# Functional ìAJAXî in Secure Synchronous Programming

Ramesh Bharadwaj
United States Naval Research Laboratory

# Functional “AJAX” in Secure Synchronous Programming

Ramesh Bharadwaj
Center for High Assurance Computer Systems
Naval Research Laboratory
4555 Overlook Avenue
Washington, District of Columbia 20375-5337
ramesh@itd.nrl.navy.mil

Supratik Mukhopadhyay
Department of Computer Science
West Virginia University
Morgantown, West Virginia 26506-6109
supratik@csee.wvu.edu

## ABSTRACT

AJAX (Asynchronous Javascript and XML) is a combination of technologies aimed at supporting an improved user/application interactivity in the context of web-based service-oriented computing. Based on the XMLHttpRequest API, AJAX provides an engine for handling service invocations asynchronously while interacting with other applications/users in the foreground. The AJAX combination of technologies have already been deployed in popular applications like Google Maps. The adherence to XML-based format for data exchange makes this combination of technologies and similar other frameworks suitable for deployment in service-oriented architectures based on lightweight services (REST or Web) augmenting existing architectures with increased capabilities of interaction.

While AJAX promises improved interaction capabilities, it is also accompanied by its baggage of problems. The lack of formal semantics makes it difficult to understand and validate the functionalities that an application is supposed to provide. The support for individual component technologies of AJAX (e.g., XMLHttpRequest, Javascript etc.) are different for different infrastructures (and browsers). The adoption of Javascript (an interpreted scripting language) makes it inefficient for running heavyweight processes. Besides, the source code must be downloaded by the client for execution which raises concerns in security and intellectual property issues. These problems render the existing AJAX framework unsuitable for deployment in mission-critical enterprise applications.

In this paper, we present an “AJAX”-like framework in an event-driven secure synchronous programming environment. More precisely, we present a synchronous programming programming language called SOL (Secure Operations Language) that has capabilities for handling service invocations asynchronously, strong typing (including dynamic) to ensure enforcement of information flow and security policies and the ability to deal with failures (both benign and byzantine) of components. While our framework provides “AJAX”-like functionalities in a synchronous programming environment, unlike AJAX, it is not a combination of disparate technologies. As opposed to the AJAX framework, it is supported by formal operational semantics. Applications written in our framework can be verified using formal static checking techniques like theorem proving. The framework runs on the top of the SINS (Secure Infrastructure for Networked Systems) infrastructure developed at the Naval Research Laboratory.

## Keywords

AJAX, SOA

*WWW2006*, May 22–26, 2006, Edinburgh, UK.
.

## 1. INTRODUCTION

*Service-oriented architectures* (SOAs) [23] are becoming more and more common as platforms for implementing large scale distributed applications. In an SOA, applications are built by combining services, which are platform independent components running on different hosts of a network. They are now being deployed in mission-critical applications that include space, healthcare, electronic commerce, and military. Client requests are met by on-demand discovery of a set of suitable services which, when appropriately composed, will satisfy the client's service requirements. Delivery of services to clients is governed by *service level agreements* (SLAs) which additionally specify the *quality of service* (QoS) that the service provider needs to guarantee and the appropriate penalties for their violation. QoS constraints that a service provider guarantees may include security, timeliness, and availability. Such guarantees are difficult to satisfy when services are spatially distributed over a network which is subject to active attacks, network congestion, and link delays. Such attacks and failures, coupled with the need for migration of services and clients, poses a formidable challenge in delivering services that meet the SLAs.

SOA's typically are characterized by the high level of interactivity between applications (i.e., between services, brokers, and clients) and users. AJAX [12] (Asynchronous Javascript and XML) is a combination of technologies aimed at supporting an improved user/application interaction in the context of web-based service-oriented computing (or more generally development of web applications). Based on the XMLHttpRequest API, AJAX provides an engine for handling service invocations asynchronously while interacting with other applications/users in the foreground. The AJAX combination of technologies have already been deployed in popular applications like Google Maps. The adherence to XML-based format for data exchange makes this combination of technologies and similar other frameworks suitable for deployment in service-oriented architectures (using a web browser as a client interface) based on lightweight services (REST [15] or Web) augmenting existing architectures with increased capabilities of interaction.

While AJAX promises improved interaction capabilities, it does come with its baggage of problems. The lack of formal semantics makes it difficult to understand and validate the functionalities that an application is supposed to provide. The support for individual component technologies of AJAX (e.g., XMLHttpRequest, Javascript etc: XML Namespaces are not well supported by Internet Explorer) are different for different infrastructures (and browsers). The adoption of Javascript (an interpreted scripting language) makes it inefficient for running heavyweight processes. Besides, the source code must be downloaded by the client for execution which raises concerns in security and intellectual property issues. These problems render the existing AJAX framework unsuitable for deployment in mission-critical enterprise applications where formal guarantees with respect to functionalities and security are of utmost importance.

