A Framework for the Formal Analysis of Multi-Agent Systems

In Proc. ***Formal Approaches to Multi-Agent Systems (FAMAS)*****, affiliated with ETAPS 2003, April 12, 2003. Warsaw, Poland.**

Ramesh Bharadwaj

Center for High Assurance Computer Systems
Naval Research Laboratory
Washington DC 20375 USA

Abstract. In this paper we present an integrated formal framework for the specification and analysis of Multi-Agent Systems (MAS). Agents are specified in a synchronous programming language called Secure Operations Language (SOL) which supports the modular development of *secure* agents. Multi-agent systems are constructed from individual agent modules by using the composition operator of SOL, the semantics of which are guaranteed to preserve certain individual agent properties. The formal semantics and the underlying framework of SOL also serve as the basis for analysis and transformation techniques such as abstraction, consistency checking, verification by model checking or theorem proving, and automatic synthesis of agent code. Based on this framework, we are currently developing a suite of analysis and transformation tools for the formal specification, analysis, and synthesis of multi-agent systems.

1 Introduction

Building trusted applications is hard, especially in a distributed or mobile setting. Existing methods and tools are inadequate to deal with the multitude of challenges posed by distributed application development. The problem is exacerbated in a hostile environment such as the Internet where, in addition, applications are vulnerable to malicious attacks. It is widely acknowledged that intelligent software agents provide the right paradigm for developing agile, reconfigurable, and efficient distributed applications. Distributed processing in general carries with it risks such as denial of service, Trojan horses, information leaks, and malicious code. Agent technology, by introducing autonomy and code mobility, may exacerbate some of these problems. In particular, a malicious agent could do serious damage to an unprotected host, and malicious hosts could damage agents or corrupt agent data.

Secure Infrastructure for Networked Systems (SINS) being developed at the Naval Research Laboratory is a middleware for secure agents intended to provide the required degree of trust for mobile agents, in addition to ensuring their compliance with a set of enforceable security policies. An infrastructure such as SINS is central to the successful deployment and transfer of distributed agent technology to Industry because security is a necessary prerequisite for distributed computing.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE
2003

2. REPORT TYPE

3. DATES COVERED
00-00-2003 to 00-00-2003

4. TITLE AND SUBTITLE
A Framework for the Formal Analysis of Multi-Agent Systems

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Research Laboratory,Center for High Assurance Computer Systems,4555 Overlook Avenue, SW,Washington,DC,20375

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR’S ACRONYM(S)

11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:

a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
unclassified	**unclassified**	**unclassified**		**13**	

2 SINS Architecture

Figure 1 shows the architecture of SINS. Agents are created in a special purpose synchronous programming language called Secure Operations Language (SOL) [5–7]. A SOL application comprises a set of agent modules, each of which runs on an Agent Interpreter (AI). The AI executes the module on a given host in compliance with a set of locally enforced security policies. A SOL multi-agent system may run on one or more AIs, spanning multiple hosts across multiple administrative domains. Agent Interpreters communicate among themselves using an inter-agent protocol [18], similar to SOAP/XML [19].

3 A Brief Introduction to SOL

A module is the unit of specification in SOL and comprises variable declarations, assumptions and guarantees, and definitions. The `assumptions` section includes assumptions about the environment of the agent. Execution aborts when any of these assumptions are violated by the environment. The required safety properties of an agent are specified in the `guarantees` section. The `definitions` section specifies updates to internal and controlled variables as functions (or more generally as relations). A SOL *module* describes the required relation between *monitored variables*, variables in the environment that the agent monitors, and *controlled variables*, variables in the environment that the agent controls. Additional internal variables are often introduced to make the description of the agent concise. In this paper, we only distinguish between monitored variables, i.e., variables whose values are specified by the environment, and *dependent variables*, i.e., variables whose values depend on the values of monitored variables. Dependent variables include all the controlled variables and internal variables of an agent module.

3.1 Events

SOL borrows from SCR the notion of *events* [13]. Informally, an SCR event denotes a change of state, i.e., an event is said to occur when a state variable

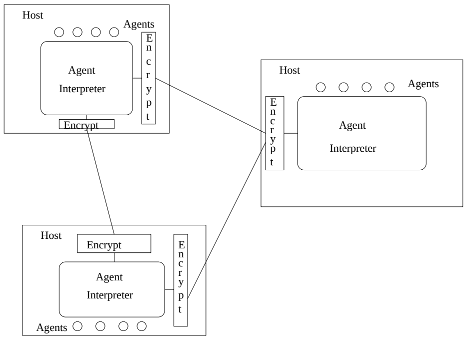

Fig. 1. Architecture of SINS.

