

# Verifiable Middleware for Secure Agent Interoperability<sup>\*</sup>

In Proc. 2<sup>nd</sup> Goddard IEEE Workshop on Formal Approaches to Agent-Based  
Systems (FAABS II)

October 28–30 2002. Greenbelt, MD USA.

Dr. Ramesh Bharadwaj

Center for High Assurance Computer Systems  
Naval Research Laboratory  
Washington DC, 20375  
ramesh@itd.nrl.navy.mil

**Abstract.** There is an increasing need, within organizations such as the Department of Defense and NASA, for building distributed applications that are rapidly re-configurable and survivable in the face of attacks and changing mission needs. Existing methods and tools are inadequate to deal with the multitude of challenges posed by application development for systems that may be distributed over multiple physical nodes separated by vast geographical distances. The problem is exacerbated in a hostile and unforgiving environment such as space where, in addition, systems are vulnerable to failures. It is widely believed that intelligent software agents are central to the development of agile, efficient, and robust distributed applications. This paper presents details of agent-based middleware that could be the basis for developing such applications. We pay particular attention to the correctness, survivability, and efficiency of the underlying middleware architecture, and develop a middleware definition language that permits applications to use this infrastructure in a scalable and seamless manner.

## 1 Introduction

There is an increasing need, both within Government and Industry, for methods and tools to develop highly distributed and robust computer-based systems. Moreover, software-intensive systems in safety- and mission-critical areas, such as software for manned and unmanned space missions within NASA, or the Network-Centric Warfare [6], Total Ship Computing, and FORCEnet initiatives of the Department of Defense (DoD), are of exceedingly high complexity and must in addition be dependable, robust, and adaptive. A recent Department of Defense (DoD) report to Congress [10] identifies the lack of secure, robust connectivity and interoperability as one of the major impediments to progress in Network Centric Warfare.

---

<sup>\*</sup> This work is sponsored by the Office of Naval Research

## 2 Why Software Agents?

It is widely acknowledged that intelligent software agents are central to the development of the capabilities required to write robust, reconfigurable, and survivable distributed applications. This is because agents are an efficient, effective, and survivable means of information distribution and access. Agents are efficient because only relevant information needs to be passed along. Agents are effective because they allow local control over updates and the dissemination of data. Agents are survivable because their control is distributed. This new technology, which includes both autonomous and mobile agents, addresses many of the challenges posed by distribution of applications and is capable of achieving the desired quality of service especially over unreliable, low-bandwidth communication links. However, agents technology carries with it associated security vulnerabilities. Distributed computing in general carries with it risks such as denial of service, Trojan horses, information leaks, and malicious code. Agents 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 their data. Such threats become very real in a distributed computing environment, in which a malicious intruder may be actively trying to disrupt communications.

The Secure Agents Middleware (SAM) is being designed to provide the required degree of trust in addition to meeting a set of achievable security requirements. Such an infrastructure is central to the successful deployment and transfer of agents technology to industry because security is a necessary prerequisite for distributed computing. To make agent-based systems economically viable, it is imperative that their development, upgrade, integration, testing, certification, and delivery be rapid and cost-effective. However, immense and profound challenges of software trustworthiness remain. Methods and tools for software development that are available commercially are not sufficient to meet the challenges posed by the distribution of processing functions, real-time and non-real-time integration, multi-level security, and issues characteristic of COTS products, such as malicious code, viruses, worms, and Trojan horses.

## 3 Requirements for Secure Mobile Agents

Security is a **fundamental concern** in SAM. By building security from the ground up into SAM, we gain efficiency by identifying and dealing with potential bottlenecks early, i.e., at the design state. SAM provides an efficient architecture *and* ensures security by eliminating unnecessary and/or insecure communication among agents and hosts. Our classification of requirements for secure mobile agents is from FGS96<sup>1</sup>. For the initial release of SAM we shall assume a degree of trust among the participants. This is reasonable in a large organization such as the DoD or NASA where it may be assumed that other policing methods and

---

<sup>1</sup> "Security for Mobile Agents: Issues and Requirements," William N. Farmer, Joshua D. Guttman, and Vipin Swarup, The MITRE Corporation, Bedford, MA.

techniques for intrusion detection and tolerance will identify and sift out casual intruders and eavesdroppers or programs carrying malicious payloads. However, we plan to address this very important research issue in greater detail in the later stages of this effort.

This project addresses the following security requirements:

- The author and sender of an agent are authenticated.
- The correctness of an agent’s code is checked.
- Privacy is maintained during transmission by encrypting agent data.
- Hosts protect themselves against malicious agents by first authenticating an agent and checking that its proposed activities are authorized.
- Host safety is ensured by created agents in a language SOL [2] that promotes the development of safe programs.
- Senders have control over their agents, e.g., they may restrict or increase an agent’s authorization in particular situations.
- By equipping each agent with a state appraisal function, hosts can ensure that an agent is always in a safe state.
- Senders have control over which hosts have the authority to execute an agent.

## 4 A Brief Introduction to SOL

Agents are created in a special purpose synchronous programming language called Secure Operations Language (SOL) [2, 4, 1]. A SOL application comprises a set of agent modules, each of which runs on a given host. The host executes an agent module in compliance with a set of locally enforced security policies. A SOL multi-agent system may run on one or more hosts, spanning multiple networks and multiple administrative domains.

A module is the unit of specification in SOL and comprises variable declarations, assumptions and guarantees, and definitions. The **assumptions** section typically 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.

