[0, \mathsf{N})$ to Int, where N is a positive natural number. Array access of element i is defined as function application: $a[i] = a.i$. We can then define the predicates sorted and permutation as follows:

$$\mathsf{sorted}(a,n) \mathrel{\hat{=}} (\forall i : \mathsf{Int} \bullet 0 < i \land i < n \Rightarrow a[i-1] \leq a[i])$$
$$\mathsf{permutation}(a,b) \mathrel{\hat{=}} (\exists f \bullet (\mathsf{bijective}.f) \land f.a = b)$$

Some invariants that are guaranteed by the system are implicit. The precondition as specified is empty, however, during verification condition generation the additional assumption $\mathsf{a} = \mathsf{a0}$ is added automatically. Furthermore, the type invariant for nonempty arrays allows us to assume $\mathsf{N} > 0$ in every situation in Sort.

Given this specification, the next task is to provide an executable program which takes the program state from PRE to POST.

5.2 Implementation

For simplicity and brevity we will implement a very basic sorting algorithm, selection sort, which performs in-place sorting in $O(n^2)$ time. Our implementation SelectionSort can be seen in Figure 5; two helper procedures, Min and Swap, are given in Figure 6. Min finds the index of the smallest element in the subarray $\mathsf{a}[s..\mathsf{N})$ and returns its index, while Swap exchanges the two elements at indexes m and n in the array a.

SelectionSort sorts an array by partitioning it into two portions, one unsorted followed by one sorted. Each iteration of the main loop exchanges the largest element from the unsorted portion of the array with the element just before the beginning of the already sorted portion, until no elements are left in the unsorted portion.

5.3 Testing the Implementation

We can gain an understanding of how selection sort works by implementing a simple test case and examining the transitions between program states by single-stepping through the call to SelectionSort in the SOCOS debugger. Figure 7 shows such a debugging session.

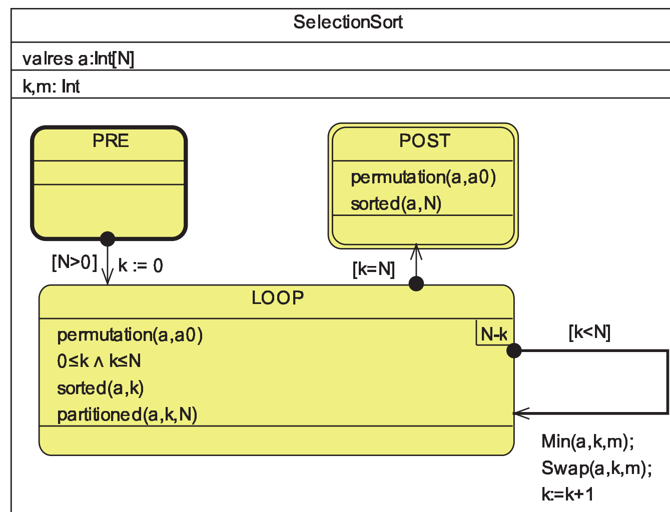


Figure 5: Selection sort

In Figure 7 the current procedure and situation is shown with a blue outline. The BODY situation has been marked as a *breakpoint* (indicated by a red dot). This causes the execution flow to temporarily halt at this point, and the current program state is shown in the pane to the right. Both the original value of the array prior to the call, a0, and the partially sorted array, a, are shown. Furthermore, invariants are evaluated and color-coded. In the absence of a breakpoint, execution also halts whenever an invariant evaluates to false.

