Tool Support for Architecture Analysis and Design

Rick Kazman
Department of Computer Science
University of Waterloo
Waterloo, Ontario Canada N2L 3G1
+519 888-4567 x4870
rnkazman@cgl.uwaterloo.ca

ABSTRACT
The needs of software architectural design and analysis have led to a desire to create CASE tools to support the processes. Such a tool should help: to document an architecture; to reuse architectural artifacts; to aid in exploring architectural alternatives; and to support architectural metrics. This position paper first presents a set of requirements that an ideal tool for architectural design and analysis, and then presents a tool—called SAAMtool—that meets most, but not all, of these requirements. SAAMtool embodies both SAAM (Software Architecture Analysis Method) and an architectural description framework which describes architectural elements according to their static and temporal features. The tool serves several purposes. It supports and documents the results of architectural design and analysis efforts at varying degrees of resolution, it acts as a repository of both designs and design rationales in the form of scenarios, it applies metrics to architectures, and it visualizes architectures with respect to architectural metrics.

Keywords
Software Architecture Analysis; Visualization; Metrics

INTRODUCTION
Software architecture is important as a discipline because software systems are becoming too complicated to be completely design, specified, and understood by an individual. One of the aspects of software architecture which is heavily discussed as a motivation for the field, but lightly supported by tools and research in the field thus far, is the *human* aspect. Architecture, if supported properly, can have the following "people" benefits:

- focussing design inspections and design activity where they are needed most
- enhancing high-level communications within a development team
- encouraging reuse of designs
- enhancing communications between developers and managers or customers of software
- aiding high-level comparisons of competing designs

Architecture is a way of representing complex systems simply—at a high level of abstraction. As appealing as this may seem, on the surface, the messy details—implementation details—do need to be attended to at some point, otherwise the only thing that gets built is designs.

It is my contention that we need tool support for architectural design. However, tool support does not end at the design stage. A tool can also help ensure that the system that system that gets built conforms with its architecture, and can automate architectural analysis. This position paper will delineate the requirements for an architectural support tool, and will introduce such a tool, called SAAMtool., that meets some but not all of these requirements.

REQUIREMENTS FOR AN ARCHITECTURAL ANALYSIS TOOL
The process of collecting, maintaining, and validating architectural information is tedious and error-prone. Tasks that are tedious and error prone are ideal candidates for tool support. Revision control, debugging, dependency analysis, test coverage analysis are a few examples of tools that successfully automate repetitive development tasks.

An architectural analysis and design tool has to be able to meet a number of requirements for it to be useful:

1. The tool must be able to describe *any* architecture. Ideally it should support the graphical specification of architectures, as this models what people do already: they sketch architectures as an aid to design, exploration, and communication. To support this requirement best, the tool needs to use the components and connectors that people are already using, rather than a predetermined, statically defined set. Commercial design tools, such as Software through Pictures or Rational Rose, only partially meet this requirement. Commercial tools typically assume a fixed set of components and connectors, such as objects as components and method calls as connectors.
2. A tool for software architecture should be able to aggregate architectural elements recursively and be able to associate meaningful semantics with elements at any level of abstraction. This requirement is, again, only partially met by existing tools. Meeting this requirement is crucial to supporting the iterative refinement and analysis of architectures, where the analysis is meaningful at *any* level of refinement. For example, I should be able to sketch a node in the architecture and call it "database", and be able to ask meaningful design questions, or run performance analyses, without further decomposing the database. By meeting this requirement, a tool should support architectural design and analysis at any degree of design "resolution". Of course, the better the resolution, the better the answers to questions comparing design alternatives or asking about the feasibility of a design.
3. Such a tool should be able to determine conformance to interfaces, both within the architecture being designed and with any external systems. The tool should also be able to determine

conformance of the as-built architecture with the as-designed architecture [2.]. The second part of this requirement implies a substantial reverse engineering capability.

4. The tool should be able to analyze architectures with respect to metrics. Currently, few such metrics exist at the level of architecture (but some do exist, e.g. [2.], [3.]).

5. The tool should aid in design as a creative activity, and design as an analysis activity (walk-throughs for example). To do this it must support the appropriate processes for these activities. For example, Rational Rose supports both OMT and Booch design methods, and the processes that these methods imply. Related to this is the next requirement.

