Electronic Notes in Theoretical Computer Science 65 No. 5 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html pages

SOL: A Verifiable Synchronous Language for Reactive Systems

Ramesh Bharadwaj [1,2]

Center for High Assurance Computing Systems
Naval Research Laboratory
Washington DC 20375-5320, USA

Abstract

SOL (Secure Operations Language) is a synchronous programming language for implementing reactive systems. The utility of SOL hinges upon the fact that it is a *secure* language, i.e., most programs in SOL are amenable to *fully automated* static analysis techniques, such as automatic theorem proving using decision procedures or model checking. Among the unique features of SOL is the ability to express a wide class of *enforceable* safety and security policies (including the *temporal* aspects of software component interfaces) in the language itself, thereby opening up the possibility of eliminating runaway computations and malicious code, such as worms and viruses.

1 Introduction

SOL is a synchronous programming language which supports a process for the specification, design, automatic analysis, and implementation of systems, more specifically software-intensive *distributed* reactive computer systems as typified by safety- and security-critical applications in the aerospace, automotive, avionics, biomedical, military, nuclear, railway, and telecommunications industries. The process underlying SOL [4,13] helps users construct precise and unambiguous system descriptions that are easy to understand and change, and that satisfy a number of application specific as well as application independent properties such as consistency and completeness, safety, and security. Support in SOL for programming-in-the-large includes *modules*, *interfaces*, and automated analysis using *assumption-guarantee* style of reasoning. It is our belief that SOL provides a cost-effective solution to industrial-strength problems.

[1] This project was funded by the Office of Naval Research.
[2] Email: ramesh@itd.nrl.navy.mil

2 Background

The design of SOL was motivated by the concern of building high assurance systems. High assurance entails the presentation of compelling evidence that a system satisfies critical properties including functional properties (i.e., properties of services that the system delivers), and other critical properties such as security, safety, timeliness (real-time), survivability, and fault-tolerance. Since SOL is a synchronous data-flow language, programs in SOL are readily amenable to automatic static analysis techniques such as automatic theorem proving using decision procedures [6] or model checking [5,7]. Admittedly, such a style imposes some limitations on expressiveness, which may pose a problem in certain circumstances[3]. One should keep in mind that it is precisely these limitations that make many interesting theorems about SOL programs decidable, thereby opening up the possibility of fully automated analysis for SOL programs.

Many researchers have been working on processes and tools for the specification, design, analysis, and implementation of safety- and mission-critical systems [4,12,20,21]. One such approach, known as SCR (Software Cost Reduction), was developed at the Naval Research Laboratory (NRL) to document the requirements of the US Navy's A-7 aircraft [2,17]. One of the goals of SOL is to be able to directly implement specifications of high-quality, such as the ones produced in SCR, in a safe and efficient manner. For illustrative SCR examples, see [5,13,14].

Researchers at NRL have provided a formal model for the SCR notation [5,16], based upon which a number of tools have been developed [6,15]. For verifying programs in SOL, our intention is to build upon one of these tools, Salsa, which is an invariant checker for state machine descriptions. Salsa establishes a formula as an invariant (properties that are true in all reachable states) by carrying out an induction proof that uses a tightly integrated set of decision procedures (currently a combination of BDD algorithms and a constraint solver for integer linear arithmetic) for discharging the verification conditions. The use of induction and a set of optimized heuristics enable Salsa to combat the state explosion problem that plagues current model checkers.

More recently, we have been investigating the problem of building *secure* distributed applications over the infrastructure of the Internet and the World Wide Web [3]. It is widely acknowledged that *intelligent software agents* are central to the development of the capabilities required to build robust, reconfigurable, and survivable distributed applications. However, agents technology carries with it the risk of security vulnerabilities such as denial of service, malicious code, and information leaks. The design philosophy of SOL is to give developers the ability to address the security problems outlined above using an easy-to-use graphical interface supported by powerful analysis and

[3] A mitigating factor is the ability within SOL to invoke arbitrary methods (i.e., functions) written in traditional programming languages.

synthesis tools. Since agents in SOL are verifiable, composable, and modular, SOL addresses the problem of statically determining *emergent behavior* [8] that eludes current agent design and implementation approaches.

