Testing and Verifying Invariant Based Programs in the SOCOS Environment

Ralph-Johan Back[1], Johannes Eriksson[1] and Magnus Myreen[2]

[1] Åbo Akademi University, Department of Information Technologies
Turku, FI-20520, Finland
{backrj,joheriks}@abo.fi
[2] University of Cambridge, Computer Laboratory
Cambridge CB3 0FD, UK
magnus.myreen@cl.cam.ac.uk

Abstract. SOCOS is a prototype tool for constructing programs and reasoning about their correctness. It supports the invariant based programming methodology by providing a diagrammatic environment for specification, implementation, verification and execution of procedural programs. Invariants and contracts (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 the PVS theorem prover.

1 Introduction

This paper presents tool support for an approach to program construction, which we refer to as *invariant based programming* [1,2]. This approach differs from most conventional 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: firstly, they are immediately available for evaluation during execution to identify invalid assumptions about the program state. Secondly, if strong enough, invariants can be used to prove the correctness of the program. To mechanize the second step, we have previously developed a static checker [3], 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* [1], 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 [4], 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 [6] we have been able to keep the software up to date with the changing requirements.

1.1 Related Work

Invariant based programming originates in Dijkstra's ideas of constructing the program and its proof hand in hand [7]. Invariant based programming (Reynolds [8], Back [9,2] and van Emden [10]) takes this approach one step further, so that program invariants are determined before the program code or even the control structures to be used have been determined.

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 method for reasoning on higher levels of abstraction becomes crucial; some approaches, such as the B Method [11], support correct refinement of abstract specifications into executable implementations. This method has had success in safety-critical and industrial applications and shows the applicability of formal methods to software systems of realistic scales.

Equipping software components with specifications (contracts) and assertions is the central idea of *Design by Contract* [12]. This method is supported either by add-on tools or, as in the case of Eiffel, is integrated into the language. Most languages and tools which support Design by Contract do not, however, provide static correctness checking.

A host of tools have been developed for Java and the JML specification language, for both runtime and static correctness checking [13]. In particular, ESC/Java2 [14] enables programmers to catch common errors by sending verification conditions to an automatic theorem prover. However, it is fully automatic and thus not powerful enough for full formal verification. The LOOP tool [15], 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 [16], allows the use of both automatic and interactive provers from an Eclipse front end.

1.2 Contribution

Many tools for verifying programs work by implementing a weakest precondition calculus for an existing language. Due to complex language semantics, the

proof obligations generated for invariant-enriched existing languages often become quite elaborate. This can make it difficult to know which part of the code a condition was generated from, and to see the relationship between code and proof obligation. Rather than adding specifications and invariants to an existing language, we start with a simpler notation for programs and their proofs, *invariant diagrams* with nested invariants. Our belief is that an intuitive notation where the proof conditions are easily seen from the program description decreases the mental gap between programming and verification. Since the notation requires the programmer to carefully describe the intermediate situations (invariants), invariant based programs provide as a side effect automatic documentation of the design decisions made when constructing the program, and are thus easier to inspect than ordinary programs.

We describe here a tool to support invariant based programming, the *SOCOS environment,* which supports the construction, testing, verification and visualization of invariant based programs by providing an integrated editor, debugger and theorem prover interface. An invariant based program is developed in SOCOS in an incremental manner, so that we continually check that each increment preserves the correctness of the program built thus far. Both light-weight and heavy-weight techniques are used to verify the correctness of a program extensions. In the early phases of development, exercising the behavior of the program with test cases is an efficient way to detect invariant violations. To achieve higher assurance, the programmer can perform automated static correctness analysis to prove that some part of each invariant holds for all input. Total correctness is achieved by proving that remaining parts of the invariants hold, using an interactive proof assistant. Our preliminary experience indicates that the tool is quite useful for constructing small programs and reasoning about them. It removes the tedium of checking trivial verification conditions, it automates the run-time checking of contracts and invariants, and 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. 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* [1], 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.

