Toward a Discipline of Scenario-based Architectural Engineering[1]

Rick Kazman, S. Jeromy Carrière, Steven G. Woods
Software Engineering Institute, Carnegie Mellon University
Pittsburgh, PA 15213

Abstract: Software architecture analysis is a cost-effective means of controlling risk and maintaining system quality throughout the processes of software design, development and maintenance. This paper presents a sequence of steps that maps architectural quality goals into scenarios that measure the goals, mechanisms that realize the scenarios and analytic models that measure the results. This mapping ensures that design decisions and their rationale are documented in such a fashion that they can be systematically explored, varied, and potentially traded off against each other. As systems evolve, the analytic models can be used to assess the impact of architectural changes, relative to the system's changing quality goals. Although scenarios have been extensively used in software design to understand the ways in which a system meets its operational requirements, there has been little systematic use of scenarios to support analysis, particularly analysis of a software architecture's quality attributes: modifiability, portability, extensibility, security, availability, and so forth. In this paper we present a unified approach to using scenarios to support both the design, analysis and maintenance of software architectures, and examples from large-scale software development projects where we have applied the approach. We also present a tool, called *Brie*, that aids in: scenario capture, mapping scenarios to software architectures, and the association of analytic models with particular portions of architectures. The approach that we have devised, and that Brie supports, is a foundation for a discipline of *architectural engineering*. Architectural engineering is an iterative method of design, analysis and maintenance where design decisions are motivated by scenarios, and are supported by documented analyses.

1 Introduction

Scenario-based analysis of software architecture is increasingly seen as an elegant, cost-effective means of controlling the risks inherent in architectural design [1]. For the past five years, software engineers at the Software Engineering Institute (SEI) have been using scenarios to guide in the design and analysis of software architectures, to help in the acquisition of such systems, and to help assess the impacts of architectural evolution. This work began initially with the development of the Software Architecture Analysis Method (SAAM) ([15], [16]), and later continued as an integral part of the Architecture Tradeoff Analysis Method (ATAM) [17]. We use architectural analysis to validate designs of complex hardware and software systems that are under consideration (either when a system is being first designed, or later in its life cycle), or to help in the acquisition of such systems. We say "design and analysis" of architectures, because we believe that the two are inseparable. Designs must be validated as early as possible in the development life cycle, and analysis techniques at the level of a software or system architecture can provide the necessary insight. Our view is that the use of scenarios to guide design and its associated analysis is a foundation for a discipline of *architectural engineering*.

1. This work is sponsored by the US Department of Defense. The Sotware Engineering Institute is a federally-funded research and development center sponsored by the U.S. Department of Defense.

The architectures of long-lived systems are not static: they grow and change over their lifetimes in response to changing requirements and technology. So, too, must the architectural analyses grow and change, so that the modifications that an architecture undergoes are advisable ones; that is, ones that ensure that the system will meet its quality goals [22]. Analysis can provide the rationale for determining which design options to pursue and which to discard.

However, an architectural analysis is only as good as the set of requirements for which it is validated. When we first began doing analyses of software architectures we used scenarios as a means of eliciting these requirements. A scenario, in our use of the term, is a brief description of a single interaction of a stakeholder with a system. This concept is similar to the notion of use cases prevalent in the object-oriented community. However, use cases focus on run-time behavior where the stakeholder is the user. Scenarios encompass other interactions with the system as well, such as a maintainer carrying out a modification. Now, what is the relationship between scenarios and requirements? Given that scenarios represent a single, end-to-end interaction of a user with the system, they may encompass *many* requirements. We typically use scenarios to operationalize requirements, so that we can understand their mapping onto the architecture, their impacts, and their interactions. Scenarios are also used for brainstorming uses of the system, which frequently results in the creation of new requirements.

As it turns out, scenarios have many other uses beyond requirements elicitation and validation, such as aiding in stakeholder understanding and buy-in, and in eliciting a variety of architectural views. When evaluating an architecture we gather as many representatives of different stakeholder groups as we can into a room for a one or two day session of scenario elicitation. Elicitation of major uses of the system is often needed for documenting an appropriate representation of the architecture.

Additional scenarios capture changes to the system and other uses of the system. Once we have elicited an adequate set of scenarios, as determined by consensus among the system stakeholders, we map them onto the documented views of the architecture. The views that we use depend on the qualities that we are analyzing; for example: a run-time view for evaluating performance and availability, a source view for evaluating development qualities such as modifiability and portability, and a “uses” view [3] for evaluating the ability to create system subsets or supersets.

Once the views and scenarios have been elicited and we’ve mapped the scenarios onto the views, we build analytic models. We build these models to more precisely understand and document the structural decisions we’ve made in the software and system architectures. This paper describes our methods for eliciting scenarios, mapping them onto architectures, and building analytic models of their impacts. One of the by-products of doing architectural analyses on a regular basis is that we have built a tool, called *Brie*, to support this process. We needed to create a tool because the software intensive systems that we evaluate are complex, they have many stakeholders and many scenarios, and they require many analytic models of their qualities. Another source of complexity in this process comes from the fact that these scenarios and their models interact: optimizing one quality comes at the expense of another. So, for example, increasing performance typically comes at the expense of modifiability, increasing reliability frequently comes at the expense of performance, and increasing security typically reduces a system’s interoperability.

These are not hard and fast rules, of course. As a consequence we use analytic models of an architecture to understand it and to document our understanding. Using these models we are able to run many different scenarios (“what if” scenarios) over the architecture to compare their effects: to see where changing one aspect of an architecture—say, the insertion of a resource manager through which all file accesses must flow, or the increase in the number of servers in a client-server architecture—affects the qualities of the architecture along many dimensions. Using scenarios, we build analytic models of a use of, or change to, the architecture to understand the performance impacts, security impacts, modifiability impacts, and so forth. But managing so much architectural information, so many scenarios, and so many interrelated models is an

enormous bookkeeping problem. For this reason, we require a tool to help us manage the mountains of information and the constraints among the pieces.

2 Using Scenarios in Architecture Analysis

The scenarios that we use in architectural analyses are of two types, which we label *direct* and *indirect*. Direct scenarios represent uses of a system built from a software architecture, where the system is viewed from an external perspective; that is a direct scenario is initiated by the arrival of some stimulus at the system and continues until the system has processed this stimulus to completion. Examples include the processing and transmission of an operator command from a satellite ground station, arrival and processing of a transaction in a transaction processing system, a user dialing a telephone and making a connection, a process control system reading a set of state variables and adjusting settings on the system that it is controlling, and so forth. Indirect scenarios represent changes to the existing architecture, and hence to the system or systems built from it. These include porting to a new hardware or software platform, adding a new feature, integrating with a new piece of software, replacing some existing functionality with a new version (changing from a home-grown distributed object communication infrastructure to CORBA, for example), scaling the system to handle greater volume, and other similar kinds of changes to what the system does.