3 Related Work

The design of SOL was directly influenced by the design of SAL (the SCR Abstract Language), a specification language based on the SCR Formal Model [16]. SAL was designed to serve as an abstract interface for analysis tools such as theorem provers, model checkers, test case generators, and consistency checkers. SOL includes certain key features of SAL including the notion of *events* and modularity. The tool Salsa [6] uses SAL as the input language, and can perform automated analysis such as checking a SAL specification for unwanted nondeterminism and missing cases, in addition to the verification of invariants. Although a theorem prover, Salsa affords "push-button" automation, ease of use, and counterexample generation that typify model checkers [5].

Another language that influenced the design of SOL was LUSTRE [11], a language developed at the IMAG Institute in Grenoble. SOL resembles LUSTRE in being a textual data-flow language, and in its ability to include program fragments (i.e., *methods calls*) from traditional programming languages in which it is embedded. However, SOL does away with the (sometimes confusing) notion of *clocks* of LUSTRE without sacrificing expressiveness – SCR events are in essence "self-timed" which obviates the need for an explicit clock. The type system of SOL is richer than that of LUSTRE, and includes enumerated types, real, and string types, in addition to *opaque* types which may denote arbitrary ADTs (abstract data types) of the embedding language. It is important to note that the analysis tools of SOL will provide support for reasoning in this richer type system. Also, SOL borrows from POLLUX [22], an extension of LUSTRE, the notions of *tuples* and *arrays*. Features of SOL not in LUSTRE include mechanisms for programming-in-the-large including modules, interfaces, and mechanisms for assumption-guarantee style proofs. Other features include the ability to directly specify a wide class of *enforceable* safety and security policies (including the *temporal* aspects of software component interfaces) using the theory of security automata in [1,23].

4 Secure Operations Language (SOL)

4.1 Events

SOL borrows from SCR the notion of *events* [16]. Informally, an SCR event denotes a change of state, i.e., an event is said to occur when a state variable changes value. SCR systems are event-driven and the SCR model includes a special notation for denoting them. The notation `@T(c)` denotes the event "condition `c` became true", `@F(c)` denotes "condition `c` became false" and `@C(x)` the event "the value of expression `x` has changed". These constructs are defined formally below. In the sequel, `PREV(x)` denotes the value of expression `x` in the *previous state*.

$$\texttt{@T(c)} \stackrel{\text{def}}{=} \neg\texttt{PREV}(c) \wedge c$$

$$\texttt{@F(c)} \stackrel{\text{def}}{=} \texttt{PREV}(c) \wedge \neg c$$

$$\texttt{@C(c)} \stackrel{\text{def}}{=} \texttt{PREV}(c) \neq c$$

Events may be triggered predicated upon a condition by including a "`when`" clause. Informally, the expression following the keyword `when` is "aged" (i.e., evaluated in the *previous* state) and the event occurs only when this expression has evaluated to *true*. Formally, a *conditioned event*, defined as

$$\texttt{@T(c) when d} \stackrel{\text{def}}{=} \neg\texttt{PREV}(c) \wedge c \wedge \texttt{PREV}(d),$$

denotes the event "condition `c` became `true` when condition `d` was `true` in the previous state". Conditioned events involving the two other event constructs are defined along similar lines.

In SOL we extend the SCR event construct to include events that are triggered by the invocation of a *method* (i.e., a procedure or function call) of the embedding language. For example, the event associated with the invocation of method `push(x)` of a stack is denoted as `@push`. This provides users the ability to implement *security automata*, a special class of Büchi automata that accept safety properties [1,23].

4.2 SOL Overview

In this section we give an informal overview of SOL using the stopwatch of [11] as a running example. We have chosen this example to illustrate the expressiveness, readability, usability, and formal verification capabilities of the SOL approach. Interested readers may refer to [11] to compare this approach with other synchronous languages such as Esterel, Argos, LUSTRE, and SIGNAL. Informally, the stopwatch includes a *display* of elapsed time and a button *start_stop* that alternately puts the stopwatch in "running" and "stopped" states. Initially the stopwatch is stopped. It also receives a signal *HS* every 1/100 second, which is used to compute the time spent in the "running" state. The stopwatch includes a second button – *button_2* – whose interpretation depends on the mode (i.e., externally visible state) of the stopwatch. When the stopwatch is stopped and the displayed time is running, the button is interpreted as a RESET command; otherwise it corresponds to a LAP command, which freezes the display while the stopwatch is still running.