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, 17].

3.2 Definitions

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 → 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` $\{\ []g_1 \rightarrow \text{expr}_1\ \ []g_2 \rightarrow \text{expr}_2 \ldots\ \}$ is defined along the lines of Dijkstra's *guarded commands* [10] – 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* $\{\ []v_1 \rightarrow \text{expr}_1\ \ []v_2 \rightarrow \text{expr}_2 \ldots\ \}$ is equivalent to the conditional

```
deterministic reactive enforcement module
  safestack(integer max_depth) {
  // assumption: max_depth > 0
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 > 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. 2. Agent module for safestack.

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.

3.3 An Example: Safety Policy Enforcement

We examine how SOL agents are used to enforce safety policies on a given agent interpreter. 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 and leaves the stack unchanged. The stack can accommodate at most `max_depth` items. We

```
deterministic reactive enforcement module SecureRead {

interfaces
  string file_read(string filename, integer position, integer 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. 3. A SOL module that enforces safe access to local files.

assume that other agents (not shown) access the stack by invoking its methods. 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. Figure 2 shows a SOL enforcement agent `safestack` which enforces these safety policies on *all other* SOL agents which use the stack object (implemented in the embedding language). Note that by deliberately omitting the `otherwise` clauses in the `if` statements, we abort the execution of a SOL agent when none of the guards is `true` during execution. 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.

3.4 Security Automata

We use the example from [17] to illustrate how we may enforce 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 enforce such a policy. For example, it cannot be enforced in the "sandbox" model of Java [11] in which one may either always or never allow access to a system resource. As shown in Figure 3, this policy is easily implemented in SOL.

4 Formal Semantics

State Machines A SOL module describes a state machine [6]. A *state machine* Σ is a quadruple (V, S, Θ, ρ), where $V = \{v_1, v_2, \ldots, v_n\}$ is a finite set of *state variables*; S is a nonempty set of *states* where each state $s \in S$ maps each $v \in V$ to its range of legal values; $\Theta : S \rightarrow \mathit{boolean}$ is a predicate characterizing the set of *initial states*; $\rho : S \times S \rightarrow \mathit{boolean}$ is a predicate characterizing the *transition relation*. We write Θ as a logical formula involving the names of variables in V. Predicate ρ relates the values of the state variables in a previous state $s \in S$ to their values in the current state $s' \in S$. We write ρ as a logical formula involving the values of state variables in the previous state (specified using operator `PREV` or by a `when` clause) and in the current state.

SOL Predicates Given a state machine $\Sigma = (V, S, \Theta, \rho)$ we classify a predicate $p : S \rightarrow \mathit{boolean}$ as a *one-state* predicate of Σ and a predicate $q : S \times S \rightarrow \mathit{boolean}$ as a *two-state* predicate of Σ.

More generally, *SOL predicate* refers to either a one-state or two-state predicate, and *SOL expression* refers to logical formulae or terms containing references to current or previous values of state variables in V.

Reachability Given a state machine $\Sigma = (V, S, \Theta, \rho)$, a state $s \in S$ is *reachable* (denoted $\mathit{Reachable}_\Sigma(s)$) if

(i) $\Theta(s)$ or
(ii) $\exists s' \in S : \mathit{Reachable}_\Sigma(s')$ and $\rho(s', s)$

Invariants A one-state predicate p is a *state invariant* of Σ if and only if

$$\forall s : \mathit{Reachable}_\Sigma(s) \Rightarrow p(s)$$

A two-state predicate q is a *transition invariant* of Σ if and only if

$$\forall s, s' : (\mathit{Reachable}_\Sigma(s) \wedge \rho(s, s')) \Rightarrow q(s, s')$$

More generally, a SOL predicate x is an *invariant* of Σ if x is a state invariant or transition invariant of Σ.

Verification For a SOL module describing a state machine Σ, and a set of SOL predicates $X = \{x_1, x_2, \ldots x_m\}$, verification is the process of establishing that each SOL predicate $x_i \in X$ is an invariant of Σ.