We use direct and indirect scenarios in different ways in our architectural analyses, and at different phases. Direct scenarios are used early in an analysis. For example, we can use them to *create* an appropriate architectural representation, because we commonly find that organizations either do not know what the architecture of their system is, or they know but haven't documented the architecture, or they have documented it but not in a way that supports analysis. As one example, we frequently find that organizations will have a set of object diagrams, but little or no notion of how these objects interact or how the system infrastructure supports this interaction. Mapping a set of direct scenarios onto a set of objects elicits precisely this kind of information. In doing the mapping, we cause the architects to reveal how data and control flow through the system: we in fact cause the architecture underlying the objects to be revealed and documented. From this information, we can determine the prototypical flows—which define the major functions of the system—and distinguish these from the idiosyncratic ones.

An important point to note here is that this activity allows us to raise to the architectural level the set of infrastructure services. These services determine issues such as: how system activities are scheduled and coordinated, how data moves through the system, how faults are propagated and handled, how security is achieved, how performance goals are measured and met, how new components are integrated, and how the system scales. It is this set of infrastructure services, far more than the functional goals of the system, that determine the form of the architecture. And yet this aspect of the architecture is frequently not documented, or passed over as "just implementation details". So, the use of direct scenarios to elicit this information has become an essential part of architectural analysis.

Once we have an appropriately represented software architecture and have mapped a set of direct scenarios onto it, we can begin to consider the indirect scenarios: representations of proposed and anticipated changes to the system. We use the analysis of these scenarios as a form of risk mitigation. The indirect scenarios are collected from as broad a group of system stakeholders (e.g. system owner, program manager, development contractor, potential reuser, component engineer, maintainer, software librarian, end user, system administrator) as possible. The scenarios are mapped onto the architecture, one at a time. In this mapping, a determination must be made by the architects as to what changes are necessary to satisfy the scenario. This determination takes the form of a set of changes to interfaces, component internals, data connections, and control connections. In addition, and more importantly, we look at the *interaction* of these scenarios: areas of the software architecture that are host to scenarios of different types.

Examining scenario interaction identifies important areas of the architecture: areas that have not been properly modularized with respect to the chosen set of scenarios, or which are architectural tradeoff points. Scenario interaction is a focused coupling and cohesion measure [5]. But unlike traditional coupling and cohesion measures, scenario interaction determines the coupling and cohesion of components with respect to a set of anticipated changes. Portions of the architecture that are host to large numbers of distinct scenarios represent areas of anticipated high future (or even high current) complexity. These then become areas for discussion of refactorization of functionality.[2]

Therefore, indirect scenarios are a guide to risk management, where the risks being managed are those related to controlling the complexity of the software architecture, and hence the robustness and maintainability of products derived from the architecture. For example, understanding the interaction of scenarios can cause today's enhancements to be met in such a way that they support, rather than hinder, the achievement of tomorrow's. Mapping the scenarios onto an architecture is simply a technique for addressing "what if" questions. The choice of which scenarios to consider, and in what order, is a matter of how the stakeholders prioritize their various indirect scenarios.

2.1 The Benefits of Using Scenarios in Analysis of Architectures

We have found a number of other benefits that accrue to the collection and analysis of scenarios. Initially, in doing architectural analyses, we gathered scenarios in an ad hoc fashion. However, as time went on, one of the significant ways in which our expertise in architecture analysis grew was in our sophistication with eliciting, recording, and grouping scenarios, and associating these with architectural artifacts. We have come to realize that scenarios are significant for several reasons:

Better understanding of requirements: the mapping of direct and indirect scenarios onto an architecture frequently reveals implied requirements and conflicting requirements and sheds light on how requirements interact in the implemented system

Stakeholder buy-in and shared understanding: allowing stakeholders to see how their scenarios are fulfilled by the architecture, or could be accommodated by the architecture, increase a stakeholder's confidence that the architecture would support them. This then increases their understanding of the architecture. The architecture is a common reference point from which many different needs are understood and many different perspectives (e.g. user, administrator, maintainer) are integrated.

Better documentation: scenarios can help in eliciting the architecture of an existing system, by having the system's designer "walk through" a number of common direct scenarios, explaining the mapping of these scenarios to system components. This can also be used to elicit the architecture of a system that is being built, but which has not had any architectural documentation. Furthermore, since part of an architecture's power is that it is abstract, it is not always obvious what the right level of abstraction is to adequately represent the architecture—too much detail is burdensome, while not enough is inadequate. The mapping of scenarios helps to determine the appropriate level of architectural representation, which is the level at which the impact of these scenarios can be adequately assessed [16].

Requirements traceability at the architectural level: the satisfaction of requirements is frequently not attributable to a single software component, or even a small set of components. Requirements are best understood when mapped onto the architecture as a whole. To do this we realize requirements as scenarios and map these scenarios onto the architectural representations. The satisfaction of some requirements, particularly those related to system quality attributes—security, portability, maintainability, and so forth ([2],

2. See [16] and [3] for a more extensive discussion of this point.

[22])—crucially depends upon architectural decisions. As a result these requirements must be traced at the architectural level, requiring an understanding of both application-specific and infrastructure software [21].

Analysis of quality attributes: stakeholders are frequently not the initial creators of a software product, but rather purchasers, integrators, testers, and maintenance programmers: the people who are going to have to adapt an architecture, maintain it, or integrate with it. The concerns of these stakeholders are with performance, availability, integrability and so forth as much as they are with a system's function. For these stakeholders, seeing how scenarios (that are representative of their modification, integration and maintenance activities) are fulfilled by the architecture has many benefits. These benefits accrue because mapping the scenarios onto an architecture aids them in seeing how easy or difficult the scenarios will be to achieve. Thus, it helps the stakeholders manage risk, plan resource allocation, determine an ordering of activities based upon their achievability, and reason about the tradeoffs that inevitably occur when design decisions are being made. The most important impact of analyzing quality attributes is on cost; scenario-based architectural analysis has been proven to uncover architectural faults early in the software development lifecycle [1]. Architectural analysis also provides insight into the relative cost of achieving various system quality goals by manifesting the mechanisms by which those goals are realized. Frequently these mechanisms are not represented or analyzed because no requirement is directly traceable to them; they are considered "just infrastructure".