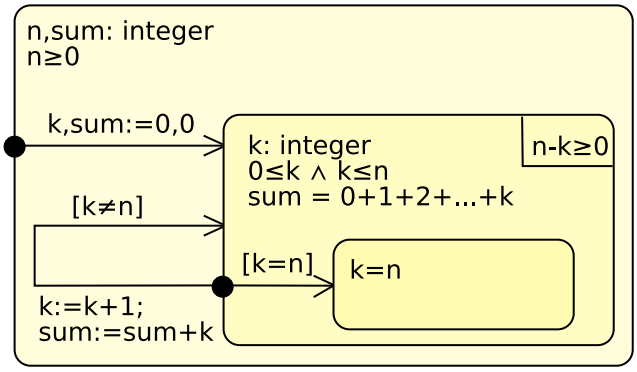


Fig. 1. Summation program

Rounded boxes in the diagram represent *situation*s. A situation describes the set of program states that satisfy the predicate inside the box. When multiple predicates are written on consecutive lines, all are assumed to hold. Furthermore, nested situations inherit the predicates of enclosing situations. Inside the largest box in Figure 1, variables n and sum are of type integer and n is greater than or equal to 0. Due to nesting this is also true in the middle-sized box, and additionally the variable k is an integer between 0 and n, and the variable sum has the value $0 + 1 + 2 + ... + k$. In the smallest situation, all these predicates hold and in addition $k = n$.

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 intermediate choice points (forks) to branch the transition. However, joins and cycles between choice points are not allowed. The transitions described by the tree are all the paths in the tree, from the start situation to some end situation. A choice point in the tree can be seen as a conditional branching statement.

It should be noted that the nesting semantics of invariant diagrams that apply to situations do not apply to transitions. The program state is not required to satisfy any situation while executing a transition, even if the arrow itself is drawn inside a situation box.

In general, any situation that does not have an incoming transition is considered an *initial situation*. Conversely, we will consider a situation without outgoing transitions a *terminal situation*.

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 [17]. The program is *complete* if there is at least one enabled transition in each state, with the exception of terminal situations. 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 is indicated in the upper right corner of the situation box.

The notion of correctness for invariant diagrams is further discussed in Section 4 where we consider formal verification of SOCOS programs. For a more general treatment of invariant diagrams and invariant based programming we refer to [1].

2.1 Invariant Diagrams in SOCOS

Figure 1 shows 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 to simplify the implementation. Figure 2 shows the equivalent summation program implemented as a SOCOS procedure.

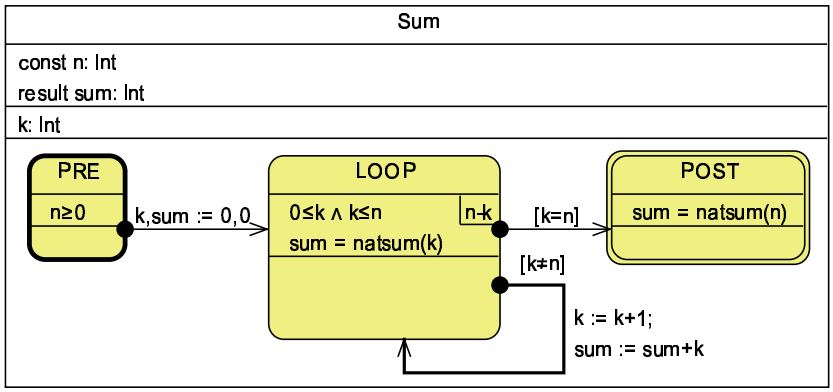


Fig. 2. Summation program, SOCOS syntax

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.