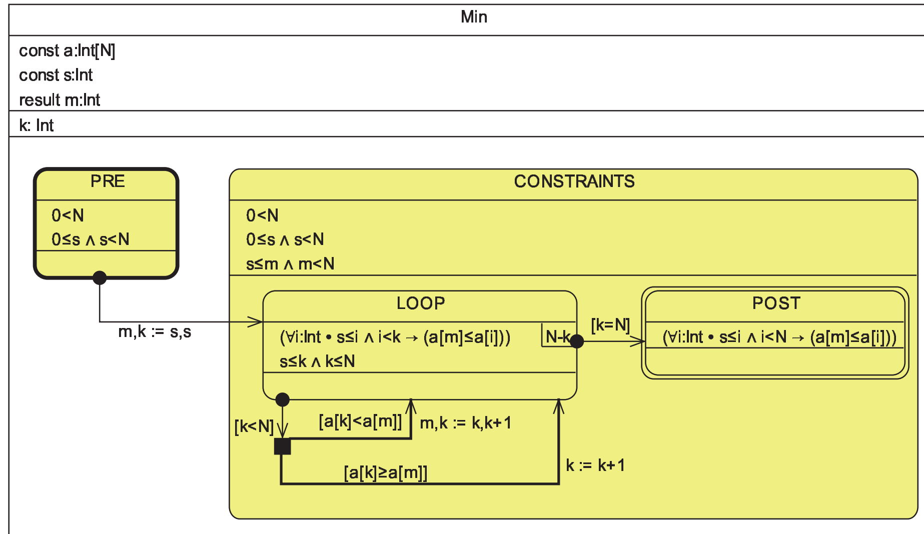
SOCOS can translate most simple invariants automatically to Python run-time checks based on a number of built-in rules. However, permutation is a predicate that is not automatically translatable to Python, and by default such an invariant will be colored gray during execution to indicate that it was not evaluated. If we want to enable runtime checking of permutation, we can add a Python function which checks whether the array `xs` is a permutation of the array `ys`:

```
def permutation( xs, ys ):
    xs,ys = list(xs),list(ys)
    xs.sort()
    ys.sort()
    return xs==ys
```

In addition to the code snippet we also provide a rewrite rule to make SOCOS generate a call to this function whenever it encounters permutation during evaluation of an invariant.

5.4 Verifying the Implementation

While dynamic checking of invariants is valuable in that it catches many common programming errors, its efficiency is highly dependent on good test cases.



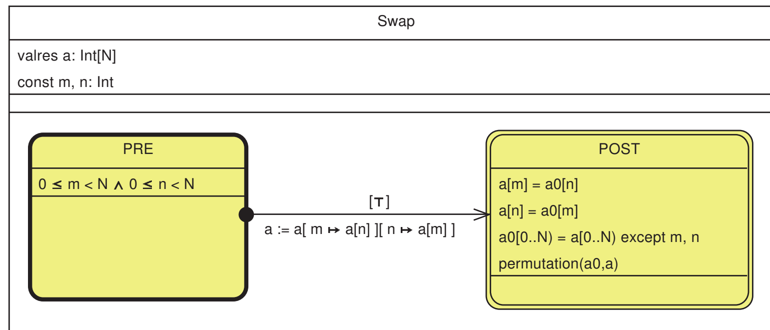


Figure 6: Utility procedures Min and Swap

Since we have put much effort into writing down the invariants, we can go one step further and attempt *formal verification*. In this mode, SOCOS generates verification conditions for consistency, completeness and termination as described in the previous section. The automatic correctness checking command, Verify ▷ Check Correctness (Simplify), employs Simplify to attempt automatic discharging of verification conditions. If we run this command on the example, SOCOS will tell us that Simplify was able to discharge 99.7 percent of the conditions (Figure 8). While all conditions for SelectionSort and Max are automatically discharged, problems occur due to the use of permutation in Swap.

SOCOS has pinpointed a specific verification condition for us that we need to check. However, since permutation is a higher-order property, we can not give a definition of permutation that Simplify can use. In this situation we have two options—we can temporarily get rid of the error by adding assumptions: in the case of Swap we would add an assumption statement, [permutation(a, a0)], following the assignment statement in the transition from PRE to POST if we believe that a[m ↦ a[n]][n ↦ a[m]] is indeed a permutation of a. This could correspond