In this paper, we present an "AJAX"-like framework in an event-driven synchronous programming [4] environment (a' la' LUSTRE [17], SCR [8], and Esterel [5]). More precisely, we present a synchronous programming language SOL (Secure Operations Language) that has capabilities of handling service invocations asynchronously, provides strong typing to ensure enforcement of information flow and security policies, and has the ability to deal with failures (both benign and byzantine) of components. In the synchronous programming paradigm, the programmer is provided with an abstraction that respects the synchrony hypothesis, i.e., one may assume that an external event is processed completely by the system *before* the arrival of the next event. One might wonder how a synchronous programming paradigm can be effective for dealing with widely distributed systems where there is inherent asynchrony. The answer may seem surprising to some, but perfectly reasonable to others: We have shown elsewhere [10] that under certain sufficient conditions (which are preserved in our case) the synchronous semantics of a SOL application are preserved when it is deployed on an asynchronous, distributed infrastructure. While our framework provides "AJAX"-like functionalities in a (functional) synchronous programming environment, unlike AJAX, it is not a combination of disparate technologies. As opposed to the AJAX framework, it is supported by formal operational semantics. The individual modules follow a "publish-subscribe" pattern of interaction while asynchronous service invocations akin to the XMLHttpRequest (API) are provided continuation-passing-based [3] semantics. The design of SOL was heavily influenced by the design of SAL (the SCR Abstract Language), a specification language based on the SCR Formal Model [18]. Applications written in our framework can be verified using formal static checking techniques like theorem proving. We provide both a static and a dynamic type system to ensure respectively (1) static type soundness and (2) to ensure runtime type soundness in the presence of third party (possibly COTS) component services that may undergo reconfigurations at runtime due to network faults or malicious attacks. The framework runs on the top of the SINS (Secure Infrastructure for Networked Systems) infrastructure developed at the Naval Research Laboratory. SINS is built on the top of the Spread toolkit [1] which provides a high performance virtual synchrony messaging service that is resilient to network faults. A typical SINS system comprises SINS Virtual Machines (SVMs), running on multiple disparate hosts, each of which is responsible for managing a set of modules on that host. SVMs on a host communicate with SVMs on other hosts using the secure group communication infrastructure of Spread. SINS provides the required degree of trust for the modules, in addition to ensuring compliance of modules with a set of requirements, including security policies.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 provides a brief description of the SOL language along with several illuminating examples. Section 4 provides a brief description of the SINS platform. Formal semantics of SOL as well as a static type system are provided in Section 5. Section 6 concludes the paper.

## 2. RELATED WORK

Service-based systems (some times identified with web services even though the scope of service-based systems is much broader) have traditionally adopted document-oriented SOAP-based [23] messaging for communicating XML data across a network. SOAP, by default, is bound to the HTTP [11] transport layer. HTTP does not provide any means of correlating requests with responses. SOAP over HTTP provides a basic one-way synchronous communication framework on the top of which other protocols like request/response type RPC [11] can be implemented. SOAP and WSDL specifications are not executable; the protocol adopted by a particular application needs to be supported by the underlying runtime infrastructure. SOAP, as it is, does not support interaction patterns like request/callback, publish/subscribe or asynchronous store and forward messaging. The definition of SOAP can be extended to provide such interaction patterns; such extensions require providing new semantics to an existing system.

In contrast, our framework is based on the synchronous programming language SOL. In SOL, the message passing between modules (henceforth we will use the term agent for module instances) is based on a (push) publish-subscribe. A module listens to those "controlled variables" of another module that it "subscribes to" by including them as its "monitored variables". A module receives the values of its monitored variables as input and computes a function whose output can change the values of its controlled variables. Service invocations (both synchronous and asynchronous) needed to compute the function are dealt uniformly using continuation passing. SOL agents run on the SINS platform which is built on the top of the Spread toolkit that provides guaranteed message delivery and resilience to network faults.

Traditionally, BPEL4WS [14] and semantic web-based frameworks (e.g., OWL-S) corresponding to business processes have been used for describing, modeling and executing workflows for web service-based systems. It is difficult in such frameworks to interoperate in networks involving sensors and other physical devices having complex dynamical behavior, for continuously accessing information, monitoring the environment and reacting to changes in it.

Both BPEL4WS and OWL-S [2] lack satisfactory formal operational semantics - while OWL-S specifications are assigned semantics based on first order logic, such axiomatic semantics are not helpful in building a programming model with formal operational semantics. In particular, both BPEL4WS and OWL-S lack a satisfactory programming model with formal semantics. Lack of a proper programming model with formal operational semantics makes it difficult to understand, predict and debug the behavior of service compositions described in such languages. As a result it is risky to deploy an application based on such a framework in a mission-critical environment.

Both BPEL4WS and OWL-S do not provide an effective framework for modeling non-functional properties/QoS goals such as security (policies), context-aware reconfiguration, time-deadlines, availability, physical/logical migration of processes etc. Security policies can be expressed in the BPEL4WS framework using external plug-ins like WS-Security. Embedding external plug-ins respecting different standards creates problems in interoperability. We believe that security should be provided as an integral component of service-based systems in order to free application developers of security concerns.

Service-based systems executing workflows specified by BPEL4WS (or OWL-S) processes do not provide any techniques for automatically reconfiguring a workflow dynamically in response to fast changing contexts.

In our framework, workflows are expressed as a collection of SOL modules with "proper plumbing" to ensure desired coordination. As already stated above, the SOL language is supported by formal operational semantics. Besides the event-driven publish-subscribe-based interaction between the individual modules make SOL ideal for programming service-based systems that are deployed in networks involving sensors and other physical devices having complex dynamical behavior. The SINS platform provides the required degree of trust for the modules, in addition to ensuring compliance of modules with a set of requirements, including security policies. Dynamic reconfiguration of workflows can be obtained using a "hierarchical plumbing" a' la' [27].

AJAX provides an engine that acts as a client-side brokerage and orchestration point for web services and provides support for the XMLHttpRequest API, XSLT, DOM and Javascript. Calls to the services are handled asynchronously using the XMLHttpRequest API. Responses from the server are handled by Javascript code running at the client end. AJAX can utilize the XML and XSL services available in browsers to provide a highly interactive interface for web services. AJAX libraries are available for platforms such as .NET. In our case, the SOL agents can be directly deployed on the SINS platform that acts as a coordination point for the different agents and the services. We have already outlined the shortcomings of the conventional AJAX framework in the Introduction and have also outlined how our framework ameliorates them. In [25], the authors use a synchronous framework for globally asynchronous designs. However, their framework is more suited to a hardware design environment rather than a large scale distributed computing one.

The nesC [16] programming language at U.C. Berkeley has been designed for programming networked embedded systems. It supports asynchronous calls to components using events to signify the completion of a call. In the polyphonic C# [22] programming language, asynchronous method calls are supported using queues. A set of methods at the server end defines a "chord". A method call is delayed until all methods in the corresponding chord are invoked.