- Each situation is labeled with a descriptive name, such as LOOP for a recurring situation. The name is used as a general situation identifier in error reports and generated proof conditions.
- For simplicity the variant is assumed to be a function to natural numbers, so in SOCOS we write just n − k since the lower bound (zero) is implicit.
- In general, it is only necessary for one transition in a cycle to *strictly decrease* the variant. Such transitions are rendered in the diagram as thicker arrows. 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 of the procedure. The precondition situation is called PRE and is additionally marked with a thick outline, while the postcondition is called POST and marked with a double outline. Note that in the example POST is not nested within LOOP, but instead part of the invariant is repeated. Since the contract constitutes an external interface to other procedures, it must not constrain the local variable k.
- Local variables have procedure scope and it is presently not possible to introduce new variables in nested situations.
- SOCOS supports declaration of global predicates and functions. In this case we assume that natsum has been defined to give the sum from 0 up to its argument, e.g. based on a recursive definition or using the direct formula $\frac{k(k+1)}{2}$.

The procedure is the basic unit of decomposition in SOCOS. A procedure consists of two parts: an externally visible *interface*, and a hidden *implementation*. The interface can further be divided into a *signature* and a *contract*. A signature is a list of formal input and output parameters and describes the name, type and qualifier of each parameter. Five primitive types (natural numbers, integers, Booleans, characters, strings) along with one composite type (array) are presently supported; the four available parameter qualifiers are listed in Table 1. The contract defines the obligations and benefits of the procedure as a pre- and postcondition pair. Recursive procedures are supported. As in the case of cyclic transitions, all procedures in a recursion cycle must provide a common variant as part of their interfaces.

The implementation defines the structure of the internal computation that establishes the contract. It consists of a transition diagram from the precondition to the postcondition. Each transition can be labeled with a statement according to the syntax:

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

Here magic is the miraculous statement, which satisfies every postcondition. abort represents the aborting program, which never terminates. The assignment

Table 1. Parameter qualifiers

Qualifier	Role	Keyword	Description
Value	In	-	Can be read and updated by the implementation, but updates are not reflected back to the caller
Constant	In	const	Can only be read by the implementation
Value-result	In, out	valres	Can be read as well as updated by the implementation. Updates are not reflected back to caller until the procedure returns
Result	Out	result	Like value-result, but may not occur in preconditions

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 that point 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 [18].

2.2 Diagram Editor

Programs are constructed in the SOCOS invariant diagram editor. A program is represented by a collection of procedure diagrams. A screen shot of the diagram editor is shown in Figure 3. The highlighted tab below the main toolbar indicates that an invariant diagram is currently being edited. On the left is an outline editor for browsing model elements, and the bottom pane holds 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* [19]. Coral is a metamodel-independent toolkit which can easily be extended to custom graphical notations.

