Editor: Jeffrey Voas ■ Cigital ■ voas@cigital.com

Software Architecture Correctness

K. Suzanne Barber, *University of Texas at Austin*
Jim Holt, *Motorola*

Software quality stems from many factors, including implementation decisions, software architecture, and requirements. In this column, we focus on software architecture, which can enable or inhibit many of a software system's qualities. Because the cost of addressing quality concerns is a function of how late you address them (the later, the more costly), addressing them in the architectural or requirements phase makes sense.

Architectural quality

Researchers have begun to recognize that we can view software architecture as a formal model for requirements specification. This recognition expands the possibilities for evaluating quality attributes early during development. In this context, when evaluating architecture properties, our goals include

- providing an early opportunity to correct requirements defects and
- ensuring that the software architecture provides an accurate blueprint for system implementers.

In the first case, you could use a software architecture evaluation to check qualities such as safety, liveness, and completeness. In the second case, you might predict qualities including maintainability, performance, and reliability.

First things first

You must establish an architecture's correctness before using it as a system blueprint. An incorrect architecture will not produce quality evaluations of merit. So, the blueprint must be correct before you can build a system targeting quality goals. Correctness evaluations also mitigate the effort and cost of performing simulations or other evaluations to predict qualities such as performance and reliability.

Many techniques evaluate correctness properties of software architectures; however, no existing single technique can evaluate all correctness aspects. For example, model checking works well to evaluate safety and liveness but is not useful for evaluating completeness. Conversely, simulation helps detect completeness errors, but you can't use it to prove the absence of safety or liveness errors.

Automated translation of a software architecture specification into a form suitable for a model checker or simulation tool removes the necessity of having the software architect be an expert in model checking or simulation techniques. For the

0740-7459/01/$10.00 © 2001 IEEE

same reason, it's desirable to automatically collect and present the results of model checking or simulation. Automation is reasonably straightforward, provided the specification contains declarative elements, integration details, and behavioral information. *Declarative elements* typically take the form of sets of services collected into class interfaces, *integration details* define what data and events are exchanged between interfaces, and *behavioral information* outlines pre- and postconditions for state changes. You can use this set of information to translate the specification into a set of communicating processes defined in the model checker or simulation tool semantics.

Architectural completeness

A model checker can unambiguously detect safety and liveness errors. Completeness errors, however, are associated with unexpected system behavior, so a software architecture could exhibit an absence of safety and liveness errors but still lack a certain required functionality (a form of completeness error).

Completeness errors typically manifest in sequences of service executions (*scenarios*) that

- are missing expected service executions,
- contain unexpected service executions,
- contain unexpected paths, or
- are missing paths.

At times, completeness errors have gone undetected during requirements elicitation and have propagated into the software architecture. Such omissions or inaccuracies in the requirements are not unlikely, given the challenges of the knowledge acquisition and modeling process used to gather requirements. In fact, the knowledge contained in the requirements is a function of many variables, including the

- spectrum of expertise held by domain experts,
- time spent with each expert,
- knowledge acquisition approach,
- ability of knowledge engineers and experts to conceptualize, and
- degree to which experts can express knowledge and offer necessary detail.

So how does the software architect or domain expert use simulation results to evaluate the architecture's completeness?

Simulations can produce visualizations of the architecture's execution. These visualizations are particularly useful for identifying completeness errors by inspection—you can visually spot errors in threads of execution the simulation captures. Unfortunately, simulations have two limitations: they do not provide an exhaustive means of evaluating an architecture model, and they require a human to interpret their output. So, simulations are best for showing the presence of completeness errors, but cannot prove their absence.

A *scenario space* can help architects and domain experts visualize completeness errors. A scenario space is a directed graph that represents possible threads of execution composed of services in the software architecture. A vertex in the graph represents a service execution state, and an edge represents a path showing reachability from one service execution state to another. This visualization technique provides a high-level view of software architecture execution through partial-order reduction of simulation data to merge similar threads of execution produced over many simulation runs. Creating such a visualization can help evaluate whether executing the architecture will result in threads of execution that support the anticipated scenarios for the application domain.

Simulations can produce visualizations of the architecture's execution. These visualizations are particularly useful for identifying completeness errors by inspection.

A human approach

Regardless of method, the state of the art in evaluating correctness qualities of software architectures still requires human involvement. This invariably means human experts familiar with the application domain inspecting an artifact. Inspecting requirements specifications or static software architecture diagrams of large systems is daunting. However, cost savings provides clear motivation to perform these evaluations early. A formal software architecture representation allows for accurate, repeatable model checking and simulation for correctness evaluations, thus mitigating human effort and potential inaccuracies.

K. Suzanne Barber is an associate professor in the Electrical and Computer Engineering Department at the University of Texas at Austin. She is also the director of the Laboratory for Intelligent Processes and Systems, where ongoing research projects address distributed, autonomous agent-based systems and formal software engineering methods and tools. Contact her at barber@mail.utexas.edu.

Jim Holt is a senior verification engineer for the Motorola Semiconductor Products Sector and a PhD candidate in the Department of Electrical and Computer Engineering at the University of Texas at Austin under the supervision of K. Suzanne Barber. His work at Motorola involves creation of advanced tools for automated code generation and verification of Motorola microprocessors and systems-on-a-chip. Contact him at jim.holt@motorola.com.