Virtual Integration of Software Intensive Systems by Architectural Modelling (VISISAM)

Dr. Ramesh Bharadwaj
Computer Engineer (Code 5546)
Center for High Assurance Computer Systems
Naval Research Laboratory
Washington DC 20375
USA

Ramesh.Bharadwaj@nrl.navy.mil

ABSTRACT

Although interest in architectural definition languages (ADLs) and architectural frameworks (AFs) is fairly widespread, there is little consensus on how these languages and frameworks can help manage system complexity, or effectively model complex system interactions, in order to mitigate risks and reduce overall system development costs. ADLs provide a notation for describing and reasoning about the relationships and interactions between systems and their sub-systems, in diverse fields such as avionics and tactical weapons. However, most modelling notations currently lack a rigorous formal semantics, absent which they merely serve as (poor) documentation aids. Also, absent the semantics, tools cannot be built to analyze or predict the behaviour of systems based on their architectural descriptions in these languages. In this paper, we explore tools and methods for the construction of models in architectural description languages that may be analyzed, validated, verified, simulated, and eventually deployed automatically on a distributed infrastructure.

1.0 INTRODUCTION

Model-driven development (MDD) is an approach to software development in which comprehensive system models are created *before* writing a single line of source code. However, the research challenge of MDD is whether it is possible to glean useful information from these models, i.e., whether it is possible to automatically analyze the models to predict system behaviour; in particular, to determine whether the system built as specified will have the required properties. If so, the associated argument may be viewed as a proof that the system, as specified, necessarily exhibits these properties; otherwise, users should be presented with a use-case (also known as a counterexample) that demonstrates a scenario in which the required system properties do not hold. In such an approach, analysts, systems engineers, and software developers are given the flexibility of scrutinizing the system at different levels of abstraction; e.g., visual notations that provide an architectural overview of the system, or formal specification notations that facilitate communication with systems engineers, prove properties, or be transformed into a form that may be compiled and executed on a distributed platform. Whereas agile programming methods are code-centric, the model-driven approach instead focuses on developing high level artefacts with sufficient fidelity that are efficiently transformed into executable code.

2.0 BACKGROUND

In order to meet current systems engineering challenges such as pervasive and ubiquitous computing [21], one has to adopt model-based approaches to the development of distributed applications. One answer to the system integration problem is the use of the synchronous paradigm for system and sub-system integration and coordination, where developers are provided with an abstraction that respects the synchrony hypothesis, i.e., they may assume that an external event is processed completely by the system *before* the arrival of the next event.

Based on the synchronous model, the Secure Operations Language (SOL) [6] we use in this paper is designed for the verification, validation, and integration of high assurance systems. The utility of SOL hinges upon the fact that it is a *verifiable* language. Programs in SOL are amenable to fully automated static analysis techniques—such as automatic theorem proving using decision procedures [10], theorem proving [22, 23, 27], or model checking [9, 19]—to ensure compliance of a system with application specific requirements. For a detailed treatment of the language SOL and its formal semantics, the reader is referred to [6, 7, 8]. In the following section, we take a brief look at SOL and its formal semantics.

2.1 SOL

A *module* is the unit of specification in SOL, which comprises variable declarations, assumptions and guarantees, and definitions. An *agent* is a module instance. In the sequel, we use the terms module and agent interchangeably. 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. 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. We define *dependent variables* as the set of controlled variables and internal variables of an agent. SOL borrows from SCR the notion of events [20]. Informally, an *event* denotes a change of state, i.e., an event is said to occur when a state variable changes value. SOL is event-driven and the language, therefore, includes special notation for denoting events. 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*.

@T(c) = not PREV(c) and c

@F(c) = PREV(c) and not c

@C(x) = PREV(x) ≠ x

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 evaluates to *true*. Formally, a *conditioned event* defined as

@T(c) WHEN d = not PREV(c) and c and 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. This provides users the ability to implement *security automata*, a special class of Büchi automata that accept safety and liveness properties [1, 30].

A variable is defined in SOL either as a *one-state* or a *two-state* predicate. A one-state definition, of the form:

x = 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 expression

(where expression may be 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).