5 SOL Module

A SOL module describes both an agent's environment, which is usually nondeterministic, and the required agent behavior, which is usually deterministic [8, 12]. Recall that for each agent we distinguish between its monitored variables, i.e., variables in its environment, and *dependent variables*, i.e., variables whose values are determined by the agent. Dependent variables include all the controlled variables and internal variables of an agent module. In the sequel, we assume that variables $v_1, v_2, \ldots, v_I$ are an agent's monitored variables, and that variables $v_{I+1}, v_{I+2}, \ldots, v_n$ are the agent's dependent variables. The notation $NC(v_1, v_2, \ldots, v_k)$ is used as an abbreviation for the SOL predicate $(v_1 = PREV(v_1)) \wedge (v_2 = PREV(v_2)) \wedge \ldots \wedge (v_k = PREV(v_k))$.

Components of the state machine $\Sigma = (V, S, \Theta, \rho)$ are specified in the section `definitions` of a SOL module. The initial predicate Θ is specified in terms of the initial values for each variable in V, i.e., as predicates $\theta_{v_1}, \theta_{v_2}, \ldots, \theta_{v_n}$, so that $\Theta = \theta_{v_1} \wedge \theta_{v_2} \wedge \ldots \wedge \theta_{v_n}$. The transition relation ρ is specified as a set of assignments, one for each dependent variable of Σ, i.e., as SOL predicates $\rho_{v_{I+1}}, \rho_{v_{I+2}}, \ldots, \rho_{v_n}$, each of which is of the form:

$$v_i = \begin{cases} e_1 \; if \; g_1 \\ e_2 \; if \; g_2 \\ \vdots \\ e_k \; if \; g_k \end{cases}$$

where $I+1 \leq i \leq n$, and $e_1, e_2, \ldots, e_k$ are SOL expressions, and $g_1, g_2, \ldots, g_k$ are SOL predicates. To avoid circular definitions, we impose an additional restriction on the occurrences of state variables in these expressions as below:

> Define *dependency relations* D_{new}, D_{old}, and D on $V \times V$ as follows: For variables v_i and v_j, the pair $(v_i, v_j) \in D_{new}$ iff v_j occurs outside a $PREV()$ clause in the SOL expression defining v_i; the pair $(v_i, v_j) \in D_{old}$ iff $PREV(v_j)$ occurs in the SOL expression defining v_i; and $D = D_{new} \cup D_{old}$. We require D^+_{new}, the transitive closure of the D_{new} relation, to define a partial order.

5.1 Composing SOL Modules

Consider two SOL modules describing the state machines $\Sigma_1 = (V_1, S_1, \Theta_1, \rho_1)$ and $\Sigma_2 = (V_2, S_2, \Theta_2, \rho_2)$. We define the *composition* of the two SOL agents $\Sigma = (V, S, \Theta, \rho)$ as $\Sigma = \Sigma_1 \| \Sigma_2$ where

$$\begin{aligned} V &= V_1 \cup V_2 \\ \Theta &= \Theta_1 \wedge \Theta_2 \\ \rho &= \rho_1 \wedge \rho_2 \\ &\text{Each } s \in S \text{ maps each } v \in V \text{ to its range of legal values} \end{aligned}$$

provided that there is no circularity in the occurrences of variables in ρ. Also by assumption, it is the case that ρ_1 and ρ_2 define disjoint sets of state variables.

6 Verification

In this section, we discuss how two well-known verification approaches may be used for establishing the invariance of predicates for a state machine Σ.

6.1 Theorem Proving

The first approach, which uses *induction*, is popularly known as **theorem proving**. Due to its use of *logical weakening*, this approach avoids the explicit construction of the state space and the calculation of predicate *Reachable*.

Proof Rules

Rule SINV Let p be a one-state predicate of $\varSigma$. The following are sufficient conditions to show that p is an invariant of $\varSigma$, i.e., $\forall s : \mathit{Reachable}_\sigma(s) \Rightarrow p(s)$:

$$\boxed{\begin{array}{l}\text{S1: } \forall s : \varTheta(s) \Rightarrow p(s) \text{ and} \\ \text{S2: } \forall s, s' : (p(s) \wedge \rho(s, s')) \Rightarrow p(s').\end{array}}$$

Rule TINV Let q be a two-state predicate of $\varSigma$. The following are sufficient conditions to show that q is a transition invariant of $\varSigma$:

$$\boxed{\begin{array}{l}\text{T1: } \forall s, s' : (\varTheta(s) \wedge \rho(s, s')) \Rightarrow q(s, s') \\ \text{T2: } \forall s, s', s'' : (q(s, s') \wedge \rho(s', s'')) \Rightarrow q(s', s'')\end{array}}$$

Proof: The soundness of the above rules follows by induction from the definition of *Reachable*.