6. The tool should be a repository for: designs, design chunks, design rationales, and requirements/scenarios. As a repository, it should support searching an architecture, the extraction of meaningful subsets of the architecture, and of course updating the architecture. Many design tools support the first part of this requirement—they can store and retrieve designs—but few support the creation, storage, and retrieval of design rationales as first-class entities.

7. The tool should also provide for the generation of code templates, to simplify the transition from design to code, and to help ensure consistency between the two. It should also provide data and control flow modeling, including performance modeling.

SAAMTOOL

In this section I will describe *SAAMtool*, a tool that was first created to support the Software Architecture Analysis Method (SAAM) of reviewing architectural designs, as described in [4.]. Since its inception, however, SAAMtool has been extended to meet some, but not all, of the requirements listed above. In particular, SAAMtool does not currently support requirements 3 and 7.

The semantic portion of the tool—the database, graph manipulation, and architectural element definition—is written in C and C++. The user interface written in Tcl/Tk. A design analysis portion has been recently added, and it is written in Lisp.

SAAMtool has three modes in which it operates: build mode, for designing architectures and architectural patterns; scenario mode, for creating scenarios and associating them with portions of the architectures; and view mode, which allows a user to switch between the various views of an architecture, and to visualize the architecture according to metrics.

Describing an Architecture

Semantic Foundations

SAAMtool represents architectures as multi-graphs of architectural elements. Architectural elements are not simply components and connectors—such as pipe, filter, object, process, socket, etc.—but are represented more abstractly, as untyped elements which have a set of descriptive properties [5.]. These properties refer to the way the element stores and transmits data and control, the element's temporal characteristics (e.g. when can it accept and relinquish control and data), and the element's binding relationships (e.g. when does it bind; how many other elements can this element bind to; its signature, etc.). Because we do not name specific elements, but rather describe the properties of elements, our classification has no inherent scale. We can just as easily describe a file, an object, a process, or a sub-system. SAAMtool can describe single elements, clusters of elements, or entire systems in this notation. Furthermore, we can describe "near matches" among elements and patterns of elements. We can do this because we hierarchically nest elements, and recursively define the properties of a "parent" node as the composition of the properties of its children.

Hierarchical Nesting

In SAAMtool, any collection of connected nodes can be coalesced into a single node. Because of the semantic foundations described above, any coalesced node can be treated exactly as though it were an atomic element. Typically, coalesced nodes represent systems, sub-systems, or major functional components of an architecture. SAAMtool ensures that the connectivity constraints of the individual nodes are met by the coalesced "parent' node. It also ensures that the semantics of the collection of nodes are accurately represented by the parent.

Scenarios

Scenarios, as with all requirements, are difficult to elicit and validate. It is therefore useful to maintain existing scenarios in a database, as a reusable asset: as a means of determining scenario coverage (i.e. whether a given product, as designed, meets the requirements expressed in the scenarios), for scenario regression testing, or for creating scenarios for new products within the same product family. In addition, scenarios have relationships not only with the architecture, but with other scenarios. SAAMtool allows for the creation and management of arbitrary graphs of scenarios, as a hypertext database.

Supporting Multiple Views

It is clear that software is created, transformed, and managed in multiple ways, and so must be viewed in multiple ways [7.]: as code, as executing elements (also known as the "dynamic" view), through its allocation to hardware, etc. Each view in SAAMtool is maintained as a distinct multi-graph, and any node within any view can have links to nodes in one or more of the other views. Currently the tool supports a dynamic model, a code view, and a functional decomposition. Using these views one could see, for example, not just a code view of a product, but could select elements from this code view and use the set of selected elements to limit what parts of the dynamic view are viewed. This sort of flexibility is of prime importance to architectural *mining* and understanding: being able to flexibly constrain the view of the architecture based upon user-specified criteria.

Scenarios are maintained as a set of arbitrary graphs. That is, any scenario can be related to any number of other scenarios. Scenarios are also related to portions of an architectural view. Currently we only support the relationship between scenarios and the dynamic view of an architecture.