The communicating concurrent processes, the dominant paradigm for distributed application development, has remained unchallenged for almost 40 years. Not only is this model difficult to use for the average developer, but in addition it fails as a paradigm for designing applications that must satisfy critical requirements such as real-time guarantees [21]. Therefore, applications developed using conventional programming models are vulnerable to deadlocks, livelocks, starvation, and synchronization errors. Moreover, such applications are vulnerable to catastrophic failures in the event of hardware or network malfunctions. Here we present an alternative approach. We embed an "AJAX"-like framework in an event-driven synchronous programming environment (a' la' LUSTRE [17], SIGNAL [17] SCR [8], and Esterel [5]). As opposed to other synchronous programming languages like ESTEREL, LUSTRE and SIGNAL, SOL is a synchronous programming language for distributed applications.

Preliminary versions of SOL and SINS have been introduced in [6, 7]. The current paper extends those versions by providing asynchronous service-invocation management functionalities, type systems for safe information down grading and secure information flow. Besides, it provides operational semantics for the SOL language.

## 3. SOL: THE SECURE OPERATIONS LANGUAGE

A *module* is the unit of specification in SOL and comprises of type definitions, flow control rules, unit declarations, unit conversion rules, variable declarations, service declarations, assumptions and guarantees, and definitions. A module in SOL may include one or more *attributes*. The attribute `deterministic` declares the module as being free of nondeterminism (which is checked by the SOL compiler). Attribute `reactive` declares that the module will not cause a state change or invoke a method unless its (visible) environment initiates an event by changing state or invoking a method (service); moreover, the module's response to an environmental event will be immediate; i.e., in the next immediate step. The attribute `continuation` declares that the module will serve as a continuation for some (external) service invocation. Each (asynchronous) external service invocation is managed by a continuation module that receives the response for the invocation and informs the module that originally invoked the service of the response as soon as it arrives. As defined previously, an *agent* is a module instance. In the sequel, we use module and agent interchangeably.

C-style comments are supported – all text between an opening "`/*`" and 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.

"Integer", "Real", "Boolean", and "String" are the built-in data types in SOL. User-defined types as well as enumerated types can be 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. Besides, this section allows the user to declare "secrecy" types (e.g., secret, classified, unclassified etc.) in order to enforce information flow policies and prevent unwanted downgrading of sensitive information from "secret" variables to "public" variables. The flow control rules section provides rules that govern the downgrading/flow of information between variables of different "secrecy" types (e.g., the rule unclassified => classified, signifies that a variable of type unclassified can be assigned to a variable of type classified, i.e., information flow from an unclassified to a classified variable is allowed). The flow control rules can be used to compute the secrecy types of expressions from those of its constituent variables. If not specified in the flow control section, information flow between variables/expressions with different secrecy types is allowed only in the presence of explicit coercion provided by the programmer. These policies are enforced statically by a static type system. The unit declaration section declares units for the physical quantities that the module monitors and manipulates (e.g., lb, kg, centigrade etc.). This section provides conversion (coercion) rules between the different units (e.g., kg=2.2 lb). Units of expressions can be computed from the units of their constituent subexpressions. The variable declaration section for reactive/deterministic modules is subdivided into five subsections. The continuation variable declaration subsection defines continuation variables that will be used for service invocations. There will be one continuation variable for each service invocation in a module. The type "continuation" before a variable designates it as a continuation variable (e.g., continuation cont;). Corresponding to each continuation variable, there will be a continuation module handling the service invocation associated with that variable. The other four subsections declare the "monitored" variables in the environment that an agent monitors, the "controlled" variables in the environment that the agent controls, "service" variables that only external service invocations can update, and, "internal" variables introduced to make the description of the agent concise. The monitored variables section can include failure variables that are boolean variables indicating the failure of other modules (e.g., the declaration `failure boolean I;` declares a boolean variable `I` that will become true if a module named `I` in the environment fails). A variable declaration can specify the unit (declared in the unit declaration section) of the physical quantity that it is supposed to assume values for (e.g., int weight unit lb;). Assignment of a variable/expression with a unit $U$ to a variable with unit $V$ is allowed only if it is specified in the unit conversion rules section. In that case, the value of the variable/expression is converted to the unit $V$ using the corresponding conversion rule before being assigned to a variable with unit $V$. The declaration of a monitored variable can be accompanied by failure handling information that may specify it being substituted in all computations by another monitored variable in case the module publishing it fails (e.g., the declaration `integer x on I y` specifies that the monitored variable `y` should replace the variable `x` if the failure variable `I` corresponding to the module named `I` in the environment is true). The service declarations section declares the methods that are invoked within a module along with the services providing them. It also describes for each method the preconditions that are to be met before invoking the method as well as the post conditions that the return value(s) from the method is/are supposed to respect. The preconditions and postconditions consist of conjunctions of arithmetic constraints as well as type expressions. A type expression is a set of atomic type judgements of the form $x :: T$ where $x$ is a variable and $T$ is a type. These conditions are enforced dynamically under a runtime environment.

The `assumptions` section includes assumptions upon which correct operation of the agent depends. Execution aborts when any of these assumptions are violated by the environment resulting in the failure variable corresponding to that agent to be set to true. The required safety properties of the agent are specified in the `guarantees` section. Variable definitions, provided as functions or more generally relations in the `definitions` section, specify values of internal and controlled variables. A SOL module specifies 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 often distinguish between monitored variables, i.e., variables whose values are specified by the environment, and *dependent variables*, i.e., variables whose values are computed by a SOL module using the values of the monitored variables as well as those returned by the external service invocations. Dependent variables of a SOL module include the controlled variables, service variables, and internal variables. SOL provides type constructors such as *arrays* and *tuples*. In this paper, we shall not elaborate on the *tuple* and *array* constructs of SOL (see [24] for details).