3.0 SOL EXTENSIONS FOR ARCHITECTURAL MODELING

In this paper, we extend SOL to reason about functional and non-functional attributes such as fault tolerance, security, and real-time requirements of applications. Formal architectural models defined via extended SOL can then be instantiated and automatically compiled to a group of agents running on multiple processors in a distributed simulation environment such as the one described in [8].

In extended SOL, we assume that any agent A has available a number of attributes that can be accessed for defining the architectural models:

A.Exception: A Boolean variable giving the failure status of module A.

A.Monitored: A generic tuple $[m_1, m_2, \ldots, m_k]$ representing the monitored variables of module A.

A.MonitoredType: A generic type, which is the type of A.Mon.

A.Dependent: A generic tuple $[d_1, d_2, \ldots, d_n]$ representing the remaining (dependent) variables of module A.

A.DependentType: A generic type, which is the type of A.Dep.

A.Controlled: A generic tuple $[c_1, c_2, \ldots, c_n]$ representing the controlled variables of module A.

A.ControlledType: A generic type, which is the type of A.Con.

We clarify the use of module attributes with a simple example in Fig 1.

```
deterministic reactive module Average  raises fail {

monitored variables
    real x, y;
    integer z;

internal variables
    real temp;

controlled variables
    real sum, avg;

assumptions
InRange = (x + y + z <= 100) else fail;

definitions
    temp = x + y;
    sum = temp + z;
    average = sum/3;
}
```

Figure 1: A simple example in extended SOL.

For this module we have the following interpretations for its generic module attributes:

```
Average.Exception = [fail]
Average.Monitored = [x, y, z]
Average.MonitoredType = [real, real, integer]
Average.Dependent = [temp, sum, average, fail]
Average.DependentType = [real, real, real, Boolean]
Average.Controlled = [sum, average]
Average.ControlledType = [real, real]
```

In the most general case, the function computed by a deterministic module A could depend on the current values of its monitored variables (of MonitoredType) as well as the module's state history[1]. Since a module's state history can be stored in an internal variable of type "list of state," we may consider the most general function computed by a SOL module as having the following signature:

A.MonitoredType, A.DependentType, A.MonitoredType → A.DependentType

more precisely, if we designate the function computed by deterministic SOL module A as F then we assume that:

A.Dependent = F (PREV(A.Monitored), PREV(A.Dependent), A.Monitored)

[1] A module's state is a mapping of each element of A.Monitored and A.Dependent to a type correct value. A state history is the sequence of all states since the initialization of the module.

i.e., function F maps module A's previous state

    [PREV(A.Monitored), PREV(A.Dependent)]

and its new inputs A.Monitored to new values of A.Dependent. However, this specification is not complete as it does not account for the module's initial state, so we posit a function

    I : MonitoredType → DependentType

specifying initial values for A.Dependent. We can prove that the generic function computed by module A is:

```
A.Dependent = INITIALLY  I ( A.Monitored) THEN
                    F (PREV(A.Monitored, PREV(A.Dependent), A.Monitored)
```

More generally, we can show that *any* deterministic SOL module A computes a function of this form. For reasons of brevity, we omit details of the proof in this paper.

We emphasize that agents imported (or specified) by an architectural model need not be compilations of modules written in SOL. The imported agents may be any Commercial off the Shelf (COTS) hardware or software components that compute a function of the appropriate generic signature and satisfy the appropriate assumptions on their monitored variables and guarantees on their controlled variables. In fact, an architectural description in SOL may be viewed as a contract obligated upon a component or sub-system that needs to be honored for the correct behavior of the system as a whole. This insight is the key enabler for the virtual integration of (yet to be implemented) software modules or sub-systems into an overall system-of-systems verification framework.

4.0 FORMAL VERIFICATION

Proofs of invariants were carried out via the standard induction technique (see [26]) where a property P is (1) shown to hold in the initial state and (2) assuming that it holds in any state, show that it holds in the next state (using the module's transition relation), while using previously proved or assumed invariants as auxiliary lemmas. The main difficulty in performing such proofs is the discovery and proof of appropriate auxiliary lemmas. One means of discovering such auxiliary lemmas is to examine proof dead ends of a theorem proving system such as PVS [22, 23], which correspond to a consecutive pair of states for which the property is false.