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

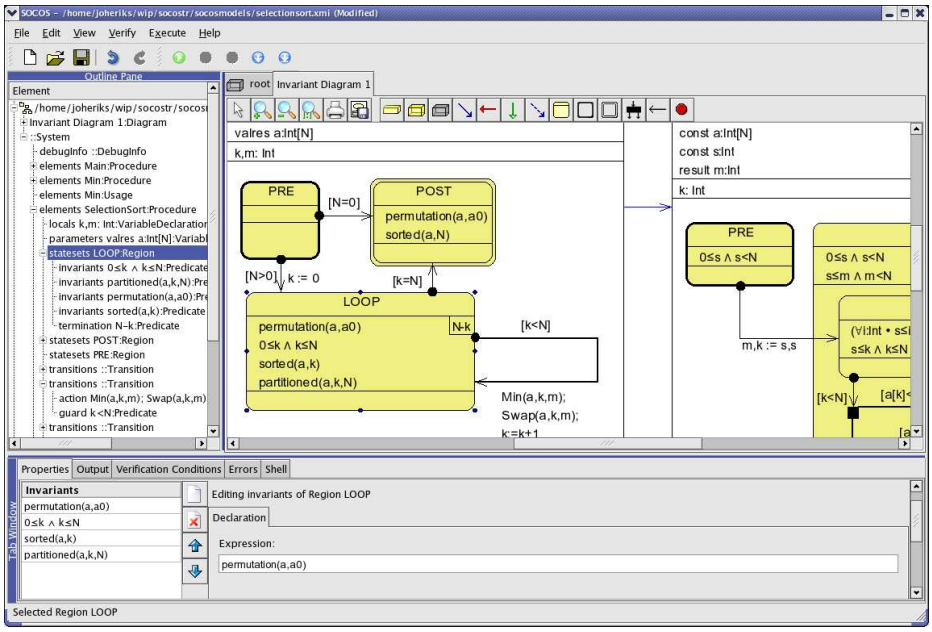


Fig. 3. Invariant diagram editor of SOCOS

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. If any of these checks evaluate to false, an exception is raised. While SOCOS automatically evaluates only a pre-defined subset of all expressible invariants (namely arithmetic expressions and Boolean expressions containing only bounded quantifiers), it is possible to extend the dynamic evaluation capabilities for special cases by adding a side-effect free Python script to perform the evaluation.

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(⊤) ≡ True.
rule Py03[group=python] python(a ∧ b) ≡ python(a) and python(b).
rule Py13[group=python] python(m + n) ≡ 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. 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 *breakpoints* 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 in 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 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 [20] and PVS [21]. 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 [3]. Three types of verification conditions are generated: consistency, completeness and termination conditions. All of these use the weakest precondition semantics as their basis [17]. 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 non-terminal situation is complete. A situation I is complete iff

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

where S^* is the statement that we get from the transition tree from I when each branching with branches $[b_1]; S_1, \ldots, [b_k]; S_k$ is treated as an $\mathsf{if}\ b \rightarrow_1 S_1 \,[]\, \ldots \,[]\, b \rightarrow_k S_k\ \mathsf{fi}$ statement and each leaf statement is replaced with magic [3] [4].

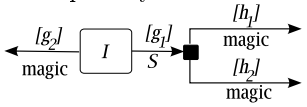
Termination:

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.[5]

---

[3] A single guard statement $[b]; S_1$ without an alternative branch has also to be written as an $\mathsf{if} \ldots \mathsf{fi}$ statement, $\mathsf{if}\ b \rightarrow S_1\ \mathsf{fi}$.

[4] We disregard the statements at the leaves by replacing them with miracles. A simple example may be useful here:



The completeness condition for 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: $I \Rightarrow (g_1 \Rightarrow wp.S.(h_1 \vee h_2)) \wedge (g_1 \vee g_2)$

[5] 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.

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.[6]

The interested reader is referred to [1] 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 [3].

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.

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

[6] 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 $v \leq v_0$.

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 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 is 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 [1].

5.1 Specification

We start by introducing a procedure specification consisting of a signature and a contract. 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 $N-1$. 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 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 says that each element is less than or equal to its successor in the array.

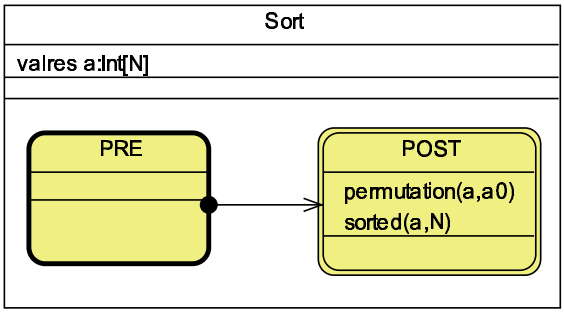


Fig. 4. A specification of a sorting procedure

A SOCOS integer array is modeled as a function from the interval $[0, N)$ to Int, where N is the size of the array. N is assumed to be a positive natural

number. Access to an array element at index i is defined as function application: $a[i] = a.i$. We then define the predicates sorted and permutation as follows:

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

Some invariants are guaranteed by the system and thus implicit. The precondition as given above 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 transforms any state in PRE to a state in POST.