While some of these benefits accrue to other analysis techniques such as requirements engineering [8] and domain analysis [25], scenario-based analyses are highly cost-effective and focus the stakeholders on areas where scenarios interact. These areas of interaction are where engineering tradeoffs must be made in the architecture.

We will examine these benefits of scenario-based analysis in more detail this paper. In doing these analyses, we have come to realize that scenarios were themselves important artifacts in the development of complex software systems, and these artifacts needed to be managed. For this reason, we have developed the Brie architectural representation and manipulation environment, which represents scenarios as first-class entities, and not just as annotations to the architecture. Scenarios can be represented alongside architectural artifacts and, crucially, can be associated with these artifacts.

2.2 Abstract Scenarios for a Reusable Software Architecture

Scenarios can also play an important role in capturing requirements for a family of systems, often called a "product line" [27], an application framework, or a reusable toolkit. In product lines, the architecture is the central reusable asset shared among the systems. These requirements are different from those traditionally collected for single-use systems. Consider the following example scenario:

Database updates are propagated to clients within a bounded amount of time

This direct scenario—propagating database updates to clients—embodies a requirement that can only be specified in an oblique fashion—a bounded amount of time. This is because the scenario applies to a *family of systems*, rather than a single system, and so few assumptions can be made about the hardware platform, the environment in which the system is running, other tasks competing for resources, and so forth. Given the variability inherent in a family of systems, the stakeholders must resort to a *generic* requirement, represented by an *abstract scenario*. Traditional requirements of the form "the system shall do such and such" are frequently inappropriate for families of systems. They are replaced with abstract scenarios that represent entire classes of system-specific scenarios, appropriately parameterized. This has several important implications:

- an architectural analysis for a product line must identify infrastructure mechanisms for ensuring that this scenario can be met *in general*. To meet this scenario in general, an appropriate infrastructure for meeting performance deadlines must be put into the architecture. This would include mechanisms such as performance monitoring, task scheduling and potentially the support of multiple scheduling policies, as well as the ability to prioritize and preempt tasks. These mechanisms are necessary in a product line architecture, because other forms of meeting performance deadlines that are appropriate in a single system (such as manual performance optimization) do not scale up to a family of systems.
- tracing abstract scenarios from their original statement, through a specific version for a specific realization of the architecture to a single realized system is much more complex than the equivalent tracing activity for a traditional single-use architecture. But we need to maintain the scenario at this level of abstraction because it is only here that it becomes a constraint on the architecture *as a reusable asset* rather than a system-specific goal that might be met, in a single-use architecture, through more traditional means such as code optimization or manual load balancing.
- because tracing an abstract scenario is more complex than tracing a scenario mapped onto a single-use architecture, it suggests a need for tool support. Such support for tracing abstract scenarios and for managing the constraints that they impose upon a product line architecture (and all its instances) will be discussed in Section 5.

3 A Scenario Elicitation Matrix

Architectural analysis is motivated by various quality attributes [22] and needs input from many stakeholders. Why is this? No single stakeholder represents all the ways in which a system will be used. No single stakeholder will understand the future pressures that a system will have to withstand. Each of these concerns must be reflected by the scenarios that we collect.

We now have a complex problem however. We have multiple stakeholders, each of whom might have multiple scenarios of concern to them. They would rightly like to be reassured that the architecture satisfies all of these scenarios in an acceptable fashion. And many of these scenarios will have implications for multiple system qualities, such as maintainability, performance, security, modifiability, and availability.

We need to reflect these scenarios in the architectural views that we document and the architectures that we build. We need to be able to understand the impacts of the scenarios on the software architecture. We further need to trace the connections from a scenario to other scenarios, to the analytic models of the architecture that we construct, and to the architecture itself (this will be discussed in detail in Section 4). As a consequence, understanding the architecture's satisfaction of the scenario depends on having a framework that helps us to ask the right questions of the architecture.

To use scenarios appropriately, we typically work from a 3 dimensional matrix, as shown in Figure 1:

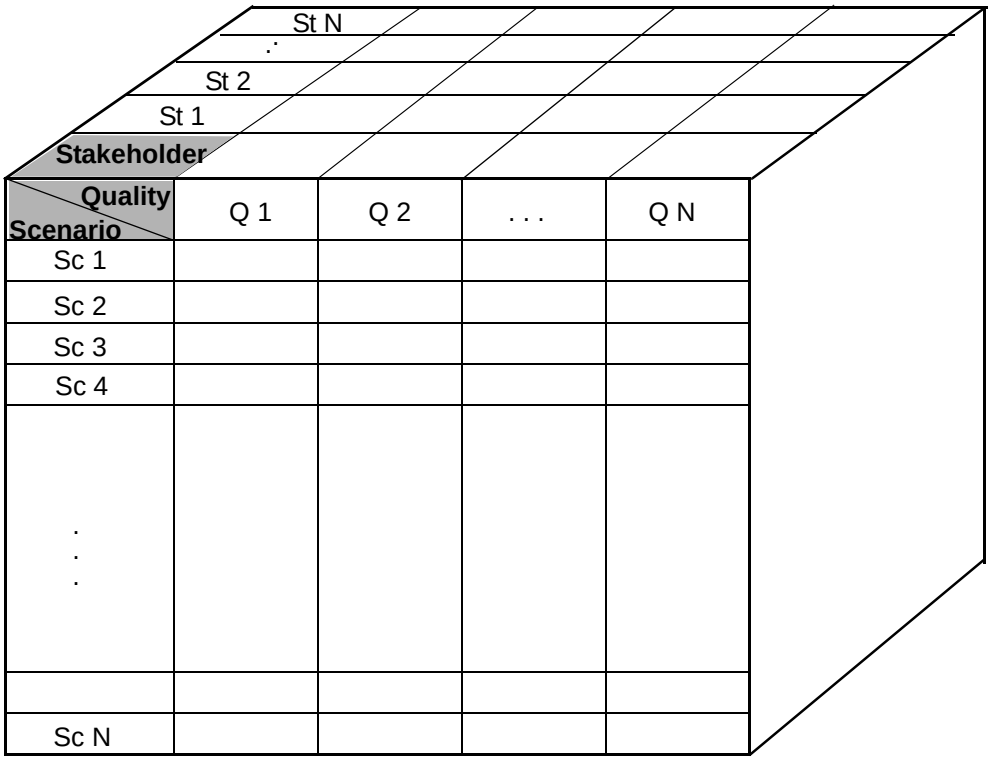

Figure 1: A Generic Scenario Elicitation Matrix

Having an explicit matrix helps us to *elicit* and *document* scenarios that we might otherwise overlook, and causes us to consider their implications from the perspectives of multiple stakeholders and multiple qualities. The matrix is, in practice, typically sparse. However, the fact that we create the matrix and reason about its population ensures that the identified stakeholders are included in the scenario elicitation process on an equal basis. The existence of the matrix *makes* us consider each scenario from each stakeholder's perspective, and from the perspective of each quality of interest.