## 3.1 Events

SOL borrows from SCR the notion of *events* [18]. 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 following are the notation for events that can trigger reactive/deterministic modules. The notation `@T(c)` denotes the event "condition **c** became true", `@F(c)` denotes "condition **c** became false" , `@Comp(cont)` denotes that "the result of the service invocation associated with the continuation variable `cont` (i.e., the service invocation in which `cont` was passed) is available", and `@C(x)` the event "the value of expression `x` has changed". These constructs are explained below. In the sequel, `PREV(x)` denotes the value of expression `x` in the *previous state*.

$$\begin{aligned}
\texttt{@T(c)} &\stackrel{\text{def}}{=} \neg\texttt{PREV}(c) \wedge c \\
\texttt{@F(c)} &\stackrel{\text{def}}{=} \texttt{PREV}(c) \wedge \neg c \\
\texttt{@C(c)} &\stackrel{\text{def}}{=} \texttt{PREV}(c) \neq c
\end{aligned}$$

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 `@F` and `@C` constructs are defined along similar lines. The event `@Comp(cont)` is triggered by the environment in which the agent is running and is received as an event by the agent whenever the result of a service invocation is received by the continuation module associated with the continuation variable `cont` that was passed as a continuation while invoking the service. We will define the event `@Comp` in terms of associated continuation modules in Section 3.3.

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., $(\mathtt{a}, \mathtt{b}) \in D_m$) 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**[1]. 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 of a SOL module comprises a set of variable updates and service invocations 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, service, and controlled) variables. In the programmer's view all updates and service invocations of the system are assumed to be synchronous (similar to the Synchrony Hypothesis of languages such as Esterel, LUSTRE, etc. [17]) – 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.

## 3.2 An Automated Therapeutic Drug Monitoring System in SOL

In this subsection, we present a (part of a) skeleton in SOL of a distributed automated therapeutic drug monitoring system in a hospital. We will use this as a running example later in this paper. A scenario of the operation of the system is depicted in Figure 1. A sensor (can be a nurse sitting at a terminal) at a patient's bed in the hospital monitors the patient's vital data (e.g., saturation, heartbeat, blood pressure etc.). As soon as the vital data indicate that the patient's condition is critical, the sensor reports the vital data to the

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

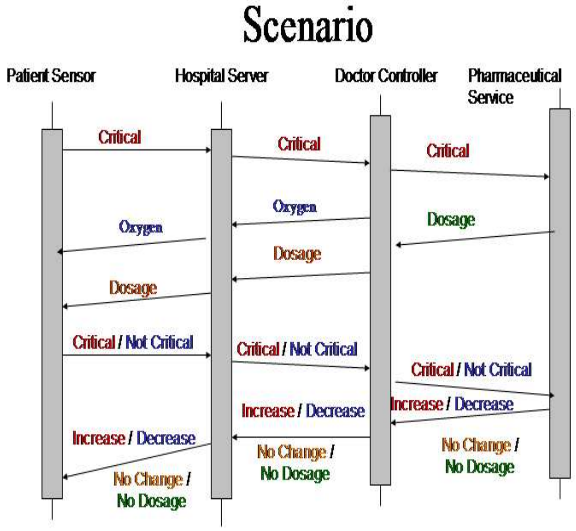

**Figure 1: Automated therapeutic drug monitoring scenario**

central hospital server along with a report on the patient's condition (critical). The central hospital server contacts the patient's doctor (e.g., by sending a message to her palmpilot) with the patient's vital data and the report (critical) from the sensor. The doctor can look up a drug appropriate for the patient's condition and invoke a service provided by the pharmaceutical company (producing the drug), with the vital data of the patient, that computes the correct dosage corresponding to the patient's current state. Further, if the patient's saturation is below a certain threshold, the doctor can order her to be put on oxygen. The doctor communicates her response (dosage, oxygen) to the central hospital server which in turn communicates it to the nurse (patient sensor and actuator) that attends the patient by administering the required dosage of the drug or by putting her on oxygen. The patient sensor (or the nurse) reports to the hospital service whenever the state of the patient changes (e.g., turns from critical to noncritical) which in turn reports to the doctor for appropriate action. Due to space limitations, we show here only the SOL module running on the doctor's palmpilot in Figure 2. The complete therapeutic drug monitoring system consists of SOL modules for the "doctor", the "hospital server" and the "nurse/patient sensor and actuator". The modules translate directly into Java and runs unmodified on the SINS middleware. Interested readers may refer to [17] to compare SOL with other synchronous programming languages such as Esterel, Argos, LUSTRE, and SIGNAL.

The `doctor` module is implemented as a deterministic reactive module. We identify four monitored variables – `heartrate`, `pressure` (unit lb/sqinch), `saturation` and `patient_condition` corresponding to the vital data heart rate, blood pressure and saturation of the patient as well as the condition of the patient (critical or noncritical) that the module obtains from the hospital server. We also identify a service variable `c_dosage` (unit mg) that is defined by invoking the pharmaceutical service, a continuation variable that `cont` that is passed as a continuation while invoking the service, and two controlled variables `output_dosage` (unit cc) and `oxygen` that correspond respectively to the dosage and the de-

```
deterministic reactive module doctor {

 type definitions
  dosage = Integer;
  condition={critical,not_critical};

 units
  lb_per_sqinch, mg, cc;

 unit conversion rules
  mg=cc;

 services
  dosage pharmserv:compute_dosage(x,y,z),
   pre= x::Integer, y::Integer, z:: Integer
   -- post=true;

  monitored variables
   Integer heartrate;
   Integer pressure unit lb_per_sqinch;
   Integer saturation;
   condition patient_cond;

  service variables
   dosage c_dosage unit mg;

  contination variables
   continuation cont;


 controlled variables
   dosage  output_dosage unit  cc;
   Boolean oxygen;

 definitions
// definitions of controlled
//and service variables
    c_dosage = initially null then
       if{
        [] @C(patient_cond) && @C(heartrate)
                      && @C(pressure)
           -> pharmserv:
              compute_dosage(
             heartrate,pressure,
                     saturation)
                          ˆcont;
         }// service invocation

    output_dosage= initially null then
       if{
        [] @Comp(cont)-> c_dosage;
         } //update of controlled variable
    oxygen= initially false then
       if{
        [] @T(saturation<65) -> true;
        [] @T(saturation>90) -> false;
         }
}
```