Because scenarios are maintained as a distinct data structure, they can be used to specify (primarily, to limit) the scope of one of the other views of the architecture. This is of use when trying to understand an architecture. We often want to constrain our current view of the software through some other view. For example, we might want to know what source modules would be affected if we moved an executing element from one hardware or software platform to another. Or, we might want to know what executing elements would be affected if we change some source module. We have extended this notion to include scenarios. We can see the effects of a scenario mapped onto one of the other views. This manifests itself in two ways. For direct scenarios (those that describe some execution of the system) we can use the scenario to walk through a trace of the execution. For indirect scenarios (those that describe some desired or anticipated change to the system), we can use the scenario to describe and visualize the set of architectural elements that will be affected.

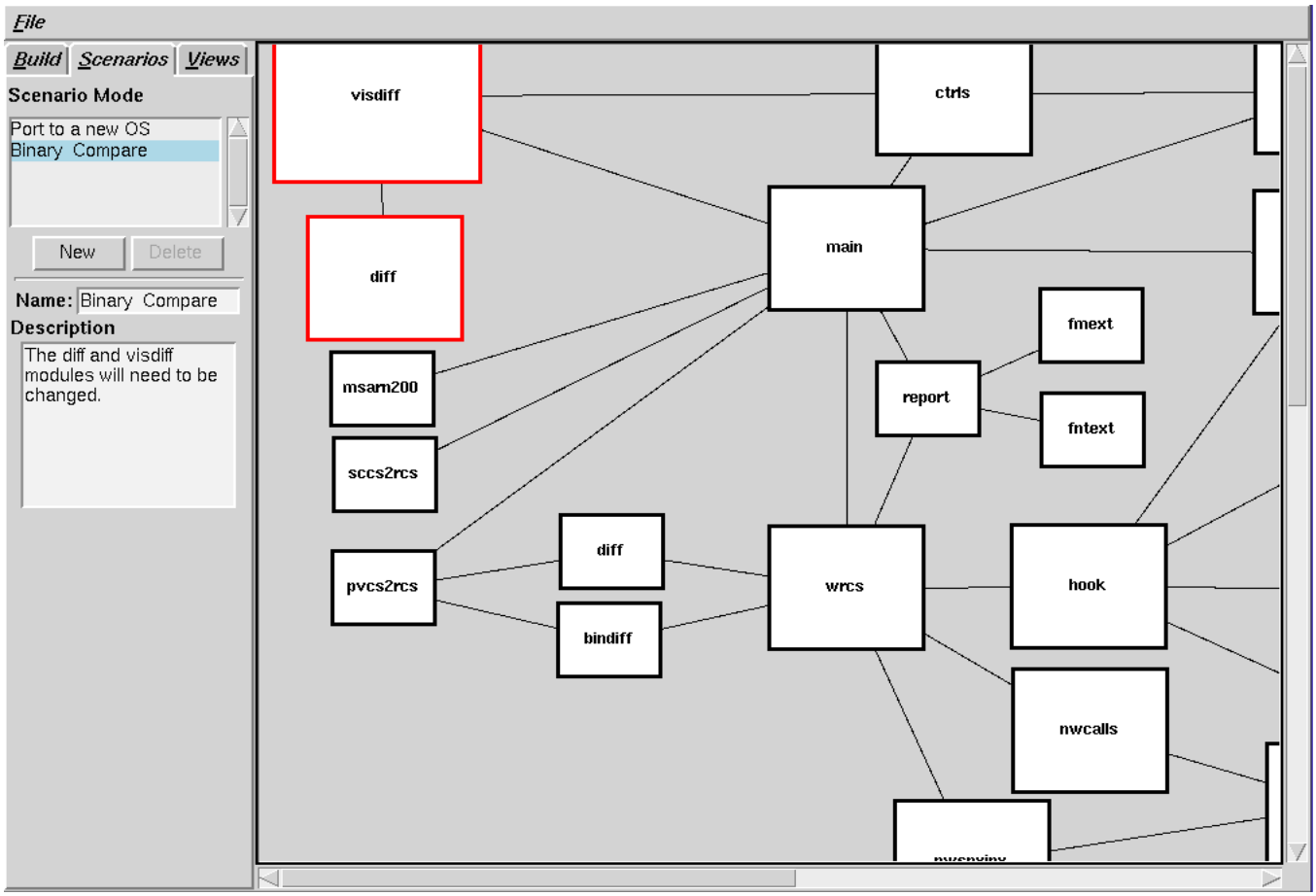

Figure 1: A Fish-eye View of an Architecture in SAAMtool

Visualization of Architectures

Related to the notion of supporting multiple views of an architecture, we would like to be able to visualize information about a software architecture in the same tool that we create and maintain the architecture. Architectures for complex systems need to be visualized for the same reasons that any complex information set needs to be visualized: because humans are better pattern recognizers than symbolic information processors. Visualization is a powerful tool for guiding the designer to potential trouble spots in the architecture. For example, Figure 1 shows the architecture for a Windows-based revision control system, visualized with respect to change analysis (larger rectangles indicate elements which are more heavily affected by the set of proposed changes). This visualization vividly draws a designer's attention to the *visdiff* module, for example, as being a potential "hot spot" in the architecture.