Alternatively, we use the automatic invariant checking tool Salsa [10] by providing a suitable instantiation of the module with simpler functions for I and F, and by using the same auxiliary lemmas used in the theorem prover. All proofs using Salsa are automatic except for the introduction of auxiliary lemmas—once the auxiliary lemmas are formulated, the tool achieves push button automation. The tool either responds that the property under consideration is an invariant, or if not, provides a state pair (similar to a proof dead-end or counterexample) for which the property does not hold. This would indicate that the process of proving architectural models would be considerably simplified by using the more engineer friendly tool Salsa. For most practical models, it is conjectured that once the proof of a more specific module instance is generated using Salsa, it should be straight forward to reuse the same auxiliary lemmas for the more general case with arbitrary modules having generic attributes, using a theorem proving system such as PVS that supports this level of genericity. For future work, we plan on extracting specifications in SOL from artifacts derived from commonly used architectural description languages such as AADL, NAF, or DoDAF (see Section 6.0 below).

5.0 RELATED WORK

Several languages for describing architectural models at various levels of abstraction exist, ranging from UML [28] to languages for describing hardware and software architectures such as the Architecture Analysis and Design Language (AADL), SpecC, and Esterel [15,16,17]. However none of these languages has a well-defined formal operational or denotational semantics, which are necessary for the development of architectural specifications with verifiable properties. Current models for the development of distributed systems [11, 29] are difficult to adopt by the average systems engineer; further, in these paradigms, each application has to explicitly deal with non-functional issues such as reliability, security, and fault tolerance. The use of synchronous languages such as LUSTRE [17], Reactive Modules [2], SCR [9], and Esterel [5] for

systems development is becoming widespread. The use of the synchronous programming language LUSTRE for embedded systems running on tightly-coupled processors is illustrated in [12]. However, since their approach is based on a timed triggered architecture [24], it is unsuitable for applications that are deployed on loosely coupled processors connected over local or wide area networks. Also, the underlying clocked semantics of their implementation has two weaknesses: (1) The semantics of composition and algorithms for distribution executions are highly complex [13], and therefore not easily amenable to formal analysis. (2) The resulting implementation is highly inefficient since at every clock tick all updates to all variables are distributed to all recipients. Since SOL is based on a lazy event-based semantics [7, 8], communication between SOL agents is relatively efficient because updates to a variable are communicated *only* when they result in a change in its value. The work of [31] on compositional verification has a similar approach to our work in simplifying the specification of temporal properties by limiting use of temporal logic to a few intuitive operators. In contrast, their approach applies to an interleaving model with communication via message queues, and uses model-checking for verification. The trade-off with model-checking vs. theorem proving is that model-checking does not require the discovery of auxiliary invariants, but does require equally difficult abstraction/ refinement techniques that are often done in an ad hoc manner [9].

6.0 CONCLUSION

Building robust and secure distributed systems poses a multitude of challenges in defence: Software running on a distributed infrastructure must take into account changing situations such as network and node failures, transient faults, changes in network topology and bandwidth, congestion, and latency. Moreover, today's systems are built using highly reusable hardware and software components using the so called "system of systems" approach. Most defence systems being built today are the integration of highly disparate components that interact with one another over an interconnection fabric or a middleware infrastructure. Some of these system components may be Commercial off the Shelf (COTS), which may have been developed without taking into account the requirements of the system in which they are to be integrated and deployed. Further, during the design of a sub-system, consideration of non-functional requirements such as reliability may complicate the overall design. Therefore, satisfying certain system requirements, such as fault tolerance, is often deferred to the later stages in the development cycle, i.e., during system integration. This increases project risk. Also, any system rework that has to be undertaken is at greatly increased cost. The goal of this project is to help mitigate some of the system development risks by providing early feedback to systems engineers and software developers on certain desired properties of overall system behaviour even before all its sub-systems are built or even completely specified. This approach provides more rigor to the system development process, reduces risk, and helps mitigate unforeseen problems that may otherwise be evident only after the system has been fielded.