**Figure 2: Doctor module in SOL.**

cision whether to put the patient on oxygen or not sent back to the hospital server. The hospital server listens to these two controlled variables (among others). We also identify a service invocation `pharmserv:compute_dosage` that invokes the `compute_dosage` method of the pharmaceutical service named (and addressed) `pharmserv` with the vital data of the patient as arguments and the variable `cont` being passed as a continuation. The service invocation is used to obtain the required dosage of the patient and defines the service variable `c_dosage`. The preconditions for invoking the service provided in the `services` section specify that the types of all the three formal parameters `x`, `y` and `z` should be Integer while the postcondition always holds true. The return value from the service invocation should be of type `dosage`. The unit conversion rules section defines an `mg` to be equal to 887 times a `cc` so that the value of the variable `c_dosage` is to be multiplied by 887 (by the runtime environment) before being assigned to the controlled variable `output_dosage`.

The module `doctor` responds to a triggering event[2] by updating its dependent variables in compliance with the dependency (partial) order. One possible order is `oxygen` → `saturation`, `c_dosage`→ `heartrate`, `c_dosage`→ `pressure`, `c_dosage`→ `saturation`, `c_dosage`→ `cont`, and `output_dosage` → `c_dosage`.

## 3.3 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, service, and controlled variable of the module is updated in response to events (i.e., state changes) generated either internally or by the module's environment.

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 informal 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* [13] – in a given state, its value is equivalent to expression $\text{expr}_i$ 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 setting the failure variable corresponding to that module to true. The *case expression* **`case`** *expr* $\{ []v_1 \rightarrow \text{expr}_1 \;\; []v_2 \rightarrow \text{expr}_2 \ldots \}$ is equivalent to the conditional expression **`if`** $\{ [](expr == v_1) \rightarrow \text{expr}_1 \;\; [](expr == v_2) \rightarrow \text{expr}_2 \ldots \}$. The conditional expression and the case expression may optionally have an `otherwise` clause with the obvious meaning.

## 3.4 Service Invocation

A service variable is defined by a one-state or a two-state definition in terms of a service invocation expression (service_invocation). A service invocation expression is of the form `A:B(var_list)ĉont` where the identifier `A` is the name/URL of the service, `B` is the name of the method invoked, `var_list` is the list of variables passed as arguments to the method, and, `cont` is the passed continuation variable. In this case, the service variable depends on the variables in `var_list`. For each service invocation in a module, a distinct continuation variable is used. Internally, corresponding to each continuation variable there is a continuation module handling the result of the service invocation in which the variable is passed. A continuation module has the same structure as the reactive/deterministic ones except that it can have an additional subsection in the variable declaration section: *channel variables*. Channel variables receive values from external services. In addition, it can have another section called *triggers* that lists actions in the environment that the module can trigger. Actions in the trigger section can be defined in the same way as variables. Along with the usual notation for events as in reactive/deterministic modules, a continuation module can have an additional event denoted by `@Rec(Chan)`, where `Chan` is a channel variable, which is triggered as soon as a value is received from an external service on the variable `Chan`. A continuation module for a service invocation is generated internally automatically by the SOL compiler from the SOL definitions and is kept away from the view of the programmer. For example, the continuation module corresponding to the service invocation in Figure 2, where the service variable `c_dosage` (with type dosage) defined by a two-state definition is given below.

```
continuation module cont{
 type definitions
  dosage = Integer;
 controlled variables
  dosage  c_dosage;
 channel variables
  dosage Chan;
 triggers
  Boolean @Comp(cont);
 definitions

 @Comp(cont) =
        if{
         []@Rec(Chan)-> true;
        }

 c_dosage= initially null then
        if{
         []@Rec(Chan)->Chan;
        }
}
```

When the agent `doctor` defining the service variable `c_dosage` is executed, the agent environment invokes the service by sending it a message. The preparation of this message involves marshaling the arguments as well as the continuation, which includes information about the channel `Chan` on which the result of the service invocation is to be returned. Once the service returns the result on the channel `Chan`, the guard `@Rec(Chan)` in the continuation module associated with the continuation variable becomes true. This event results in the controlled variable`c_dosage` (in the continuation module) being set the value received on `Chan` as the response for the service invocation. Also `@Comp(cont)` in the environment gets set to true. In module `doctor`, this in turn sets the value of the service variable `c_dosage` to the value received as the response from the service (i.e., the value of the controlled variable `c_dosage` of the continuation module `cont`) and triggers the event `@Comp(cont)`. The triggering of the event `@Comp(cont)` in the `doctor` module results in the controlled

[2] Since `doctor` 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 | service_invocation | "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 ")" | "@Comp"
                     "("cont_var")" | "@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        : "+" | "−" | "*" | "/"
service_invocation : ID ":"ID"(" var_list ")^cont"

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

**Figure 3: The syntax of SOL definitions.**

variable output_dosage being assigned the value of c_dosage which at that point is the value returned as a response to the service invocation. Note that the invocation of the service can be *a*synchronous, i.e., the response from the service may not arrive instantaneously. Computations that do not depend on the response received from the service invocation (i.e., definitions of dependent variables that do not depend on the service variable receiving the response from the service invocation) are not blocked waiting for the response from the service. For example, in Figure 2, the decision whether to put the patient on oxygen can be made without waiting for the pharmaceutical service to return the required dosage. Hence the definition of the variable oxygen can be executed while waiting for the response from the pharmaceutical service, if one of the events @T(saturation<65) or @T(saturation>90) is triggered. Computations dependent on the result of the service invocation must be guarded by @Comp(cont), where cont is the variable passed as continuation in the service invocation, so that they wait until the result of the service invocation is available (signaled by the triggering of the @Comp(cont) event). The asynchronous nature of the service invocations create the effect of the XMLHttpRequest API in AJAX-like applications.