5.2 Implementation

For brevity we implement a simple sorting algorithm, selection sort, which performs in-place sorting in $O(n^2)$ time. 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 sorted portion by one. The loop terminates when no elements are left in the unsorted portion.

The implementation SelectionSort can be seen in Figure 5; the two helper procedures, Min and Swap, are given in Figure 6. Min finds 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.

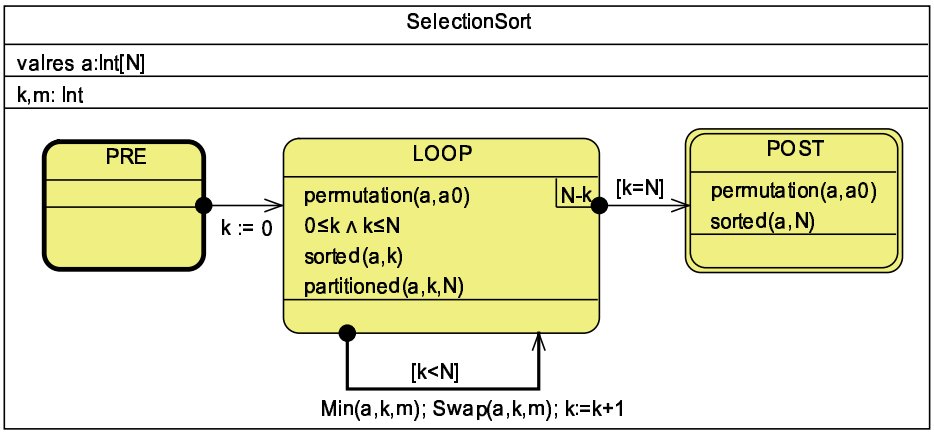


Fig. 5. Selection sort

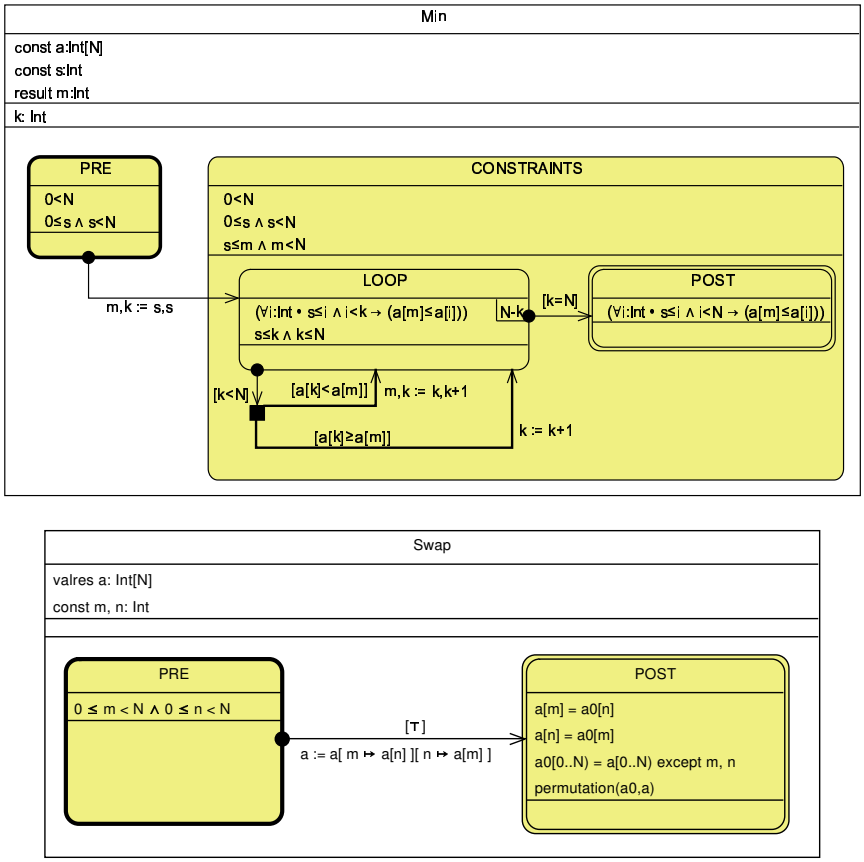


Fig. 6. Utility procedures Min and Swap

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.