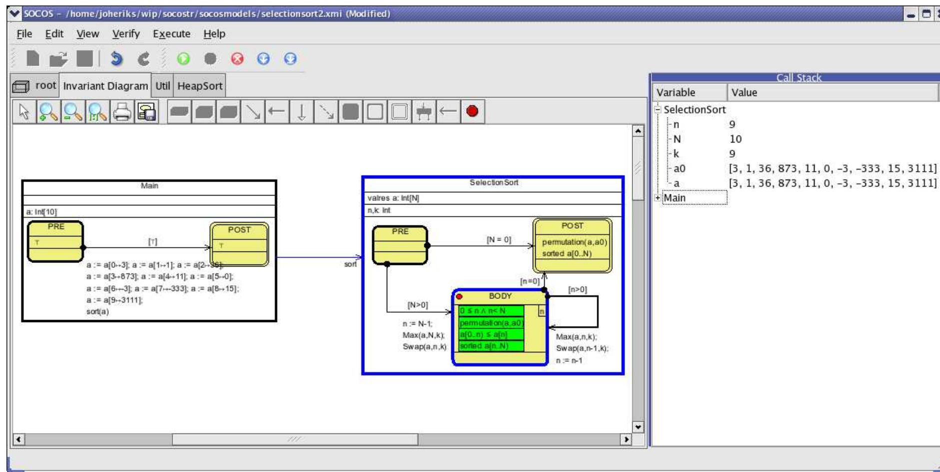


Figure 7: Stepping through a test case of selection sort

> Verification initiated for SelectionSort, Swap and Min.
> 99.7% of the verifications were proved automatically.
> Condition: POST (Swap)
> Assumptions:
> $0 < \mathsf{N}$
> $0 \leq \mathsf{m}$
> $\mathsf{m} < \mathsf{N}$
> $0 \leq \mathsf{n}$
> $\mathsf{n} < \mathsf{N}$
> $\mathsf{a0} = \mathsf{a}$
> Imply:
> $\mathsf{permutation}(\mathsf{a0}, \mathsf{a}[\mathsf{m} \mapsto \mathsf{a}[\mathsf{n}]][\mathsf{n} \mapsto \mathsf{a}[\mathsf{m}]])$

Figure 8: Remaining condition for Swap

to simple “belief”. During initial development of a procedure it is 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 conditions interactively in PVS. The prover to be used (PVS or Simplify) can be chosen on the level of single transitions, with Simplify being the default. In this case the PVS language is expressive enough to allow us to provide a higher-order definition of permutation:

```
index:  type = {i:nat|i<N}
permutation( a:index, b:index ):  bool =
    exists(f:(bijective[index,index])):  a = b o f
```

In PVS, SOCOS arrays are represented by functions from indexes (natural numbers below N) to values (integers). An array is a permutation of another array if

there exists a one-to-one correspondence (a bijection) 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 which is automatically loaded when PVS verification is initiated.[4] In addition the background theory includes previously proved lemmas about arrays and permutations to facilitate new proofs.

Given the PVS definition of permutation it is easy to prove the remaining conditions in PVS by providing a bijection and applying built-in lemmas from the PVS prelude; however, to conserve space we have not included the actual proofs here.

6 Conclusion and Future Work

We have here presented SOCOS, a tool to support diagrammatic invariant based programming. SOCOS can currently be used to develop procedural programs. In the early phases of development simple errors are found by testing. At a later stage of development the programmer can prove, using formal reasoning, that the program is *error-free*. All but the most trivial programs generate a large number of lemmas to be proved. The tool translates these lemmas into the PVS and Simplify input languages. Most of the generated lemmas are rather trivial and automatically discharged by Simplify or the PVS `grind` strategy. For more difficult lemmas, the proofs can be completed interactively in PVS.

The SOCOS system is currently in early stages and the framework is still being worked on. Most importantly, the issue of applicability and scalability should be addressed. We have so far limited our focus to programming "in the small", which is indeed the main target for invariant based programming. However, to make SOCOS suitable for systems of realistic scales, support for classes and other software decomposition mechanisms becomes critical. As a first step we are currently adding support for object-orientation in SOCOS. Introducing objects makes the verification problem more difficult and significantly complicates reasoning; the challenge here is to equip a formalism for classes and objects with an intuitive diagrammatic notation, and provide means for reasoning in terms of these diagrams. Refinement diagrams [5], a diagrammatic representation of lattice theory, will provide the basis for the SOCOS class notation.

Another issue of key importance is performance; SOCOS is currently rather slow—generating and checking (with Simplify) the proof conditions of the example in Section 5 takes several seconds on a modern PC.[5] Replaying PVS proofs is even slower. This limits the use of SOCOS to simple programs. While our implementation is in some cases sub-optimal, it is inevitable that automated ver-

[4] A (much simpler) background theory is also sent to Simplify; part of this theory is that permutation is reflexive—this explains why Simplify was able to prove the transition between PRE and LOOP in SelectionSort.

[5] 2.8 GHz Intel Pentium 4 with 1 GB of random access memory.

ification of correctness conditions is computationally taxing. We are currently working on *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, much like how many modern IDEs (Integrated Development Environments) semantically analyze programs as the user is typing.