Proof Rules of SOLver We are constructing an automatic verification tool SOLver, based on theorem proving by induction, for the verification of agent properties. The proof rules we use for verification are weaker forms of the proof rules **SINV** and **TINV**. The tool SOLver is based upon our patented technology developed in connection with the formal verification tool Salsa [9].

Rule SINV-W Let p be a one-state predicate of $\varSigma$. The following are sufficient conditions to show that p is an invariant of $\varSigma$:

$$\boxed{\begin{array}{l}\text{S1-W: } \forall s : \varTheta(s) \Rightarrow p(s) \text{ and} \\ \text{S2-W: } \forall s, s' : \rho(s, s') \Rightarrow p(s').\end{array}}$$

Proof: This is a weaker form of **SINV**.

Rule TINV-W Let q be a two-state predicate of Σ. The following is a sufficient condition to show that q is a transition invariant of Σ:

$$\boxed{\text{T1-W: } \forall s, s' : \rho(s, s') \Rightarrow q(s, s')}$$

Proof: The result follows directly from the definition of transition invariant.

6.2 Model Checking

In general, the approach popularly known as *model checking* computes the set characterized by the predicate *Reachable* either explicitly or implicitly. *Explicit state* model checking computes the set by enumerating each state in the state space. *Symbolic model checking* computes predicate *Reachable* by symbolic execution, using a canonical representation (such as BDDs) for logical formulae characterizing sets of states. One important advantage of model checking over theorem proving is its ability to provide a *counterexample* by which we mean a sequence of states $s_0, s_1, \ldots, s_{n-1}, s_n$ such that $\Theta(s_0)$ and $\forall(1 \leq i \leq n) : \rho(s_{i-1}, s_i)$, and either $\neg x(s_n)$ (for a one-state predicate x) or $\neg x(s_{n-1}, s_n)$ (for a two-state predicate x) where x is a presumed invariant of state machine Σ[1]. In addition to its limited applicability to finite state systems, a major disadvantage of model checking is *state explosion* which refers to the intractable complexity of the algorithms for computing predicate *Reachable*.

7 Abstraction

The general idea behind abstraction is that in order to verify the invariance of a SOL predicate x for a state machine $\Sigma = (V, S, \Theta, \rho)$, one verifies instead that x is an invariant of a different state machine $\Sigma_A = (V_A, S_A, \Theta_A, \rho_A)$. We say that Σ_A is a *sound abstraction* of Σ relative to the invariance of x if x is an invariant of Σ_A implies that x is an invariant of Σ. We say that Σ_A is a *complete abstraction* of Σ relative to the invariance of x if x is an invariant of Σ implies that x is an invariant of Σ_A. If Σ_A is a sound and complete abstraction of Σ relative to the invariance of x, it is also called an *exact abstraction*. In general, the quotient system Σ_A, generated by an equivalence relation on S that is a *bisimulation* on Σ, will be exact for *all* the invariants of Σ. For some interesting methods that are both sound and complete, we refer the reader to [8].

The following theorems about abstraction are a generalization of many of the practical "abstraction methods" used in the verification of invariants for SCR specifications [8, 12].

[1] Theorem proving *does* provide some information, which may be thought of as a "partial counterexample", upon proof failure. For example, when application of the proof rule **SINV** (see previous section) fails, one can usually extract information about the state or pair of states for which one of the premises of the rule is false.

7.1 A Theorem of Sound Abstractions

Let $\Sigma = (V, S, \Theta, \rho)$ be a state machine and let $\Sigma_R = (V, S, \Theta_A, \rho_A)$ be another state machine constructed such that:

(a) $\forall s \in S : \Theta(s) \Rightarrow \Theta_A(s)$, and
(b) $\forall s, s' \in S : \rho(s, s') \Rightarrow \rho_A(s, s')$

then Σ_R is a sound abstraction of Σ with respect to the invariance of any SOL predicate x, i.e., if x is an invariant of Σ_R, then x is also an invariant of Σ.

Proof: We initially prove the following lemma:

> *Lemma 1:* For state machines Σ and Σ_R defined as above, $\forall s : \mathit{Reachable}_{\Sigma}(s) \Rightarrow \mathit{Reachable}_{\Sigma_R}(s)$.
> Proof: Let s be a state of Σ. Suppose s is reachable in Σ. Then, by the definition of $\mathit{Reachable}_{\Sigma}$, $\Theta(s)$ *or* $\exists s_0, s_1, \ldots, s_{n-1}, s : \Theta(s_0) \wedge \rho(s_0, s_1) \wedge \ldots \wedge \rho(s_{n-1}, s)$. From conditions (a) and (b) above, and by the definition of $\mathit{Reachable}_{\Sigma_R}$, the conclusion follows.