The current situation is highlighted with a blue outline. The LOOP situation has been marked as a breakpoint (indicated by a red dot in the upper left corner). 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 simple invariants automatically to Python based on built-in rules. However, permutation is not automatically translatable, but we

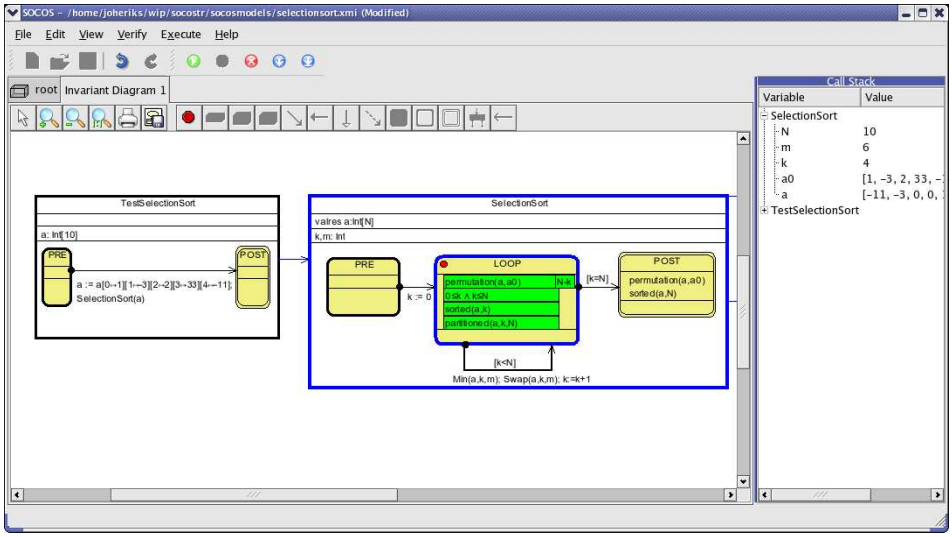


Fig. 7. Stepping through a test case of selection sort

can add a Python function to check 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 need to 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. 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 discharged, problems occur due to the use of permutation in Swap.