The qualities, however, are included only as placeholders. You cannot directly evaluate an architecture for the generic quality of flexibility, modifiability, or performance, despite the fact that these sorts of qualities are regularly stipulated in requirements documents (e.g. "The architecture shall be flexible and robust"). These qualities, as stated, are too vague to be directly evaluable: an architecture cannot be flexible with respect to all possible uses and modifications, for example. Every architecture embodies some decisions that restrict flexibility. Thus, we cannot say that an architecture is flexible, but only that it is appropriately flexible with respect to an anticipated set of uses which we characterize using scenarios.

Thus, we use scenarios to illustrate and realize specific aspects of these qualities—the particular instantiations of the quality that are important to the stakeholders. So why do we include these categories in the matrix? We do it for the purposes of stimulating discussion among the stakeholders, and for seeding the set of concepts under consideration. Furthermore, a single scenario does not necessarily have a single related quality attribute. A portability scenario may have implications for security; an integrability scenario may have implications for performance. But simply having the categories explicitly placed in the matrix causes the stakeholders to consider the potential implications of each quality for each scenario.

Consider the following example of a product that is to be ported to several different hardware/software platforms. In doing an architectural analysis of the product, we would consider the ramifications of this scenario from multiple perspectives, as shown in Figure 2:

End User

Quality / Scenario	Modifiability	Performance	Maintainability	Security
Software is available on Windows, Solaris, Macintosh		**X**	**X**	

Administrator

Quality / Scenario	Modifiability	Performance	Maintainability	Security
Software is available on Windows, Solaris, Macintosh			**X**	

Programmer

Quality / Scenario	Modifiability	Performance	Maintainability	Security
Software is available on Windows, Solaris, Macintosh	**X**	**X**		

Figure 2: A Single Scenario from Three Stakeholder Perspectives

The issues that arise from the elicitation process have different ramifications for different stakeholders. Consider the example above. The performance and maintainability issues for the end user are demonstrable by acquiring and running the system. They are direct scenarios for the end user and are satisfied if the system is maintainable (for the end user this simply means that new versions are “installable”) and runs “fast enough”.

For the administrator, the effects of the scenario are less obvious. The administrator is responsible for installing the latest version of the software on the three platforms, and so is, in this sense, maintaining the software. For example, the software might be highly configurable and this configuration is entirely contained within and managed by an installation program, in which case the scenario is direct. Or the installation might be manual, requiring extensive tailoring of resource files and possibly programming to achieve, in which case the scenario is indirect.

Finally, this scenario is clearly indirect for the programmer. But the means of making the software available on all three platforms are not specified in the scenario. This is as it should be: the scenario is a statement of an expectation and should not constrain how that expectation is to be met. So, the programmer has a tradeoff to make between modifiability and performance. He might choose to develop and maintain three distinct but functionally identical products, one at a time. In this case each product can be tuned for optimal performance on each specific platform. Or he might only develop a single software base that includes a portability service (typically implemented as a layer). Once this service is constructed, all that remains is to compile and test the resulting system on each platform. In this latter case, however, there is a performance issue that the programmer must face as the use of a portability service has a run-time cost, and typically precludes specialized platform-specific performance tuning. This issue might then catalyze a discussion between the stakeholders that have a stake in the performance of the system: the end user and the programmer.

In our experience, the existence of multiple quality attribute categories in the matrix prompts the system’s stakeholders to consider just such issues that cross categories. Once we have elicited scenarios from the stakeholders, using the elicitation matrix to stimulate questions and discussion, our next step in doing architectural analyses is to map these scenarios onto an architectural representation.

4 Mapping Scenarios

Mapping scenarios onto an architectural representation sounds, at first blush, like a simple task. And in some cases it is: where the architecture already exists in an adequately documented form and we want to understand, for example, the effects of a set of indirect scenarios. In such a case we simply associate the scenario with the portion of the architecture that it affects, and use this to guide an evaluation, as described in [2] and [16]. In such a case we present the architecture, along with a mapping of a set of scenarios onto structural elements (components, connections, and interfaces) that they affect, as exemplified in Figure 3. This mapping then catalyzes a set of discussions with the stakeholders regarding the impact of the proposed set of changes. The impacts can be analyzed individually, and their interactions can be assessed, as described in Section 2.

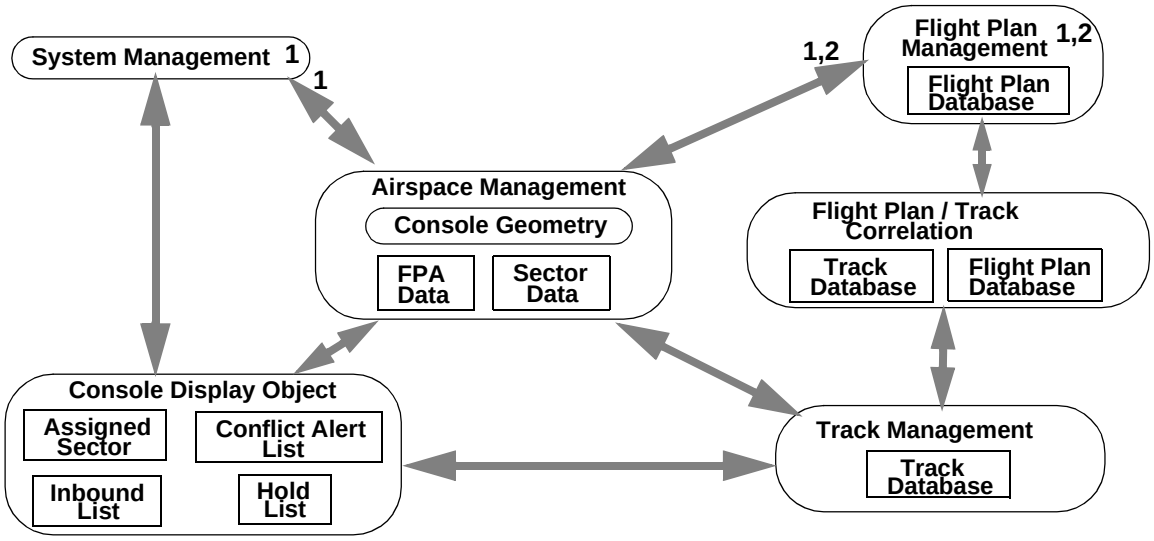

Figure 3: An Air Traffic Control Design with 2 Mapped Change Scenarios (from [24])

However, the mapping of scenarios in many architectural analyses is more complex than this. For example, a scenario may be generic, representing a class of constraints on a system or family of systems, as explained in Section 2.2. In such a case the abstract scenario must be “realized” as one or more derived scenarios, which are then mapped onto the architecture or architectures.