Case 1: Suppose x is a one-state SOL predicate. Since x is a state invariant of Σ_R (premise), i.e., $\forall s : \mathit{Reachable}_{\Sigma_R}(s) \Rightarrow x(s)$, the conclusion follows from the application of Lemma 1.

Case 2: Suppose x is a two-state SOL predicate. Since x is a transition invariant of Σ_R (premise), i.e., $\forall s, s' : (\mathit{Reachable}_{\Sigma_R}(s) \wedge \mathit{Reachable}_{\Sigma_R}(s') \wedge \rho_A(s, s')) \Rightarrow q(s, s')$, the conclusion follows from condition (b) above and by the application of Lemma 1.

7.2 Quotient Automata

Consider the state machine $\Sigma_R = (V, S, \Theta_A, \rho_A)$. Let $V_A = v_1, v_2, \ldots v_k$ be the set of state variables whose names appear in the logical formulae[2] characterizing predicates Θ_A and ρ_A. Each state $s \in S$ is an interpretation of the state variables of V. Let $s(v_1), s(v_2), \ldots s(v_k), \ldots s(v_n)$ denote the interpretation of variables $v_1, v_2, \ldots v_k, \ldots v_n$ in state s. The following equivalence relation $\approx$ on S:

$$\{(s_1, s_2) \mid (s_1(v_1) = s_2(v_1)) \wedge (s_1(v_2) = s_2(v_2)) \wedge \ldots (s_1(v_k) = s_2(v_k))\}$$

can be shown to be a bisimulation on Σ_R. This follows from the definition of a bisimulation relation – by definition, R is a bisimulation relation if R progresses to R itself; therefore, since relation $\approx$ collapses all states that are equal into an equivalence class, transitions emanating from any state in that set must be identical. Therefore $\Sigma_A = (V_A, S_A, \Theta_A, \rho_A)$, the *quotient automaton* with respect to the bisimulation, whose state space S_A is the set of equivalence classes induced by $\approx$, will be a sound and complete abstraction of Σ_R for *all* invariants. Since Σ_R is a sound abstraction of Σ relative to the invariance of all SOL predicates, Σ_A is a sound abstraction of Σ for all SOL invariants. We call this the **General Theorem of Sound Abstractions**.

[2] We assume that these formulae have been simplified to their normal form.

7.3 The SOL Theorem of Sound Abstractions

A corollary of the General Theorem of Sound Abstractions is that by establishing the invariance of an SOL predicate x for *any subset* of a collection of SOL modules, one establishes the invariance of x for the *complete* SOL multi-agent system. The proof of this follows from the observation that the components (corresponding to the dependent variables) of the initial predicate Θ and the transition relation ρ of the state machine described by a SOL module are specified as a conjunction of predicates, each one corresponding to a dependent variable in V. Selection of a subset of SOL definitions (which define the values of dependent variables) amounts to "throwing away" the conjuncts that correspond to the variables whose definitions are being eliminated. Therefore, since

$$(\ldots \wedge \theta_{r_{I+1}} \wedge \theta_{r_{I+2}} \ldots \wedge \theta_{r_k} \ldots \wedge \theta_{r_n}) \Rightarrow (\ldots \theta_{r_{I+1}} \wedge \theta_{r_{I+2}} \ldots \wedge \theta_{r_k})$$
$$(\ldots \wedge \rho_{r_{I+1}} \wedge \rho_{r_{I+2}} \ldots \wedge \rho_{r_k} \ldots \wedge \rho_{r_n}) \Rightarrow (\ldots \wedge \rho_{r_{I+1}} \wedge \rho_{r_{I+2}} \ldots \wedge \rho_{r_k})$$

it follows from the General Theorem of Sound Abstractions that an invariant of a reduced SOL module with dependent variables $v_{I+1}, v_{I+2}, \ldots, v_k$ will also be an invariant of a SOL module with dependent variables $v_{I+1}, v_{I+2}, \ldots v_k, v_{k+1}, \ldots v_n$.