In SOL, a system's behavior is described in terms of modules. A module declaration may include one or more *attributes*. The attribute `deterministic` declares the module as being free of nondeterminism (which will be checked by the SOL compiler). Attribute `reactive` declares that the module will

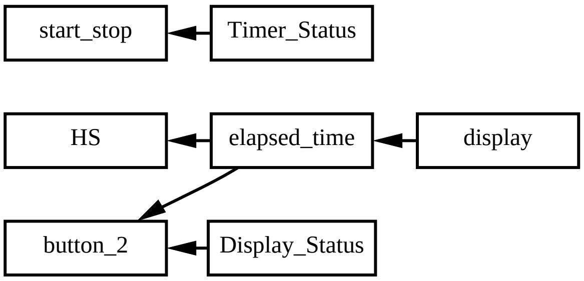

Fig. 1. The dependency graph of module stopwatch.

not cause a state change or invoke a method unless its (visible) environment initiates an event by changing state or invoking a method; moreover, the module's response to an environmental event will be immediate; i.e., in the next immediate step. Each module may contain state variables, each one falling into one of three categories: `monitored variables` which denote environmental variables observable to the module; `controlled variables` which denote environment-visible variables the module controls; and `internal variables` which are updated by the module but not visible to the environment.

For the stopwatch, we implement the system as a `deterministic` and `reactive` module. We identify three monitored variables – `start_stop`, `HS`, and `button_2` and a controlled variable `display`. The variable `start_stop`, of type `boolean`, indicates the status (pressed or otherwise) of the button *start_stop*. Similarly, variable `button_2`, also of type `boolean` indicates the status of *button_2*. The boolean variable `HS` indicates the presence (or absence) of signal *HS*. We also identify a controlled variable `display` of type `Time` (i.e., integer in the range $0 \ldots \infty$) corresponding to the stopwatch display. Finally, module `stopwatch` includes three internal variables: `elapsed_time`, also of type `Time`; `Display_Status` which can take on values from the set $\{\texttt{frozen}, \texttt{active}\}$; and `Timer_Status` of type $\{\texttt{stopped}, \texttt{running}\}$.

Each controlled and internal variable of a module has one and only one *definition* which determines when and how the variable gets updated. All definitions of a module `m` implicitly specify a *dependency relation* D_m such that a variable `a` depends on variable `b` (i.e., $(\texttt{a}, \texttt{b}) \in D_m$) if and only if `b` appears in the definition of `a`. Note that variable `a` may depend on the **previous** values of other variables (including itself) which has no effect on the dependency relation. A *dependency graph* may be inferred from the dependency relation by taking each variable in the module to be a node and including an edge from `a` to `b` if `a` depends on `b`[4]. The dependency graph of `stopwatch` is shown in Figure 1. Note that monitored variables of the module appear to the left and controlled variables to the right. It is required that the dependency graph of each module is acyclic.

Intuitively, the execution of a SOL program proceeds as a sequence of *steps*, each initiated by an event (known as the *triggering event*). Each step

[4] The notion of a dependency relation is easily extended to the entire system.

```
deterministic reactive module stopwatch {

type definitions
  DS_type = {frozen, active};
  TS_type = {stopped, running};
  Time    = integer in [0:infinity];

monitored variables
  boolean HS, start_stop, button_2;

controlled variables
  Time display;

internal variables
  Time    elapsed_time;
  TS_type Timer_Status;
  DS_type Display_Status;

assumptions
  ...

guarantees
  ...

definitions /* Definitions of all internal and controlled variables */
  ...
} // end module stopwatch
```

Fig. 2. The skeleton of the SOL module for stopwatch.

of a SOL module comprises a set of variable updates and method calls that are consistent with the dependency relation D_m of that module. Computation of each step of a module proceeds as follows: the module or its environment nondeterministically initiates a triggering event; each module in the system *responds* to this event by updating all its dependent (i.e., internal and controlled) variables and/or invoking methods. All updates and method calls of the system are assumed to be synchronous (similar to the Synchrony Hypothesis of languages such as Esterel, LUSTRE, etc. [11]) – it is assumed that the response to a triggering event is completed in one step, i.e, all updates to dependent variables and all method calls are performed by the modules of the system before the next triggering event. Moreover, all updates are performed in an order that is consistent with the partial order imposed by the dependency graph. For the stopwatch example, module `stopwatch` responds to a triggering event [5] by updating its dependent variables in compliance with the dependency order (see Figure 1). One possible order is `Timer_Status` → `elapsed_time` → `display` → `Display_Status`.