The asynchronous nature of the service invocations can be used to define a timer. We assume the existence of a timer method provided by a time service that when invoked with a (Integer or Real) delay provides a response after the delay specified by the argument. If the variable timer_cont is passed as a continuation while invoking the service, the triggering of the event @Comp(timer_cont) signifies passage of the delay. The implementation of the timer is illustrated in the example below.

```
deterministic reactive module delay{
 ...
 services
  String time:timer(x),
  pre=x::Integer && x>0 -- post = true;
 controlled variables
  Integer x;
 monitored variables
  Boolean clock;
 service variables
  String t;
 continuation variables
  continuation timer_cont;
 ...
 definitions
 ...
  t=initially null then
  if{
  [] @T(clock)->time:timer(10)^timer_cont;
    }
  x=initially null then
  if{
  [] @Comp(timer_cont)-> ...
    }
}
```

The module `delay` ensures that the controlled `x` is output 10 time units after the arrival of a clock pulse i.e., there is a delay of 10 time units between an input event (arrival of a clock pulse) and the corresponding output.

## 3.5 Failure Handling

Benign failures (we deal with byzantine failures elsewhere) in the environment are handled by program transformations incorporated in the SOL compiler that automatically transform a SOL module based on the failure handling information provided in the monitored variable declaration section. Given the declaration `failure Boolean I` in the monitored variable section of a failure variable signifying the (benign) failure of a module `I` in the environment and the declaration `Integer x on I y` of a monitored variable `x` (`y` is also a monitored variable), the SOL compiler transforms each two-state definition `z=initially null then expr`, where `z` is a dependent variable and `expr` is an expression in which `x` occurs, to

```
z= initially null then
                              if{
                          [] I -> expr[y/x];
                                }
```

where `expr[y/x]` is the expression obtained by replacing each occurrence of the variable `x` by the variable `y`. One-state definitions are transformed similarly.

## 3.6 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 theorem prover such as Salsa [9]. 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.

# 4. SINS

SOL agents execute on a distributed run-time infrastructure called SINS (see Figure 4). A typical SINS implementation comprises one or more SINS Virtual Machines (SVMs), each of which is responsible for a set of agents on a given host. SVMs on disparate hosts communicate using the Agent Control Protocol (ACP) [26] for exchanging agent and control information. An ancillary protocol, termed the Module Transfer Protocol (MTP) manages all aspects of code distribution including digital signatures, authentication, and code integrity. Agents in SOL are allowed access to local resources of each host in compliance with locally enforced security policies. An inductive theorem prover is used to statically verify compliance of an agent with certain local security policies. Other safety properties and security requirements are enforced by observer agents (termed "security agents") that monitor the execution of application-specific agents and take remedial action when a violation is detected.

# 5. THE FORMAL SEMANTICS OF SOL

In this section, we provide the formal semantics of SOL. We first present a static type system that enforces the information flow policies ensuring safe downgrading of information.

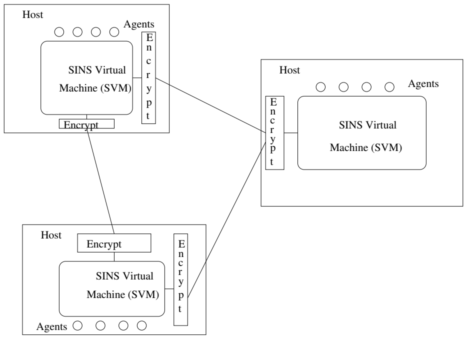

**Figure 4: Architecture of SINS.**

## 5.1 Static Type Checking for Information Flow

Let $\mathcal{S}$ denote a typing environment, $x, y$ range over the variables of a module, $expr$ over the set of expressions in the module, $s, t$ over the set of types defined in the type definition section of the module, and $u, v$ over the set of units defined in the unit definition section of the module. A typing environment $\mathcal{S}$ is defined as

$$\mathcal{S} \ ::= \ \emptyset \mid \mathcal{S} \cup \{x \mapsto t \ unit \ u\}$$

where $x \mapsto t \ unit \ u$ denotes that $x$ is of type $t$ an unit $u$. Here the unit qualifier is optional. Let us define $\mathcal{S}(x) = t$ if $x \mapsto t \ unit \ u \in \mathcal{S}$ or $x \mapsto t \in \mathcal{S}$, and $\mathcal{S}_{unit}(x) = u$ if $x \mapsto t \ unit \ u \in \mathcal{S}$. We will write $\mathcal{S} \vdash defn$ if the definition $defn$ is well-typed under the typing environment $\mathcal{S}$. The typing rules for the static type system for SOL is given in Figure 5. The judgements [type] and [unit] are obvious. The judgement [expr] is infers the secrecy type of an expression from those of its subexpressions ($op$ is a binary operator/relation symbol). If under the typing environment $\mathcal{S}$, the secrecy types of the expressions $expr_1$ and $expr_2$ are $t$ and $t'$ respectively, and $t \rightarrow t'$ is a flow conversion rule (i.e., belongs to $FlowRules$), then the secrecy type of the expression $expr_1 \ op \ expr_2$ is $t'$. Informally, the rule states that, if binary operation/relation is applied on values, one of which is classified and the other unclassified, then the secrecy type of the result is still classified. The judgements [expru1], [PREV1], [PREV2], and [if] are straightforward. In [if], $if(expr, expr_1, expr_2)$ denotes the if expression `if []expr ->` $expr_1$ `otherwise ->` $expr_2$. The judgement [expru2] states that if under the typing environment $\mathcal{S}$, the expressions $expr_1$ and $expr_2$ have units $u$ and $v$ respectively, then a binary operation can be applied on the expressions if there exists a conversion rule from the unit $u$ to the unit $v$ (or viceversa) declared in the unit conversion rules section of the module (here $e(v)$ is an expression containing $v$). In case $u$ is defined in terms of $v$, the unit of the resultant expression will be $v$. The judgements [odeft], [odefu], [tdeft], and [tdefu] provide the type and unit checking rules for one-state definitions and two-state definitions respectively. We explain [odeft]; the others are similar. Intuitively the rule [odeft] states that the value of an unclassified expression can be assigned to a variable declared as classified. More formally, under the typing environment $\mathcal{S}$, the value of an expression of type $t'$ can be assigned to a variable of type $t$ only if it is permitted by a rule in the flow conversion section. Finally, the judgements [onecast] and [twocast] state that an assignment of an expression of type $t'$ to a variable of type $t$ is allowed if explicitly coerced by the programmer. A module $m$ typechecks if $decl \cup UnitRules \cup FlowRules \vdash m$ where $decl$ is the set of declarations in the module, $FlowRules$ is the set of flow control rules and $UnitRules$ is the set of unit conversion rules. A module $m$ is secure if it typechecks.