8 Discussion and Related Work

SOL is based on ideas introduced in the Software Cost Reduction (SCR) project [14, 15] of the Naval Research Laboratory which dates back to the late seventies. 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 [13]. SOL includes certain key features of SAL including the notion of *events* and modularity. The notation of SCR has directly or indirectly influenced more recent notations such as Reactive Modules [4]. Whereas the application areas of Reactive Modules are primarily hardware and communication protocols, the focus of SCR and related languages has primarily been on specifying software systems.

Because of infinite or very large state spaces of systems described in SOL, finite-state verification methods such as that of Alur and Henzinger [3] and tools such as Mocha [2] are limited in scope. For verification to succeed, abstraction [12, 8] therefore plays a very important role when applying model checking. Although many researchers routinely apply abstractions during model checking, the abstractions are often applied in an ad-hoc way and are therefore not always sound. In our approach, a systematic application of abstraction guarantees soundness [8]. An important advantage of model checkers is their use for *refutation* since the counterexamples they provide when a check fails servers practitioners as a valuable debugging aid [8].

Theorem proving, the only other viable alternative, has the associated problems of incompleteness (i.e., a property that is not provable may indeed be valid) and lack of user guidance, along the lines of counterexamples provided by model checkers, in case of proof failure. Also, "traditional" theorem proving systems such as PVS [16] require manual effort and mathematical sophistication to use.

The tool SOLver, currently under development, is based on the tool Salsa [9] which uses SAL as the input language. Although a theorem prover, Salsa affords "push-button" automation, ease of use, and counterexample generation that typify model checkers. The results of our experiments with Salsa and a comparison of its performance with those of popular model checkers and the theorem prover PVS are presented in [9]. Since the architecture of SOLver is based on Salsa (see [9] for details), and the algorithms of SOLver are extensions of those in Salsa, we expect SOLver to have comparable performance and to possess similar attributes as Salsa.

9 Conclusion

In this paper we present an integrated formal framework for the specification and analysis of Multi-Agent Systems (MAS). Our framework uses a composition operator the semantics of which are guaranteed to preserve certain individual agent properties. We demonstrate that the formal framework may serve as the basis for various analysis and transformation techniques such abstraction, and verification by model checking or theorem proving. We are currently developing a suite of analysis and transformation tools for SOL based on this framework.

10 Acknowledgements

I thank Connie Heitmeyer and the anonymous referees for their very useful comments on previous drafts of the paper.

References

1. B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.
2. Rajeev Alur et al. Mocha: Modularity in model checking. In *Proc. Computer-Aided Verification, 10th Annual Conf. (CAV'98)*, Vancouver, Canada, 1998. Springer.
3. Rajeev Alur and Thomas A. Henzinger. *Computer Aided Verification: An Introduction to Model Building and Model Checking for Concurrent Systems.* Draft, `www-cad.eecs.berkeley.edu/~tah/CavBook`, 1998.
4. Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.
5. R. Bharadwaj. SINS: a middleware for autonomous agents and secure code mobility. In *Proc. Second International Workshop on Security of, Moble Multi-Agent Systems (SEMAS-02), First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'02)*, Bologna, Italy, July 2002.
6. R. Bharadwaj. SOL: A verifiable synchronous language for reactive systems. In *Proc. Synchr. Languages, Apps., and Programming, ETAPS 2002*, Grenoble, France, April 2002.
7. R. Bharadwaj et al. An infrastructure for secure interoperability of agents. In *Proc. Sixth World Multiconference on Systemics, Cybernetics, and Informatics*, Orlando, Florida, July 2002.

8. R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, 6(1), January 1999.
9. R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Proc.* 6^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000), ETAPS 2000*, Berlin, March 2000.
10. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
11. L. Gong. Java Security: Present and near future. *IEEE Micro*, 15(3):14–19, 1997.
12. 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.
13. 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.
14. K. Heninger, D. L. Parnas, J. E. Shore, and J. W. Kallander. Software requirements for the A-7E aircraft. Technical Report 3876, Naval Research Lab., Wash., DC, 1978.
15. K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE TSE*, SE-6(1):2–13, January 1980.
16. 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.
17. F. B. Schneider. Enforceable security policies. *ACM Trans. Infor. and System Security*, 3(1):30–50, February 2000.
18. E. Tressler. Inter-agent protocol for distributed SOL processing. Technical Report To Appear, Naval Research Laboratory, Washington, DC, 2002.
19. W3C. Simple Object Access Protocol (SOAP) 1.1. Technical Report W3C Note 08, The World Wide Web Consortium, May 2000.