The skeleton of the SOL module for stopwatch is shown in Figure 2. Note that C-style comments are supported – all text between an opening "`/*`" and

[5] Since `stopwatch` is `reactive`, all triggering events are external to the module.

defn : *lvalue* "=" *expr* | *lvalue* "=" "initially" *expr* "then" *expr* ";"

lvalue : *ID* | *ID* "[" *index* "]" | "[" *lvalue* ["," *lvalue*]* "]"

expr : *value* | "!" *expr* | *expr bool_binop expr* | *if_expr* | *case_expr* | *basic_event* | *cond_event* | "PREV" "(" *expr* ")" | *expr rel_binop expr* | "+" *expr* | "-" *expr* | *expr arith_binop expr* | *ID* "[" *index* "]" | *ID* "(" [*expr_l*]? ")" | "[" *expr_l* "]" | "(" *expr* ")"

if_expr : "if" "{" ["[]" *expr* "->" *expr*]+ ["otherwise" "->" *expr*]? "}"

case_expr : "case" *expr* "{" ["[]" *value* ["," *value*]* "->" *expr*]+ ["otherwise" "->" *expr*]? "}"

cond_event : *basic_event* "when" *expr*

basic_event : "@*ID* " ["(" *expr_l* ")"]? | "@T" "(" *expr* ")" | "@F" "(" *expr* ")" | "@C" "(" *expr* ")"

expr_l : *expr* ["," *expr*]*

value : *index* | *REAL* | *STRING* | "true" | "false" | "infinity"

index : *scalar_value* | *scalar_value* ":" *scalar_value*

scalar_value : *ID* | *INT*

bool_binop : "&" | "&&" | "|" | "||" | "=>" | "<=>"

rel_binop : "<" | "<=" | "==" | "!=" | ">" | ">="

arith_binop : "+" | "-" | "*" | "/"

Legend:

|	Choice
[ ]?	Optional
[ ]*	Zero or more
[ ]+	One or more

Fig. 3. The syntax of SOL definitions.

closing "*/" is ignored. Alternately, comments may begin with "//" and terminate by the end of the line. Comments may be nested. The module definition comprises a sequence of sections, all of them optional, each beginning with one or more keywords. User-defined types are defined in the `type definitions` section. Each entry in this section consists of an identifier for the type, followed by its definition, which may be in terms of the built-in types, their subranges, or enumerated types.

```
assumptions
  NAT = initially (!HS & !button_2 & !start_stop) then
        (!HS & !button_2) | (!HS & !start_stop) | (!button_2 & !start_stop);


guarantees
  Button2_to_frozen = initially true then
    @T(button_2) when (Timer_Status == running and Display_Status == active)
      => Display_Status == frozen;

  Button2_to_not_frozen = initially true then
    @T(button_2) when Display_Status == frozen => Display_Status == active;


definitions
  Display_Status = initially active then
    if {
      [] @T(button_2) when (Display_Status == frozen) -> active

      [] @T(button_2) when (Display_Status == active &
                            Timer_Status == running)  -> frozen
    otherwise -> PREV(Display_Status)
    };

  Timer_Status = initially stopped then
    if {
      [] @T(start_stop) when (Timer_Status == stopped) -> running
      [] @T(start_stop) when (Timer_Status == running) -> stopped
    otherwise -> PREV(Timer_Status)
    };

  display = initially 0 then
    if {
      []@C(elapsed_time) when (Display_Status == active) -> elapsed_time
    otherwise -> PREV(display)
    };

  elapsed_time = initially 0 then
    if {
      []@T(button_2) when (Timer_Status == stopped &
                           Display_Status == active) -> 0
      []@T(HS) when (Timer_Status == running) -> PREV(elapsed_time) + 1
    otherwise -> PREV(elapsed_time)
    };
```

Fig. 4. Sections **assumptions**, **guarantees**, and **definitions** of stopwatch.

