Extended Abstract: Formal Verification of Architectural Patterns in Support of Dependable Distributed Systems [*]

Ralph Jeffords and Ramesh Bharadwaj
Center for High Assurance Computer Systems, Naval Research Laboratory, Washington, DC USA
{jeffords,ramesh}@itd.nrl.navy.mil

Keywords: Architectural patterns, dependable software, component-based development, formal verification.

1. Introduction

Building robust, secure distributed systems in the presence of transient faults, node failures, and changes in network topology poses a multitude of challenges. Most systems being built today are the integration of highly disparate hardware and software components that interact via a hardware bus or middleware infrastructure. During the design of a component, non-functional requirements such as fault-tolerance may complicate the design and are better addressed during hardware/software integration by altering the run-time behavior of components. Architectural patterns, analogous to design patterns, are a means to develop such mechanisms rapidly by reusing existing solutions.

To meet current engineering challenges such as pervasive and ubiquitous computing [8], one must adopt model-driven approaches to build distributed applications. We propose the synchronous paradigm for component integration and coordination: developers use an abstraction that respects the synchrony hypothesis, i.e., each external event is processed by the system completely *before* the arrival of the next event. Based on the synchronous model, the Secure Operations Language (SOL) [2] is designed as a verifiable language for the integration of high assurance systems. Programs in SOL are amenable to fully automated static analysis—such as automatic theorem proving [5], or model checking [4]— to ensure compliance with application-specific requirements.

A *module* is the unit of specification in SOL. An agent is a module instance. Assumptions for correct agent operation may be specified: execution of the agent aborts when any assumption is violated. Fault-tolerant patterns, similar to the one we discuss in the sequel, may be used to deal with such exceptions in mission critical systems. Guarantees are the required safety properties of the agent. A SOL module specifies the required relation between input *monitored variables*, and output *controlled variables*. Each dependent variable (controlled as well as additional internal variables) is defined as a function of other module variables. SOL, like SCR [7], is an event-driven language, and borrows many concepts from that language as well as from LUSTRE [6].

[*]This research was funded by the Office of Naval Research.

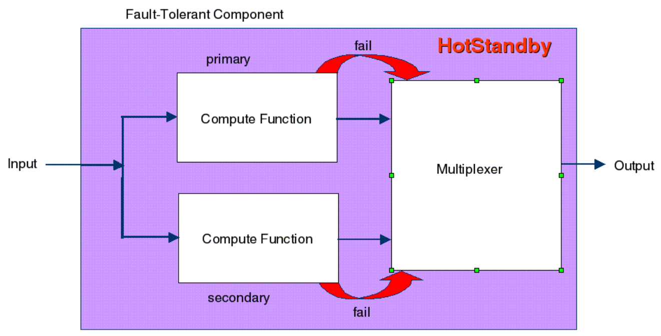

Figure 1. Hot Standby Pattern Architecture.

SOL agents run on a distributed infrastructure SINS [3]. A SINS application comprises a set of software agents that use services provided by SINS virtual machines on disparate hosts over a network. SINS provides mechanisms for the creation, deployment, and migration of agents, in addition to protocols for inter-agent communication and synchronization. SINS uses the SPREAD toolkit [1] to provide a high performance virtual synchrony messaging service.

2. Architectural Patterns

To meet fault tolerance, security, and real-time requirements of applications, we have extended SOL. Formal architectural patterns can be instantiated and then automatically compiled to a group of SINS agents running in a distributed environment. We describe the language extensions in the context of an architectural pattern denoted "Hot Standby," which provides a measure of fault-tolerant behavior via replication.

In the Hot Standby pattern a function performed by an agent at a primary site is replicated at a secondary "Standby" site. In this pattern, the failure of the primary site simply means computation results are retrieved from the secondary site rather than the failed primary site.

The Hot Standby pattern of Figure 1 is a generalization of the pattern given in [12] and is specified in an extension of SOL. In the extended language, we assume any module `A` has available attributes that can be accessed when writing architectural patterns. These include Boolean variables `X.fail` indicating failure status of the sub-modules `X` of `A`, and generic descriptions of groups of variables computed by each sub-module (such as `X.Mon` for the tuple of monitored variables comprising `X`'s input).

Additionally we employ temporal observer modules (such as NEVER(B), meaning that B has never been true) in describing the assumptions and guarantees of extended SOL modules. This notion of temporal observers was pioneered in the LUSTRE language [6].

We emphasize that sub-modules of architectural patterns need not be compilations of modules written in SOL. The sub-modules may be any COTS hardware/software components that satisfy the appropriate assumptions and provide the given guarantees (as invariants).

3. Proof of Correctness

We have used the theorem prover PVS [11] for formulating and proving the major invariant of Hot Standby:

$$(*)\quad \texttt{NEVER(primary.fail)} \text{ or } \texttt{NEVER(secondary.fail)} \Rightarrow \text{“Function computed correctly”}$$

We have shown this result for a general vector-valued sequential function over the history of the system (not simply a combinatorial function). PVS is convenient for the proofs of invariants of architectural patterns since it supports definition of uninterpreted functions in a general setting.

Invariance of q for any instance of an architectural pattern A follows from invariance of q for that instance of A in which the sub-modules are the most general ones satisfying the assumptions and guarantees imposed by A (this is a variant of the compositional proof rule given in [9]). Proof of (*) for the most general instance was done by the computational induction method of [10]: (1) the property must hold initially and (2) assuming that it holds in any state implies it must hold in the next state; previously proved or assumed invariants may be used as auxiliary lemmas.

4. Related Work

Several languages for describing architectural patterns at various levels of abstraction exist, ranging from UML to languages for describing system architectures. However none of these languages has a well-defined formal operational or denotational semantics, which is necessary for the development of architectural specifications with verifiable properties. The use of architectural patterns for dependability was introduced by Yau, et al. in [12].

5. Conclusions and Future Work

Our initial study of formal verification of architectural patterns has shown it is straightforward to prove a safety property associated with a generic module Hot Standby. We have outlined the general proof methodology, and are in the process of applying this methodology to examples that truly require this level of formal support. We envision developing and proving complex, parameterized, architectural patterns with careful attention to requisite assumptions. Our vision is that the hardware or software designer will use automatic verification tools such as model checkers to prove that the components of an architectural pattern instance satisfy the assumptions and guarantees of that pattern. In future work we plan to refine the computational induction proof methodology and to extend SOL with a polymorphic type system in support of more general architectural patterns.

The overall goal of the NRL dependable middleware project is to develop infrastructure that supports the development of secure encapsulation mechanisms for untrusted hardware and software COTS components. With such encapsulation mechanisms it should be feasible to protect mission-critical distributed applications and to design systems that provide continued service in the presence of faults or malicious attacks.

References

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

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

[3] 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.

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

[5] 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.

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

[7] 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.

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

[9] R. D. Jeffords and C. L. Heitmeyer. A strategy for efficiently verifying requirements specifications using composition and invariants. In *Proc. ESEC-9/FSE-11*, pages 28–37, Helsinki,Finland, Sept. 2003.

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

[11] 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 Int'l, Menlo Park, CA, Nov. 2001.

[12] 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.