In a maintenance context, the architect must perturb an existing architecture to meet new functional and quality requirements, while not negatively impacting requirements that have not changed.

Sometimes the architecture does not yet exist, or it only partially exists. In such cases we use the techniques presented here as part of an architectural engineering process during the initial design phase, or during a major design overhaul. In this process, we use scenarios to suggest and constrain various design alterna-

tives. The scenarios, and the mechanisms used to satisfy them, define the form of an analytic model. This process is described in Figure 4.

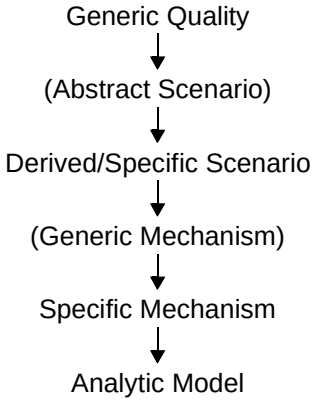

Figure 4: Artifacts and their Translations for Mapping Scenarios

The steps shown in this figure are a means of translating from the generic quality goals to specific architectural mechanisms and their associated analyses. Some of these steps are optional, indicated by the parentheses, as will be explained below.

4.1 Generic Qualities

Recall that the elicitation matrix that we use to generate our initial set of scenarios only categorizes these scenarios with respect to *generic* quality goals: performance, availability, modifiability, and so forth. While these goals are useful for elicitation, they are too general to be useful in any practical architectural design and analysis.

The first step in translating from these generic statements of desired qualities to a more usable form is to map them to scenarios; either abstract scenarios such as those described in Section 2.2, or directly to specific scenarios.

4.2 Abstract and Derived/Specific Scenarios

If we translate generic qualities to abstract scenarios—scenarios representing a class of requirements, often for a class of related systems that will be derived from the architecture—we must then instantiate these abstract scenarios. We do this for the same reason that we are unable to use generic qualities: because abstract scenarios cannot be directly analyzed; they are too abstract. So, we instantiate them as specific *derived* scenarios. This mapping process entails making some assumptions about the environment and constraints under which the scenario is valid, and these must be documented.

If we are only worried about a single system derived from the software architecture, then we can translate a generic quality directly into a specific scenario, and this scenario will then directly contain all assumptions about its applicability, such as the preconditions under which it is applicable. This is the way that “use cases” are currently applied in some forms of object oriented design, for example [14].

4.3 Generic Mechanism

The next step in the architectural engineering process is to propose a mechanism for responding to this scenario or class of scenarios. This mechanism can, once again, be generic. There are, for example, sets of engineering design principles for dealing with performance constraints, or modifiability, or fault tolerance.

For example: if one needs to meet end-to-end deadlines across distributed resources, one approach to achieve this is through priority-based preemptive scheduling. If one wants to build a system such that it can use multiple user interface toolkits, an approach is to have the application directly reference a virtual toolkit, which then mediates the interaction with any number of specific toolkits. If one wants to achieve high reliability, an approach is to use redundant components with voting; and so forth.

However, each of these is a generic mechanism; a response to a generic statement of need. To be truly useful in a design, it must be tailored to fit into the proposed design and to meet the specific set of competing quality goals and constraints in this design. This is the next step in an engineering process.

Of course, if an architecture is not being built with reusability or product lines in mind, the generic mechanism step is typically skipped (as indicated in Figure 4), with designers moving to directly build specific architectural mechanisms.

4.4 Specific mechanism

Once the mechanism (generic or not) has been decided upon, it must be instantiated within the constraints of the architecture being built. These constraints include those of cost, interoperation with other systems, mandated use of COTS (Commercial Off-The-Shelf) or standard interfaces, integration with legacy software and the environment in which the system is to operate (e.g. rate and distribution of service request arrivals, frequency of power outages, and so forth).

Within these bounds, a mechanism is chosen and tailored. For example, an architecture for a system that needs to decouple producers and consumers of information might choose as a generic mechanism a publish-subscribe scheme. As a specific mechanism this architect might choose to make use of CORBA's event notification service [26], because COTS products supporting this service can be readily purchased. Or, perhaps the system has stringent performance requirements that an off-the-shelf package is unlikely to meet; in this case the architect could design a custom event notification package. Such a package might provide for event priorities so that performance tuning is more easily achieved, or could implement a send-only-once protocol, which would have good performance, but with no provision for re-sending events (for fault tolerance). Only the specific environmental constraints will determine which specific mechanism to buy or to build.

4.5 Analytic model

Once a number of scenarios, and a mechanism for satisfying them has been chosen, an associated analytic model must be built. This is a key tenet of architectural engineering. The process of architectural engineering is so complex and involves to many competing factors that analytic models for all attributes of interest in all risk areas must be created. While this sounds like an enormous amount of additional work (that is, this model building is in addition to the job of building the actual system) it is both *necessary* and *tractable*.

It is *necessary* because there is simply no other way to mitigate risk in complex systems, where the structural decisions cause tradeoffs among various attributes: modifiability for performance, fault tolerance for time to market, security for usability, and so forth. To maintain intellectual control over the myriad possibilities, models must be built. But these models must also be interconnected, to represent the tradeoffs that live within the structure of the software (and system) architecture.

Analytic model building is *tractable* because one doesn't build a model for every part of a system. This is an iterative, risk driven process. Models are only built for areas of the system that, at a coarse level of analysis, appear to be potential risks.