4.3 SOL Definitions

The `definitions` section is at the heart of a SOL module. The syntax of SOL definitions is shown in Figure 3. This section determines how each internal and controlled variable of the module is updated in response to events (i.e., method calls or state changes) generated either internally or by the module's environment. Figure 4 includes all definitions of the module stopwatch.

A variable definition is either a *one-state* or a *two-state* definition. A one-state definition, of the form $x = expr$ (where *expr* is an expression), defines the value of variable `x` in terms of the values of other variables *in the same state*. A two-state variable definition, of the form $x =$ `initially` *init* `then` *expr* (where *expr* is a two-state expression), requires the initial value of `x` to equal expression *init*; the value of `x` in each subsequent state is determined in terms of the values of variables in that state *as well as the previous state* (specified using operator `PREV` or by a `when` clause). A *conditional expression*, consisting of a sequence of branches "`[]` guard $\rightarrow$ expression", is introduced by the keyword "`if`" and enclosed in braces (`"{"` and `"}"`). A guard is a boolean expression. The semantics of the conditional expression `if` $\{\ []g_1 \rightarrow \mathrm{expr}_1\ \ []g_2 \rightarrow \mathrm{expr}_2 \ldots\ \}$ is defined along the lines of Dijkstra's *guarded commands* [9] – in a given state, its value is equivalent to expression expr_i whose associated guard g_i is true. If more than one guard is true, the expression is nondeterministic. It is an error if none of the guards evaluates to `true`, and execution aborts. The *case expression* `case` *expr* $\{\ []v_1 \rightarrow \mathrm{expr}_1\ \ []v_2 \rightarrow \mathrm{expr}_2 \ldots\ \}$ is equivalent to the conditional expression `if` $\{\ [](expr == v_1) \rightarrow \mathrm{expr}_1\ \ [](expr == v_2) \rightarrow \mathrm{expr}_2 \ldots\ \}$. The conditional expression and the case expression may optionally have an `otherwise` clause with the obvious meaning. In this paper, we shall not elaborate on the *tuple* and *array* constructs of SOL (see [22] for details).

The definitions of module stopwatch, shown in Figure 4, have a direct correspondence to phrases in the prose specification. For example, statements

```
[] @T(start_stop) when (Timer_Status == stopped) -> running

[] @T(start_stop) when (Timer_Status == running) -> stopped
```

correspond to the phrase "... [the] button *start_stop* alternately puts the stopwatch in 'running' and 'stopped' states".

4.4 Assumptions and Guarantees

The assumptions of a module, which are typically assumptions about the environment of the subsystem being defined, are included in the `assumptions` section. It is up to the user to make sure that the set of assumptions is not inconsistent, i.e., a logical contradiction. Users specify the module invariants in the `guarantees` section, which is automatically verified by a tool such as Salsa. The syntax for specifying module assumptions and guarantees is identical to that of module definitions, in other words, we have the expressiveness of the full language in these clauses. This does not have a detrimental effect on the proof tools, since most commonly encountered theorems about SOL programs are decidable.

The assumptions and guarantees for the stopwatch example are shown in Figure 4. The assumption `NAT` specifies that all the monitored variables of the module are `false` in the initial state and that in all subsequent states at most one of the monitored variables is `true`. The guarantees are a formalization of the sentence "If button_2 is pressed when the stopwatch is running and active, it becomes frozen; when it is pressed when the stopwatch is frozen, it becomes active." from [11].

5 Formal Verification

Salsa is a tool for the verification of synchronous reactive systems. The verification performed by Salsa is invariant checking in addition to consistency checking [16] which flags undesirable instances of nondeterminism and missing cases in a module. Consider the following code fragment of stopwatch:

```
elapsed_time = initially 0 then
  if {
    []@T(button_2) when (Timer_Status == stopped &
                         Display_Status == active) -> 0
    []@T(HS) when (Timer_Status == running) -> PREV(elapsed_time) + 1
  otherwise -> PREV(elapsed_time)
  };
```

Checking for disjointness is to determine whether events

$$e_1 = \texttt{@T(button\_2) when (Timer\_Status} == \texttt{stopped \&}$$

$$\texttt{Display\_Status} == \texttt{active)}$$

$$e_2 = \texttt{@T(HS) when (Timer\_Status} == \texttt{running)}$$

of the two guards can *both* occur in **any** reachable system state. If so, the module is said to have a *disjointness error* (since the module is declared as `deterministic`).