## 5.2 Formal Operational Semantics

$$\text{[type]}\ \frac{}{\mathcal{S} \vdash x :: t}\ \text{if } \mathcal{S}(x) = t$$

$$\text{[unit]}\ \frac{}{\mathcal{S} \vdash x\#u}\ \mathcal{S}_{unit}(x) = u$$

$$\text{[expr]}\ \frac{\mathcal{S} \vdash expr_1 : t \qquad \mathcal{S} \vdash expr_2 :: t'}{\mathcal{S} \vdash expr_1\ op\ expr_2 :: t'}\ t \Rightarrow t' \in \mathit{FlowRules}$$

$$\text{[expru1]}\ \frac{\mathcal{S} \vdash expr_1\#u \qquad \mathcal{S} \vdash expr_2\#u}{\mathcal{S} \vdash expr_1\ op\ expr_2\#u}$$

$$\text{[expru2]}\ \frac{\mathcal{S} \vdash expr_1\#u \qquad \mathcal{S} \vdash expr_2\#v}{\mathcal{S} \vdash expr_1\ op\ expr_2\#v}\ u = e(v) \in \mathit{UnitRules}$$

$$\text{[PREV1]}\ \frac{\mathcal{S} \vdash expr\#u}{\mathcal{S} \vdash PREV(expr)\#u}$$

$$\text{[PREV2]}\ \frac{\mathcal{S} \vdash expr :: t}{\mathcal{S} \vdash PREV(expr) :: t}$$

$$\text{[if]}\ \frac{\mathcal{S} \vdash expr_1 :: t \qquad \mathcal{S} \vdash expr_2 :: t'}{\mathcal{S} \vdash if(expr, expr_1, expr_2) :: t'}\ t \Rightarrow t' \in \mathit{FlowRules}$$

$$\text{[odeft]}\ \frac{\mathcal{S} \vdash x :: t \qquad \mathcal{S} \vdash expr :: t'}{\mathcal{S} \vdash \mathit{defn}(x, expr)}\ t' \Rightarrow t \in \mathit{FlowRules}$$

$$\text{[odefu]}\ \frac{\mathcal{S} \vdash x\#u \qquad \mathcal{S} \vdash expr\#v}{\mathcal{S} \vdash \mathit{defn}(x, expr)}\ u = e(v) \in \mathit{UnitRules}$$

$$\text{[tdeft]}\ \frac{\mathcal{S} \vdash x :: t \qquad \mathcal{S} \vdash init :: t'', \mathcal{S} \vdash expr : t'}{\mathcal{S} \vdash \mathit{twodefn}(x, init, expr)}\ t' \Rightarrow t, t'' \Rightarrow t \in \mathit{FlowRules}$$

$$\text{[tdefu]}\ \frac{\mathcal{S} \vdash x\#u \qquad \mathcal{S} \vdash init\#w, \mathcal{S} \vdash expr\#v}{\mathcal{S} \vdash \mathit{twodefn}(x, init, expr)}\ u = e(v), u = f(w) \in \mathit{UnitRules}$$

$$\text{[onecast]}\ \frac{\mathcal{S} \vdash x :: t}{\mathcal{S} \vdash \mathit{defn}(x, (t)expr)} \qquad \text{[twocast]}\ \frac{\mathcal{S} \vdash x :: t}{\mathcal{S} \vdash \mathit{twodefn}(x, (t)init, (t)expr)}$$

**Figure 5: A static type system for SOL**

In this section, we provide (a part of) the formal operational semantics of SOL. Let $\Gamma$ be an environment. Let *Types* be the set of all types in a SOL program. We let $val$ range over values of type $T$ for $T \in TYPES$. Let $x, y$ range over the variables in SOL program. An environment $\Gamma$ is defined as

$$\Gamma \ ::= \ \emptyset \mid \Gamma \cup \{x \mapsto val\} \mid \Gamma \cup \{PREV(x) \mapsto val\}$$