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  $x = \text{initially } init \text{ then } expr$  (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). 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  $\text{if } \{ \ [g_1 \rightarrow expr_1 \ [g_2 \rightarrow expr_2 \dots \ ] \}$  is defined along the lines of Dijkstra’s *guarded commands* [7] – 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,

```

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. 1.** A SOL agent module that implements safe access to local files.

the expression is nondeterministic. It is an error if none of the guards evaluates to `true`, and execution aborts. The *case expression* `case expr { [] $v_1$  → expr1 [] $v_2$  → expr2 ... }` is equivalent to the conditional expression `if { []( $expr == v_1$ ) → expr1 []( $expr == v_2$ ) → expr2 ... }`. The conditional expression and the case expression may optionally have an `otherwise` clause with the obvious meaning.

## 5 Enforcement Automata

In this section, we shall examine how *enforceable* safety and security policies [11] 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.

### 5.1 Security Automata

We use the example from [11] 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 [8] in which one may either always or never allow access to a system resource. As shown in Figure 1, this policy is easily implemented in SOL.

## 6 Formal Semantics of SOL

*State Machines* A SOL agent module describes a state machine [2]. A *state machine*  $\Sigma$  is a quadruple  $(V, S, \Theta, \rho)$ , where  $V = \{v_1, v_2, \dots, v_n\}$  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;  $\Theta : S \rightarrow \text{boolean}$  is a predicate characterizing the set of *initial states*;  $\rho : S \times S \rightarrow \text{boolean}$  is a predicate characterizing the *transition relation*. We write  $\Theta$  as a logical formula involving the names of variables in  $V$ . Predicate  $\rho$  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  $\rho$  as a logical formula involving the values of state variables in the previous state (specified using operator `PREV`) and in the current state.

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

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  $Reachable_{\Sigma}(s)$ ) if

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

*Invariants* A one-state predicate  $p$  is a *state invariant* of  $\Sigma$  if and only if

$$\forall s : Reachable_{\Sigma}(s) \Rightarrow p(s)$$

A two-state predicate  $q$  is a *transition invariant* of  $\Sigma$  if and only if

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

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

*Verification* For a SOL agent module describing a state machine  $\Sigma$ , and a set of SOL predicates  $X = x_1, x_2, \dots$ , verification is the process of establishing that each SOL predicate  $x_i \in X$  is an invariant of  $\Sigma$ .

## 7 SOL Agent Modules

A SOL agent module describes both an agent's environment, which is usually nondeterministic, and the required agent behavior, which is usually deterministic [5, 9]. A SOL agent module describes the required relation between *monitored variables*, environmental quantities that the agent monitors, and *controlled variables*, environmental quantities 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 are dependent on the values of monitored variables. Dependent variables include all the controlled variables and internal variables of an agent module. In the sequel, we assume that variables  $v_1, v_2, \dots, v_I$  are an agent's monitored variables, and that variables  $v_{I+1}, v_{I+2}, \dots, v_n$  are the agent's dependent variables. The notation  $NC(v_1, v_2, \dots, v_k)$  is used as an abbreviation for the SOL predicate  $(v_1 = PREV(v_1)) \wedge (v_2 = PREV(v_2)) \wedge \dots \wedge (v_k = PREV(v_k))$ .

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

$$v_i = \begin{cases} e_1 & \text{if } g_1 \\ e_2 & \text{if } g_2 \\ \vdots \end{cases}$$

where  $1 \leq i \leq n$ , and  $e_1, e_2, \dots$  are SOL expressions, and  $g_1, g_2, \dots$  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.

### 7.1 Composition of SOL Agent Modules

Consider two SOL agent 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

$$\begin{aligned} V &= V_1 \cup V_2 \\ \Theta &= \Theta_1 \wedge \Theta_2 \\ \rho &= \rho_1 \wedge \rho_2 \end{aligned}$$

Each  $s \in S$  maps each  $v \in V$  to its range of legal values

provided that there is no circularity in the occurrences of variables in  $\rho$ . Also in practice, it is the case that  $\rho_1$  and  $\rho_2$  define disjoint sets of state variables.

## 8 Conclusions

We plan to continue the development of design and analysis tools for SOL agents, and verification tools such as automatic invariant generators and checkers, theorem provers, and model checkers. We currently have a compiler for SOL which generates Java code suitable for execution on multiple hosts. Planned extensions to the compiler include support for fine-grained security and problems associated with survivability such as fault-tolerance, load balancing, and self-stabilization.

The goal of the NRL secure agents project is to develop enabling technology that will provide the necessary security infrastructure to deploy and protect time- and mission-critical applications on a distributed computing platform. Our intention is to create a *robust* and *survivable* information grid that will be capable of resisting threats and surviving attacks. One of the criteria on which this technology will be judged is that critical information is conveyed to principals in a manner that is secure, safe, timely, and reliable. No malicious agencies or other threats should be able to compromise the integrity or timeliness of delivery of this information.

## References

1. R. Bharadwaj. SINS: a middleware for autonomous agents and secure code mobility. In *Proc. Second International Workshop on Security of Mobile Multi-Agent Systems (SEMAS-02), First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'02)*, Bologna, Italy, July 2002.
2. R. Bharadwaj. SOL: A verifiable synchronous language for reactive systems. In *Proc. Synchronous Languages, Applications, and Programming, ETAPS 2002*, Grenoble, France, April 2002.
3. R. Bharadwaj. An infrastructure for secure interoperability of agents. Technical report, Naval Research Laboratory, Washington, DC, To appear.
4. 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.
5. R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, 6(1), January 1999.
6. A. K. Cebrowski and J. J. Garstka. Network-Centric Warfare: Its origin and future. In *Proc. United States Naval Institute*, January 1998.
7. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
8. L. Gong. Java Security: Present and near future. *IEEE Micro*, 15(3):14–19, 1997.
9. C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11), November 1998.
10. Secretary of Defense et al. Network centric warfare. Technical report, Department of Defense, [www.c3i.osd.mil/NCW](http://www.c3i.osd.mil/NCW), July 2001.
11. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.