The analytic models that we use at this point in the process are not novel. They have existed for years within specialized software engineering subcommunities. So, for example, we use RMA for performance analysis [20] and Markov modeling for analysis of availability. Our analysis for modifiability is a predictive coupling measure [5]. And so forth. The point is that this process of architectural engineering does not require radical new thinking; it is the amalgam of many good software engineering practices that have preceded it, but is motivated by scenarios as the central concept driving requirements and design.

Each analytic model is instantiated by the scenario under consideration. For example, an analytic performance model for priority-based preemptive scheduling is generic, and can be taken directly from a book such as [20]. But for this model to have any meaning, it must be tailored by the particulars of the system being analyzed. The particulars that "flesh out" the analytic model such as execution times and arrival rates of requests come from the specific/derived scenario under consideration.

4.6 An Example Mapping for Modifiability

Let us now consider an example of how this mapping process works in practice. This example is taken from an analysis of a commercial revision control system, originally described in [16], and illustrated in Figure 5.

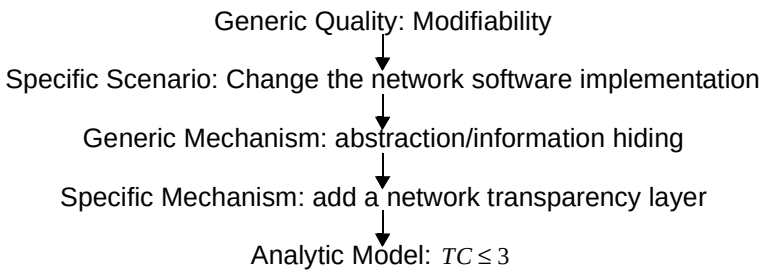

Figure 5: Mapping a Modifiability Scenario

Consider the case where a scenario has been elicited that affects the quality requirement of modifiability. The specific scenario is indirect: a system is to have its underlying network infrastructure changed. Ideally, the system should be minimally affected by this scenario, so that several different hardware interfaces and underlying protocols can be used simultaneously. This is a common requirement in systems today, and is typically met by the generic mechanism of abstraction or information hiding: the specifics of the networking infrastructure are encapsulated within an abstract interface, and the implementation details are hidden from users of the service. This generic mechanism has been realized in implemented systems in numerous ways. For example, as a network transparency layer in the World-Wide Web's original software infrastructure [32], or as the Abstract Factory design pattern [10].

The analytic model for this mechanism is related to the notion of coupling [5]: it is an attempt to measure the difficulty of future modifications involving the mechanism. In this model, the difficulty of a modification is related to the transitive closure (TC) of syntactic and semantic changes required to correctly instantiate a modification. All other things being equal, the smaller the number of changes, the easier the modification.

This is another way of saying that local changes (changes that only affect a single component) are easier to make than changes that affect many components in a system. Consider the example presented here. The scenario is the network infrastructure is changed. The mechanism ensures that this change is mitigated by a network transparency layer. As a result, we can build a model showing that the transitive closure of changes related to the scenario is less than or equal to 3. That is, changes to the underlying network infrastructure affecting the internals of the transparency layer (shown as (a) in Figure 6) might in turn affect the

layer's interface (shown as (b)) and this might in turn affect the routines that directly call this layer (shown as (c)). But this is the full extent of the changes under this scenario, if the abstraction is properly constructed and used.

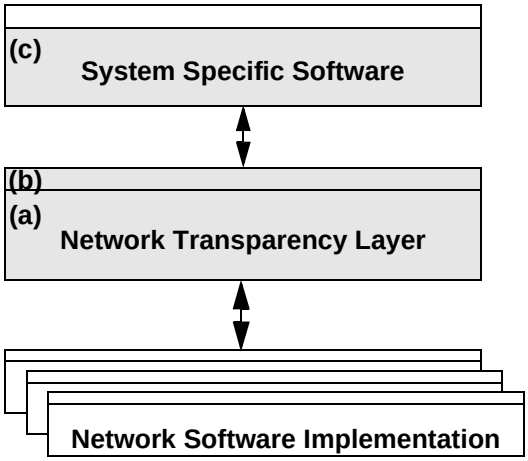

Figure 6: Propagation of Changes from a Network Transparency Layer

Different approaches to insulating the system-specific software from the various network software implementations can be modeled and compared in this way, using an analytic model that measures the impact of a change. In addition to measuring the appropriateness of a particular architecture for supporting predicted modifications, we may use the analytic models that we develop to monitor how architectural evolution over a system's lifetime affects its capability to support predicted modifications.

4.7 An Example Mapping for Usability

While usability does not admit analytic models in general, some specific aspects of usability can be modeled, as described in an architectural analysis that we conducted for an air traffic control system [24]. Consider, for example, the direct scenario where an air traffic controller issues a command and expects, not unreasonably, a response within a bounded amount of time. The assurance that this bounded response time will in fact be the case reflects a particular aspect of the system's usability [7].

A generic mechanism for meeting this scenario is to bound the computation time associated with user commands. One potential specific mechanism for realizing this generic mechanism (in the context of a multi-threaded object oriented design) is to have an independent thread that starts whenever a user command is issued, and which times out after the bounded amount of time has passed, forcing the system to respond to the user even if the original command has not yet completed. In the analytic model for this case, following [20], the response time is computed as the maximum of the computation time associated with the actual

method invocation (C_{MI}) and the time out value of the extra thread (C_{TO}). This mapping is illustrated in Figure 7.

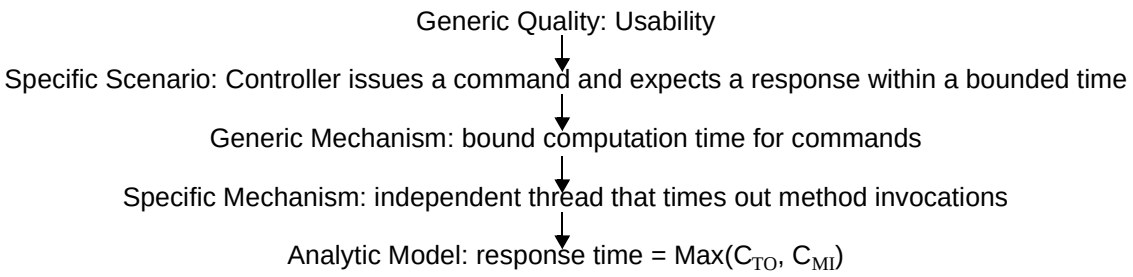

Figure 7: Mapping a Usability Scenario

While these examples are not individually complex, they exemplify the kinds of reasoning that a scenario based practice of architectural engineering promotes, and the kinds of documentation that it requires. Managing the scenarios, mechanisms, analytic models, and their mutual constraints is discussed next.