Visualization is also useful in describing the complexity of an architecture, as measured by the uniformity—or what Brooks calls "conceptual integrity" [1.] of the architecture. We measure this as the number of different primitive patterns that the architecture employs [6.]. This is another way of gauging the system's adherence to a coherent architectural style. We visualize the results of this metric by mapping patterns onto an architecture in unique colors. The number of colors required to "cover" an architecture, and the percentage of the architecture colored vividly demonstrates the conceptual integrity, or complexity, of the architecture.

An example of this kind of visualization is given in Figure 2 where a single pattern is mapped onto an architecture of about 100 elements. A single pattern covers 13% of this architecture. Covered elements are shown with a red border. The current pattern being viewed is highlighted.

Analyzing Architectures

SAAMtool supports the process of architectural analyses and walk-throughs as described in [4.]. It does this by recording and classifying scenarios, associating scenarios with architectural elements, and allowing the entire architecture, or subsets of it to be visualized with respect to architectural metrics, such as structural complexity, change analysis, coupling, and architectural pattern matching.

It aids the analyst by keeping track of the relationships between views, and in particular, the relationships between direct and indirect scenarios and the various views of the system's software.

NEXT STEPS

It has been my experience, in performing architectural analyses with large software development organizations such as Nortel, MKS, and Tektronix, that the overhead of collecting, managing, and presenting the information relevant to architectural design and analysis is a substantial impediment to organizations wanting to adopt a more mature attitude to their software architectural practice. Providing tool support for this practice is a first step in aiding its widespread adoption in industry.

Future work with SAAMtool lies in two area. I would like to develop and integrate other architectural metrics, particularly those