We are carrying out a number of case studies in invariant based programming. These case studies are conducted on two different levels: firstly, we are building a larger example of higher complexity with many interacting components (a string processing library); secondly, we will teach invariant based programming to a group of undergraduate students, using SOCOS as the programming tool. The objective of the first experiment is to evaluate the scalability of the method and its feasibility in construction larger programs. In the second experiment, we explore the educational merits of invariant based programming—it is our belief that the direct connection to logic, together with the use of diagrams and visualization, will make it a useful method for teaching the use of formal methods in programming.

SOCOS currently supports basic program proof management, but does not provide adequate facilities for managing program proofs in a way that accommodates continuous change. PVS proofs must be managed by hand by the user, and if a procedure is changed, however slightly, all proofs must be replayed. It would be desirable if the tool kept track of dependencies between program elements, and in the event of a change, only replayed proofs of possibly invalidated transitions. A nice feature of interactive provers like PVS is that advanced proof strategies do not mention specific terms but rather work on the high-level structure of a formula. So, in the case of slight changes, when an existing proof is replayed chances are good that it will be able to prove the new correctness condition as well.

Finally, there is a need for a way to make incremental software extensions and reason about their correctness. Stepwise Feature Introduction [4], a sound layered extension mechanism based on superposition refinement, is intended to be the main method by which a SOCOS program is extended with new functionality.

References

[1] Jean-Raymond Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sorensen. The B-method (software development). In W. J. Prehn, S.; Toetenel, editor, *VDM 91. Formal Software Development Methods. 4th International Symposium of VDM Europe Proceedings.*, volume 2, pages 398–405. BP Res., Sunbury Res. Centre, Sunbury-on-Thames, UK, Springer-Verlag, Berlin, Germany, October 1991.

[2] Marcus Alanen and Ivan Porres. The Coral Modelling Framework. In Kai Koskimies, Ludwik Kuzniarz, Johan Lilius, and Ivan Porres, editors, *Proceedings of the 2nd Nordic Workshop on the Unified Modeling Language*

NWUML'2004, number 35 in General Publications. Turku Centre for Computer Science, Jul 2004.

[3] Ralph-Johan 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.

[4] Ralph-Johan Back. Software construction by stepwise feature introduction. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 162–183. Springer-Verlag, 2002.

[5] Ralph-Johan Back. Incremental software construction with refinement diagrams. In Harel 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.

[6] Ralph-Johan Back. Invariant based programming. In Susanna Donatelli and P. S. Thiagarajan, editors, *ICATPN*, volume 4024 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2006.

[7] Ralph-Johan Back, Luka Milovanov, and Ivan Porres. Software development and experimentation in an academic environment: The Gaudi experience. In *Proceedings of the 6th International Conference on Product Focused Software Process Improvement, PROFES 2005*, Oulu, Finland, Jun 2005.

[8] Ralph-Johan Back, Luka Milovanov, Ivan Porres, and Viorel Preoteasa. XP as a framework for practical software engineering experiments. In *Proceedings of the Third International Conference on eXtreme Programming and Agile Processes in Software Engineering - XP2002*, May 2002.

[9] Ralph-Johan Back and Magnus Myreen. Tool support for invariant based programming. In *the 12th Asia-Pacific Software Engineering Conference*, Taipei, Taiwan, December 2005.

[10] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Graduate Texts in Computer Science.

[11] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods: International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.

[12] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.

[13] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[14] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, pages 8:174–186, 1968.

[15] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[16] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie 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.

[17] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[18] OMG. XML Metadata Interchange (XMI) Specification, version 2.0, May 2003. Available at http://www.omg.org/.

[19] J. C. Reynolds. Programming with transition diagrams. In D. Gries, editor, *Programming Methodology*. Springer Verlag, Berlin, 1978.

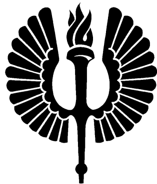
[20] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 411–414, New Brunswick, NJ, USA, July/August 1996. Springer-Verlag.

[21] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. *Lecture Notes in Computer Science*, 2031:299+, 2001.

[22] M. H. van Emden. Programming with verification conditions. *IEEE Transactions on Software Engineering*, pages SE–5, 1979.

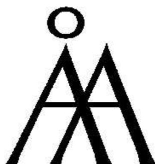
Turku Centre *for* Computer Science

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi



University of Turku

- Department of Information Technology
- Department of Mathematics



Åbo Akademi University

- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration

- Institute of Information Systems Sciences

ISBN 952-12-1835-5
ISSN 1239-1891