Of the existing architectural modelling frameworks, the Architecture Analysis and Design Language (AADL) and its extensions, the NATO Architecture Framework (NAF), as well as their ancestral standards: The

Department of Defence Architecture Framework (DoDAF) and The Ministry of Defence Architecture Framework (MoDAF) are most widespread. For these frameworks to be effective in a model-driven development approach to systems development, a number of research challenges remain. In future work we propose to execute the following tasks to address some of these challenges: (i) provide a formal semantics for a working subset of a widely used architectural definition language; (ii) automatically generate analytical models in SOL to effectively reason about certain aspects of these models, such as timing, reliability, and security, and (iii) develop methods and tools for formal reasoning about the artefacts extracted from an identified architectural framework towards automated proof and counterexample generation.

7.0 REFERENCES

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

[2] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in Sys. Design*, 15:7–48, July 1999.

[3] Y. Amir and J. Stanton. The SPREAD wide area group communication system. Technical report, Johns Hopkins University, Baltimore, MD, 1998.

[4] A. Avizienis. The methodology of N-version programming. In M. R. Lyu, editor, *Software Fault Tolerance*, pages 23–46. John Wiley & Sons Ltd., 1995.

[5] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[6] R. Bharadwaj. SOL: A verifiable synchronous language for reactive systems. In *Proc. Synchronous Languages, Applications and Programming*, Grenoble, France, April 2002.

[7] R. Bharadwaj. Verifiable middleware for secure agent interoperability. In *Proc. Second Goddard IEEE Workshop on Formal Approaches to Agent-Based Systems*, Greenbelt, MD, October 2002.

[8] R. Bharadwaj. A framework for the formal analysis of multiagent systems. In *Proc. Formal Approaches to Multi-Agent Systems*, Warsaw, Poland, April 2003.

[9] R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, 6(1):37–68, January 1999.

[10] R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Proc. TACAS 2000*, pages 378–394, Berlin, Mar. 2000.

[11] F. Buschmann et al. *Pattern-Oriented Software Architecture*. John Wiley & Sons, UK, 1996.

[12] P. Caspi et al. From simulink to SCADE/lustre to TTA: A layered approach for distributed embedded applications. In *Proc. 2003 ACM SIGPLAN conference on Language, compiler, and tools for embedded systems*, pages 153–162, San Diego, CA, 2003.

[13] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks fo processors. *IEEE Trans. Software Engineering*, 25(3):416–427, 1999.

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

[15] D. D. Gajski et al. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Boston/Dordrecht/London, 2000.

[16] D. Garlan. Research directions in software architecture. *ACM Computing Surveys*, 27(2):257–261, 1995.

[17] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[18] C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for constructing requirements specifications: The SCR toolset at the age of ten. *Comput. Syst. Science and Engg.*, 5:19– 35, Jan. 2005.

[19] 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):927– 948, November 1998.

[20] 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, 1996.

[21] T. Hoare and R. Milner. Grand challenges in computing – research. Technical report, British Computer Society, Wiltshire, UK, 2005.

[22] R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc. Sixth ACM SIGSOFT Symp. on Foundations of Software Engineering*, Nov. 1998.

[23] R. D. Jeffords and C. L. Heitmeyer. An algorithm for strengthening state invariants generated from requirements specifications. In *Proc. of the Fifth IEEE International Symposium on Requirements Engineering*, Aug. 2001.

[24] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[25] S. Y. Liao. Towards a new standard for system-level design. In *Proc. Eighth International Workshop on Hardware/ Software Co-design*, 2000.

[26] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, NY, 1995.

[27] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer- Calvert. The PVS system guide, language reference, and prover guide, version 2.4. Technical report, Computer Science Lab, SRI International, Menlo Park, CA, Nov. 2001.

[28] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1995.

[29] D. Schmidt et al. *Pattern-Oriented Software Architecture (Volume 2)*. John Wiley & Sons, Chichester, UK, 2000.

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

[31] F. Xie and J. C. Browne. Verified systems by composition from verified components. In P. Inverardi, editor, *Proc. Joint 9th Eur. Softw. Eng. Conf. (ESEC) and 11th SIGSOFT Symposium on Foundations of Software Engineering (FSE-11)*, pages 277–286, Helsinki,Finland, Sept. 2003.

[32] S. S. Yau, S. Mukhopadhyay, and R. Bharadwaj. Specification, analysis, and implementation of architectural patterns for dependable software systems. In *Proc. WORDS 2005*, Sedona, AZ, Feb. 2005.