where $x \mapsto val$ denotes that $x$ assumes value $val$. We will write $\Gamma \vdash x \mapsto val$ if $x \mapsto val \in \Gamma$. We will denote by $dom(\Gamma)$ the set $\{x \mid \{x \mapsto \ldots\} \subseteq \Gamma\}$. Let us write $\Gamma_{unit}(x) = u$ if $u \in dom(\Gamma)$ and the unit of $x$ is $u$. For a module $m$, we denote by $\Gamma_m$ the restriction of $\Gamma$ to the variables in $m$. The judgements for the operational semantics of SOL are given in Figure 6. For sake of brevity, we do not include the full operational semantics; rather we only provide a sampling of some of the more informative rules. The first judgement [no action] states that for a module $m$ if no monitored variable changes, then no computation is done. Here $MV_m$ denotes the set of all monitored variables of $m$. The second judgement [unit conv] shows how unit conversion is done automatically at runtime. If the unit of an expression is $v$, under the current environment its value is $val$, and a variable $x$ of unit $u$ is assigned the value of the expression, then the value $val$ is first transformed to unit $u$ using the rules in the unit conversion section of concerned module before assignment to $x$. The third judgement [CV] states that if $x$ is a controlled variable in the module $m$ and a monitored variable in the module $n$ and if $x$ has value $val$ under the environment $\Gamma_m$ then it has the same value under the environment $\Gamma_n$ (here $CV_m$ is the set of controlled variables of the module $m$). The judgement [@Comp] describes the $@Comp$ event. Assume that $x$ is an internal or a controlled variable defined by an expression. Assume also that the definition is guarded by $@Comp(cont)$ where *cont* is a continuation variable. If under the current environment $\Gamma$, the expression in the definition evaluates to $val$ and $@Comp(cont)$ is true, then the variable $x$ evaluates to $val$ and the environment turns off $@Comp(cont)$. The other rules that we

$$[\text{no action}]\ \frac{\Gamma \vdash \forall x \in MV_m PREV(x) = x}{}\ \forall cont \Gamma \nvdash @Comp(cont)$$

$$[\text{unit conv}]\ \frac{\Gamma \vdash expr \mapsto val \qquad \Gamma_{unit}(expr) = v, \Gamma_{unit}(x) = u}{\Gamma \vdash x \mapsto e(val)}\ u = e(v) \in UnitRules, def(x, expr) \in m$$

$$[\text{CV}]\ \frac{\Gamma_m \vdash x \mapsto val \qquad x \in CV_m}{\Gamma_n \vdash x \mapsto val}\ x \in MV_n$$

$$[\text{@Comp}]\ \frac{\Gamma \vdash expr \mapsto val \qquad \Gamma \vdash @Comp(cont) \mapsto true,}{\Gamma[@Comp(cont) := false] \vdash x \mapsto val}\ def(Comp(cont), x, expr) \in m$$

**Figure 6: Operational Semantics for SOL**

do not present here deal with checking the preconditions of a service before a service invocation, checking the postconditions after a service has responded and dealing with the external events. Note that at runtime the preconditions and postconditions of a service invocation (including types) are checked to ensure soundness in the presence of third party (possibly COTS) component services that may undergo reconfigurations at runtime due to network faults or malicious attacks.

# 6. CONCLUDING REMARKS

SOL is based on ideas introduced in the Software Cost Reduction (SCR) project [19, 20] of the Naval Research Laboratory which dates back to the late seventies. The design of SOL was directly influenced by the sound software engineering principles in the design of SAL (the SCR Abstract Language), a specification language based on the SCR Formal Model [18]. SOL provides a functionality akin to the XMLHttpRequest framework while maintaining a formal setting.

The goal of SINS is to provide an infrastructure for deploying and protecting time- and mission-critical applications on a distributed computing platform, especially in a hostile computing environment, such as the Internet. The criterion on which this technology should be judged is that critical information is conveyed to principals in a manner that is secure, safe, timely, and reliable.

# 7. REFERENCES

[1] E. Amir and J. Stanton. The spread wide area group communication system. Technical report, Johns Hopkins University, 1998.

[2] G. Antoniou and F. van Harmelen. *A Semantic Web Primer*. MIT Press, 2004.

[3] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[4] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[5] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. of Computer Prog.*, 19, 1992.

[6] R. Bharadwaj. Development of dependable component-based applications. In *In Proceedings of the First International Symposium on Leveraging Applications of Formal Methods (ISOLA)*. IEEE Computer Society, 2004.

[7] R. Bharadwaj. Development of dependable component-based distributed applications. Technical report, Naval Research Laboratory, 2005.

[8] R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Softw. Engg.*, 6(1), Jan. 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, Mar. 2000.

[10] R. Bharadwaj and S.Mukhopadhyay. From synchrony to sins. Technical report, West Virginia University, 2005.

[11] K. P. Birman. *Reliable Distributed Systems*. Springer, 2005.

[12] D. Crane, E. Pascarello, and D. James. *Ajax in Action*. Manning, 2005.

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

[14] F. C. et. al. *Business Process Execution Language for Web Services*. IBM, 2002.

[15] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2000.

[16] D. Gay, P. Levis, J. R. von Behren, M. Welsh, E. A. Brewer, and D. E. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI*, pages 1–11, 2003.

[17] N. Halbwachs. Delay analysis in synchronous programs. In C. Courcoubetis, editor, *the International Conference on Computer-Aided-Verification*, volume 697 of *LNCS*, pages 333–346. Springer-Verlag, 1993.

[18] 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.

[19] 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.

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

[21] E. A. Lee. Absolutely positively on time: What would it take? *Computer*, 38(7):85–87, 2005.

[22] G. Neumann and U. Zdun. Pattern-based design and implementation of an xml and rdf parser and interpreter: A case study. In *ECOOP*, pages 392–414, 2002.

[23] E. Newcomer. *Understanding Web Services*. Addison Wesley, 2002.

[24] F. Rocheteau and N. 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.

[25] J.-P. Talpin, P. L. Guernic, S. K. Shukla, R. K. Gupta, and F. Doucet. Polychrony for formal refinement-checking in a system-level design methodology. In *ACSD*, pages 9–19, 2003.

[26] E. Tressler. Inter-agent protocol for distributed SOL processing. Technical Report To Appear, Naval Research Laboratory, Washington, DC, 2002.

[27] S. S. Yau, S. Mukhopadhyay, and R. Bharadwaj. Specification, analysis, and implementation of architectural patterns for dependable software systems. In *IEEE WORDS*, 2005.