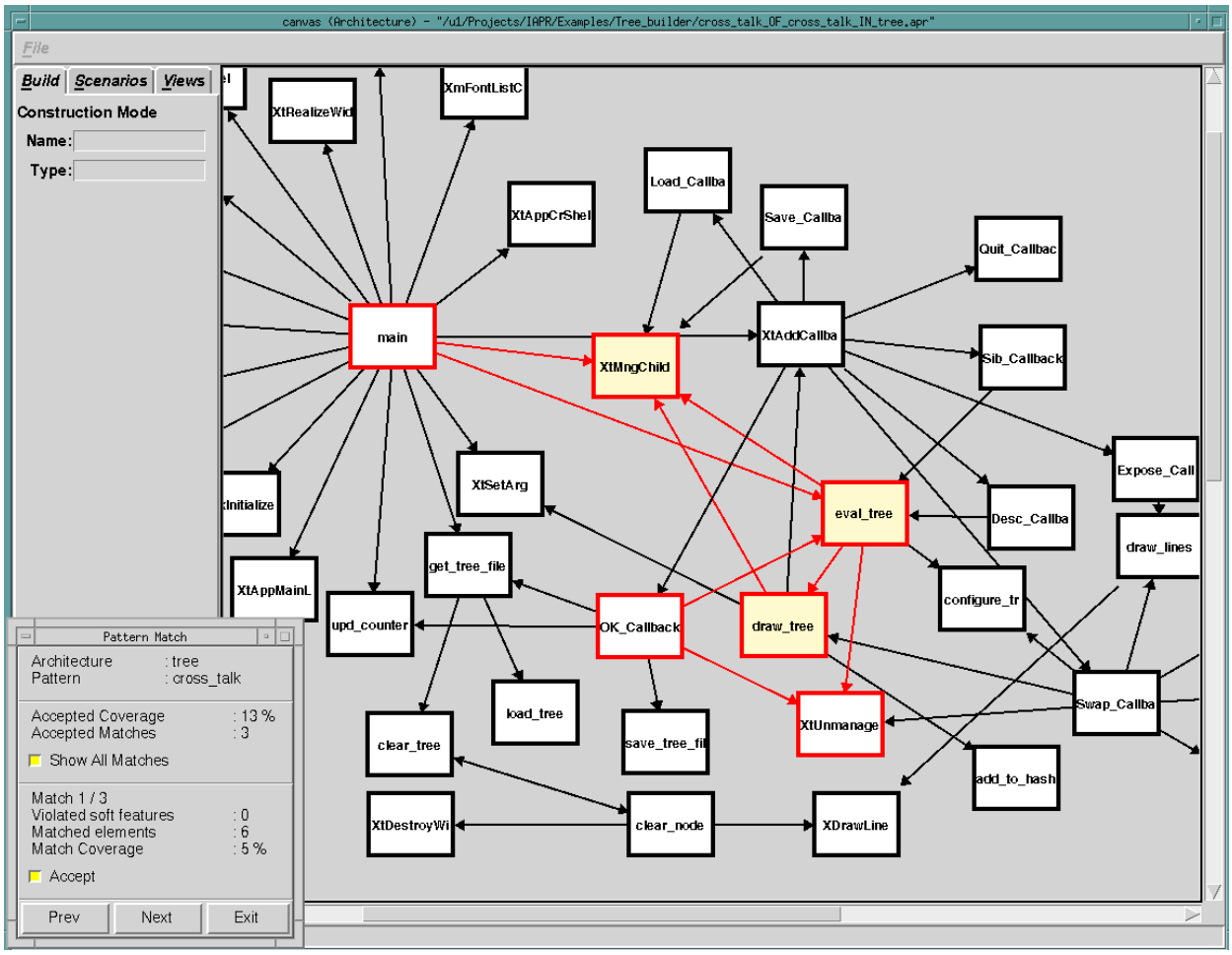

Figure 2: An Example of Architectural Pattern Matching in SAAMtool

that help a human comprehend the complexity of a design. Just as important, however, is the continued exploration of new visualization techniques, since these help to make analytical information interpretable by the designer. Finally, it is crucial to integration such a tool with performance analysis tools, and with reverse engineering tools. These last two areas remain outstanding challenges.

ACKNOWLEDGMENTS

I would like to thank Gavin Peters and Marcus Burth for all of their dedicated work on SAAMtool.

REFERENCES

1. Brooks, F. Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1982.
2. Henry, S., Kafura, D. “Software Structure Metrics Based on Information Flow”, *IEEE Transactions on Software Engineering*, SE-7(5), Sept. 1981.
3. Heyliger, G., “Coupling”, *Encyclopedia of Software Engineering*, J. Marciniak (ed.), 220-228.
4. Kazman, R., Abowd, G., Bass, L., Clements, P., “Scenario-Based Analysis of Software Architecture”, *IEEE Software*, November 1996.
5. Kazman, R., Clements, P., Abowd, G., Bass, L., “Classifying Architectural Elements as a Foundation for Mechanism Matching”, http://www.cgl.uwaterloo.ca/~rnkazman/Foundat.ps, 1996.
6. Kazman, R., Burth, M., “Assessing Architectural Complexity”, http://www.cgl.uwaterloo.ca/~rnkazman/assessing.ps, 1996.
7. Kruchten, P. “The 4+1 View Model of Architecture.” *IEEE Software 12*, 6, Nov. 1995, 42-50.
8. Murphy, G., Notkin, D., Sullivan, K., “Software Reflexion Models: Bridging the Gap Between Source and High-Level Models”, *Proceedings of the FSE '95*, Oct. 1995.