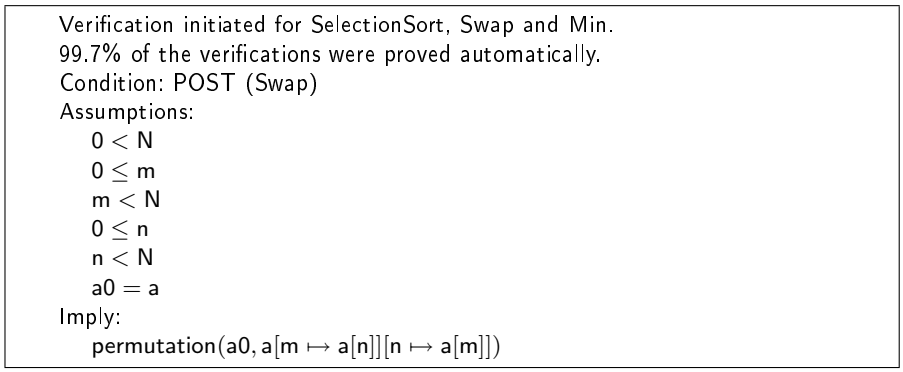


Fig. 8. Remaining condition for Swap

SOCOS has pinpointed a specific verification condition 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 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
```

This definition is actually part of the SOCOS background theory which is automatically loaded when PVS verification is initiated.[7] 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.

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

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` tactic. 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 significantly harder; 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.[8] 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 verification 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 programming 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 are teaching 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

[8] 2.8 GHz Intel Pentium 4 with 1 GB of random access memory.

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 work on the high-level structure of a formula. So, in the case of slight changes, chances are good that an existing proof is reusable.

Finally, there is a need for a way to make incremental software extensions and reason about their correctness. Stepwise Feature Introduction [22], 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. Back, R.J.: Invariant based programming. In Donatelli, S., Thiagarajan, P.S., eds.: ICATPN. Volume 4024 of Lecture Notes in Computer Science., Springer (2006) 1–18
2. Back, R.J.: Invariant based programs and their correctness. In Biermann, W., Guiho, G., Kodratoff, Y., eds.: Automatic Program Construction Techniques. Number 223-242, MacMillan Publishing Company (1983)
3. Back, R.J., Myreen, M.: Tool support for invariant based programming. In: the 12th Asia-Pacific Software Engineering Conference, Taipei, Taiwan (2005)
4. Back, R.J., Milovanov, L., Porres, I.: 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 (2005)
5. Back, R.J.: Incremental software construction with refinement diagrams. In Broy, Gunbauer, H., Hoare, eds.: Engineering Theories of Software Intensive Systems. NATO Science Series II: Mathematics, Physics and Chemistry. Springer, Marktoberdorf, Germany (2005) 3–46
6. Back, R.J., Milovanov, L., Porres, I., Preoteasa, V.: 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. (2002)
7. Dijkstra, E.W.: A constructive approach to the problem of program correctness. BIT (1968) 8:174–186
8. Reynolds, J.C.: Programming with transition diagrams. In Gries, D., ed.: Programming Methodology, Springer Verlag, Berlin (1978)
9. Back, R.J.: Program construction by situation analysis. Research Report 6, Computing Centre, University of Helsinki, Helsinki, Finland (1978)
10. van Emden, M.H.: Programming with verification conditions. IEEE Transactions on Software Engineering (1979) SE–5

11. Abrial, J.R., Lee, M.K.O., Neilson, D.S., Scharbach, P.N., Sorensen, I.H.: The B-method (software development). In Prehn, S.; Toetenel, W.J., ed.: VDM 91. Formal Software Development Methods. 4th International Symposium of VDM Europe Proceedings. Volume 2., BP Res., Sunbury Res. Centre, Sunbury-on-Thames, UK, Springer-Verlag, Berlin, Germany (1991) 398–405
12. Meyer, B.: Object-Oriented Software Construction. second edn. Prentice Hall (1997)
13. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT) **7**(3) (2005) 212–232
14. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, New York, NY, USA, ACM Press (2002) 234–245
15. van den Berg, J., Jacobs, B.: The LOOP compiler for Java and JML. Lecture Notes in Computer Science **2031** (2001) 299+
16. Burdy, L., Requet, A., Lanet, J.L.: Java applet correctness: A developer-oriented approach. In Araki, K., Gnesi, S., Mandrioli, D., eds.: FME 2003: Formal Methods: International Symposium of Formal Methods Europe. Volume 2805 of Lecture Notes in Computer Science., Springer-Verlag (2003) 422–439
17. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
18. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer-Verlag (1998) Graduate Texts in Computer Science.
19. Alanen, M., Porres, I.: The Coral Modelling Framework. In Koskimies, K., Kuzniarz, L., Lilius, J., Porres, I., eds.: Proceedings of the 2nd Nordic Workshop on the Unified Modeling Language NWUML'2004. Number 35 in General Publications, Turku Centre for Computer Science (2004)
20. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3) (2005) 365–473
21. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, M. K. Srivas: PVS: Combining specification, proof checking, and model checking. In Rajeev Alur, Thomas A. Henzinger, eds.: Proceedings of the Eighth International Conference on Computer Aided Verification CAV. Volume 1102., New Brunswick, NJ, USA, Springer-Verlag (1996) 411–414
22. Back, R.J.: 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, Springer-Verlag (2002) 162–183