5 Tool Support for Managing Scenarios

Several aspects of the use of scenarios for architectural engineering suggest the need for tool support, which we have realized in Brie. These aspects include:

- the documentation and capture of architectures, scenarios, mechanisms and analytic models
- the management of constraints
- the management of the specialization of abstract scenarios into specific derived scenarios
- the mapping of scenarios (both abstract and specific) onto architectures
- the documentation and control of the tradeoffs among architectural qualities, as they affect the software structure of the system being designed
- the management of architectural alternatives
- the evolution of the architecture and its associated artifacts
- the need for preserving and maintaining the rationale for architectural decisions

A tool to support these activities must provide a representation language that is sufficiently expressive to capture the various entities in the system model as well as the relationships among them. A system model includes architectural constructs, such as components and connectors, as well as analysis constructs such as scenarios, assumptions and analytic models. In addition to an expressive modeling language, a tool must provide facilities for multi-user access to and revision control of these artifacts.

Ongoing work on software architecture reconstruction at the SEI has led to the development of a workbench, called Dali, for capturing and manipulating architectural models [19]. Dali was designed with an open, “lightweight” integration philosophy to promote the easy incorporation of tools appropriate to particular tasks of architectural modeling, manipulation and analysis. Although Dali was specifically designed to work with information extracted from implementation-level artifacts (such as source code, execution traces, build files, and so forth), its design is centered around a generic repository (an SQL database) that is equally ap-

propriate to the tasks of scenario-based architectural engineering. Dali's central user interaction facility is based upon an end-user programmable graph editor (part of the Rigi environment [30]). We have customized Rigi's graph editor for architecture modeling and analysis tasks. For these reasons, Dali was an appropriate foundation upon which to build Brie.

Before we may begin considering scenarios for architectural analysis, we must capture a representation of the architecture with which we are working. A great deal of research has been devoted to the consideration of architectural representation, primarily in the context of Architecture Description Languages (ADLs; [23] is a useful survey). For the purposes of scenario-based analysis, we have found that a needs-driven structural description of the views of the system's architecture is sufficient. Such a description will depict a software system's components and their interconnections, initially at an abstract level. As examples: in the context of a run-time view, components could be processes interconnected by communication paths; in the context of a source view, components could be layers or other abstract collections of source code, interconnected by function calls or message passing. The choice of views is driven by our interest in specific system qualities: a run-time view will allow us to reason about performance and availability, a source view will allow us to model modifiability, and so forth.

The software architect is in the unfortunate position of requiring a great deal of detail about a system, while at the same time requiring a high-level view. In general, we manage this conflict by *successive user-driven revelation of detail*: as we guide the architect through analyses, they will indicate potential problem areas where the architect needs to expose more detail. As mentioned above, this is a risk driven process. When a portion of the architecture is discovered, via some analysis, to harbor a risk, it is modeled at a greater level of fidelity, potentially including building a simulation or prototype of this portion.

For the purposes of demonstrating the use of Brie in supporting scenario-based architecture engineering, we will use as an example a remote temperature sensor system as presented in [2]. Briefly, this system specifies the existence of a set of furnaces and a set of operators; the operators make requests for periodic updates of the temperatures of a subset of the furnaces. The furnaces are expected to deliver temperature updates within the specified periods. Our example, as shown in Figure 8 will model architectures in which a *server* (serving the furnaces) accepts update requests from *clients* (representing the operators), performs the necessary measurements of furnace temperatures and distributes temperature updates to the clients.

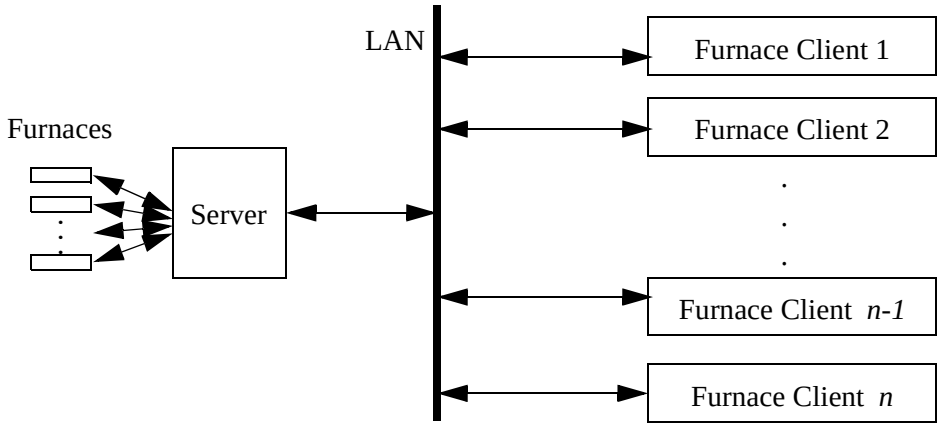

Figure 8: The Remote Temperature Sensor System Architecture

Our lexicon of modeling constructs for the purposes of this example is shown in Table 1. This lexicon comprises a relational model, and so indicates architectural elements and their relations. As we will be modeling

a simple run-time view of the system, our primary element of architectural representation is the `Process`, an autonomous thread of control executing on a piece of computing hardware. Processes are interconnected by `communicates-with` relations.

From	Relation	To
Process	**communicates-with**	**Process**
Model	**assumes**	**Assumptions**
Assumptions	**assumes**	**Assumptions**
Model	**models**	**Scenario**
Model	**uses-value**	**Value**

Table 1: The Modeling Lexicon for the Remote Temperature Sensor Example

In addition to the structural elements of architectural representation, we must provide mechanisms for the annotation of these elements with attributes. For example, servers and clients will have processing latencies and communication paths will have network latencies. For availability modeling, servers and clients will have attributes such as mean time to failure and mean time to repair. In the architectural description a system's components have named slots for these attributes. As scenarios are realized, these attributes will take on specific values.

Figure 9 depicts a view of the remote temperature sensor system's architecture, as captured using Brie.