The above problem may be reduced to checking whether the expression "`initially true then` $\neg(e_1 \wedge e_2)$" is an invariant of stopwatch. All the disjointness verification conditions of stopwatch were verified automatically by Salsa in about a tenth of a second. Along similar lines, one can prove that the properties in the `guarantees` section are invariants. These too were verified automatically by Salsa in under a second. If an invariant is not provable, Salsa returns a counterexample. However, one should keep in mind that due to the incompleteness of induction, users must *validate* that the returned counterexample is reachable. By examining the counterexamples, users will be able to either determine that there is a problem or will have to prove additional invariants as lemmas in order to prove the original invariant.

6 Enforcement Automata

In this section, we shall examine how *enforceable* safety and security policies [23] are expressed in SOL as *enforcement automata* (also known as *security agents* [3]). The enforcement mechanism of SOL works by terminating all executions of a program for which the policy being enforced no longer holds. For reasons of readability and maintainability, we prefer to use explicit automata for enforcing safety properties and security policies, although any language that allows references to previous values of variables may suffice. Unlike assertions, where no additional state is maintained, SOL enforcement automata may include additional variables that are updated during the transitions of the automata.

```
deterministic reactive module safestack(integer max_depth) {

interfaces
  void    push(integer x);
  void    pop();
  integer top();


internal variables
  {empty, nonempty}        status;
  integer in [0:max_depth] depth;


guarantees
  INV1 = (status == empty) <=> (depth == 0);


definitions
  [status, depth] = initially [empty, 0] then
  case PREV(status) {
    [] empty ->
      if {
        [] @push -> [nonempty, PREV(depth) + 1]
        // other operations illegal!
      }
    [] nonempty ->
      if {
        [] @top -> [PREV(status), PREV(depth)]
        // @pop when (depth == 0) impossible! (by INV1)
        [] @pop when (depth > 1)  -> [nonempty, PREV(depth) - 1]
        [] @pop when (depth == 1) -> [empty, 0]
        [] @push when (depth < max_depth) -> [nonempty, PREV(depth) + 1]
        // @push when (depth == max_depth) illegal!
      }
  }; // end case
} // end module safestack
```

Fig. 5. A SOL module providing a safe interface to stack.

6.1 Safety Automata

We examine how SOL automata are used to enforce safety policies. The example we shall use is a stack, which has the associated methods `push`, `pop`, and `top`. Informally, `push(x)` pushes the value of integer variable `x` on the stack and `pop()` pops the topmost value off the stack. The method `top()` returns the current value at the top of the stack. The stack can accommodate at most `max_depth` items. The safety policies we wish to enforce are: (i) No more than `max_depth` items are pushed on the stack. (ii) Invocations of methods `top` and `pop` are disallowed on an empty stack.

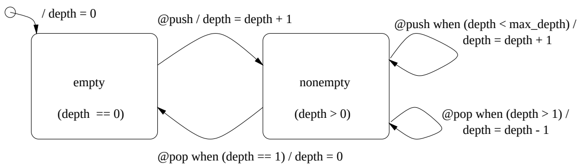

Fig. 6. Visual representation of `safestack`.

The classical way of specifying the correct use of a stack would be to write a so-called *class invariant*, often specified as predicates on the "old" and "new" values of program variables. Languages such as Eiffel [19] with explicit support for Design by Contract [18] include constructs for specifying and checking such invariants. However, presently popular object-oriented programming languages lack such mechanisms, and therefore treat class invariants mostly as comments, and provide no tool support to analyze them. A unique feature of SOL is the ability to perform such checks on existing implementations in a language-neutral manner. Figure 5 shows a SOL module `safestack` which enforces these safety policies on *all* SOL modules which use the stack object (implemented in the embedding language). Figure 6 is the module `safestack` rendered in the visual syntax of SOL. Note that by deliberately omitting the `otherwise` clauses in the `if` statements, we abort the execution of the program when none of the guards is `true`. If this is too drastic, corrective action may be specified in an `otherwise` clause; for example, to ignore all `push` actions when the stack is full.

6.2 Security Automata

We use the example from [23] to illustrate how we may implement a security policy that allows a software agent to send data to remote hosts (using method `send`) as well as read local files (using method `file_read`). However, invocations of `send` subsequent to `file_read` are disallowed. It is difficult, if not impossible, to configure current systems to implement such a policy. For example, it cannot be implemented in the "sandbox" model of Java [10] in which one may either always or never allow access to a system resource. As shown in Figure 7, this policy is easily implemented in SOL.

```
deterministic reactive module SecureRead {

interfaces
  string file_read(string filename, int position, int size);
  void   send(string address, string data);

internal variables
  {no_reads, read_performed} status;

definitions
  status = initially no_reads then
  case PREV(status) {
    [] no_reads ->
      if {
        [] @send      -> PREV(status)
        [] @file_read -> read_performed
      }
    [] read_performed ->
      if {
        [] @file_read -> read_performed
        // @send illegal!
      }
  }; // end case
} // end module SecureRead
```

Fig. 7. A SOL module that implements safe access to local files.

7 Future Work

We plan to continue the development of design and analysis tools for SOL programs, which will include a graphical user interface for a visual (Statecharts-like) representation for SOL, and verification tools such as automatic invariant generators and checkers, theorem provers, and model checkers. A Java^{TM} interpreter for SOL, SOLi, is currently under development. We plan to support other language embeddings such as C$^{\#}$ and C^{++}. Planned extensions to the interpreter includes support for security (such as authentication, authorization, non-repudiation, and confidentiality) and *decentralized* distributed execution, for which we need to address associated problems such as fault-tolerance, load balancing, self-stabilization, and survivability. Another area of research is to provide support for implementing hard real-time systems.

8 Acknowledgements

This project is funded by the Office of Naval Research. The author thanks Ralph Jeffords, Amit Khashnobish, and James Tracy for many helpful discussions and comments. Ralph suggested several improvements to SOL and to previous drafts of this paper. Amit implemented the SOL compiler. The author also thanks the anonymous referees for their insightful comments.

References

[1] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[2] T. A. Alspaugh, S. R. Faulk, K. H. Britton, R. A. Parker, D. L. Parnas, and J. E. Shore. Software requirements for the A-7E aircraft. Technical Report NRL-9194, Naval Research Lab., Wash., DC, 1992.

[3] R. Bharadwaj. An infrastructure for secure interoperability of agents. Technical report, Naval Research Laboratory, Washington, DC, To appear.

[4] R. Bharadwaj and C. Heitmeyer. Hardware/software co-design and co-validation using the SCR method. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT'99)*, San Diego, CA, November 1999.

[5] R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, 6(1), January 1999.

[6] R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '2000)*, Berlin, March 2000.

[7] E. M. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Trans. on Prog. Lang. and Systems*, 8(2):244–263, April 1986.

[8] A. Cottrel. Emergent properties of complex systems. In *The Encyclopedia of Ignorance*, pages 129–135, Premagon Press, 1977.

[9] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[10] L. Gong. Java Security: Present and near future. *IEEE Micro*, 15(3):14–19, 1997.

[11] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[12] Mats P. E. Heimdahl and Nancy Leveson. Completeness and consistency analysis of state-based requirements. In *Proc. of 17th Int'l Conf. on Softw. Eng. (ICSE '95)*, pages 3–14, Seattle, WA, April 1995. ACM.

[13] C. Heitmeyer and R. Bharadwaj. Applying the SCR requirements method to the Light Control Case Study. *Journal of Universal Computer Science*, 6(7), 2000.

[14] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Softw. Eng.*, 24(11), November 1998.

[15] C. Heitmeyer, J. Kirby, Jr., B. Labaw, and R. Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *Proc. Computer-Aided Verification, 10th Annual Conf. (CAV'98)*, Vancouver, Canada, 1998.

[16] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, April–June 1996.

[17] K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE TSE*, SE-6(1):2–13, January 1980.

[18] Bertrand Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, 1992.

[19] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[20] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[21] D. L. Parnas and J. Madey. Functional documentation for computer systems. *Science of Computer Programming*, 25(1):41–61, October 1995.

[22] Frederic Rocheteau and Nicolas Halbwachs. POLLUX: A Lusture based hardware design environment. In P. Quinton and Y. Robert, editors, *Proc. Conf. on Algorithms and Parallel VLSI Arch. II*, Chateau de Bonas, June 1991.

[23] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.