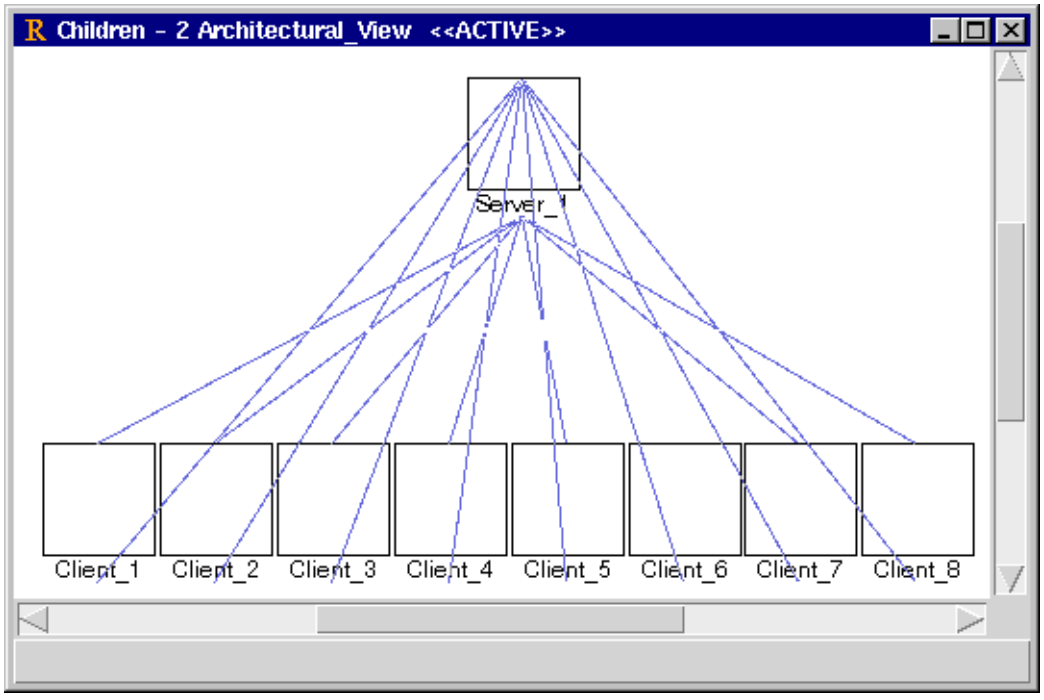

Figure 9: A Run-time View of The Remote Temperature Sensor System Architecture

To support the scenario-based analysis of software architectures, Brie must also model analysis constructs. For the purposes of our example, these are `Scenarios`, `Assumptions`, `Models` and `Values`. A `Scenario`

represents a generic scenario; it is through the enumeration of `Assumptions` that generic scenarios become instantiated as specific derived scenarios. A `Model` represents the analytic model associated with a generic scenario. Finally, a `Value` is used to represent the computation of a global attribute of the architecture. As indicated in Table 1, `Assumptions` may be related hierarchically and `Models` may depend on `Values`.

We now proceed with two examples of the application of Brie to the translation of generic qualities into generic scenarios, the instantiation of generic scenarios as specific derived scenarios, and the mapping of derived scenarios onto our example architecture. Currently, our support environment does not provide modeling facilities for explicitly capturing and reusing mechanisms. Mechanisms used to realize specific qualities are captured implicitly in the architecture and the analytic models. In the following discussion, we will identify this implicit representation.

5.1 Using Brie to Model Performance Scenarios

Figure 10 depicts the translation of the generic quality goal of "performance" into an abstract scenario that speaks to an aspect of the system's performance: during the operation of the system, furnaces will be sending updates to operators with a distribution based on the requests of the operators. We are concerned with the performance of the system with respect to its adequacy to serve the operators' requests; that is, the *periodic latency* of temperature updates. Note that this scenario is not specific to the architecture presented above; it would apply equally well to a discussion of this aspect of performance in any architecture of a remote temperature sensor system.

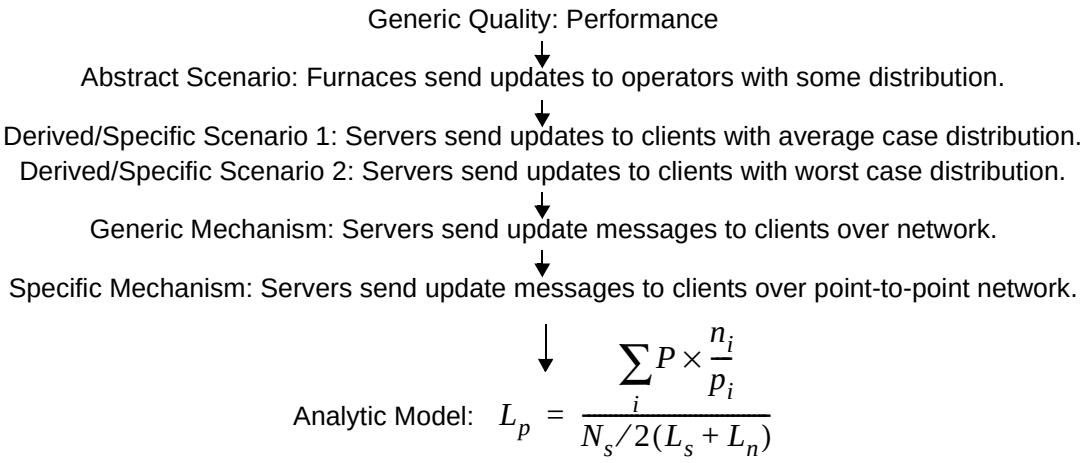

Figure 10: Mapping a Performance Scenario

The abstract scenario is then mapped onto two derived scenarios that *do* depend both on the architecture under consideration and on particular assumptions about this architecture's context. In this case, the abstract scenario specified the distribution of updates as the key abstraction; the derived scenarios instantiate the abstract scenario by specifying particular distributions: average case and worst case.

The generic mechanism by which this scenario is accommodated in the architecture is message-passing over a network. Specifically, the mechanism used in this example is a point-to-point network which serves requests from the processes in a first-in-first-out sequential fashion [2].

Based on the chosen mechanism, the abstract scenario, and its realization as derived scenarios, we may now construct an analytic model that computes the periodic latency of temperature updates for a client from a distribution of client requests. The update distribution is specified as a set of pairs (n_i, p_i) where n_i cli-

ents have requested updates with a period of p_i seconds (each p_i in the set is distinct). The formula computes the periodic latency by summing the contributions of each update period to the number of updates expected in a particular measurement period, P, then dividing by twice the sum of the server latency and the network latency. For a detailed discussion of the derivation of this formula, see [2].

Again, consideration of architectural evolution is important: as the architecture evolves, we may benchmark sets of scenarios and their associated analyses to assess how an architectural change affects its quality attributes. It is also important to point out that changes to an architecture will almost invariably affect more than one quality attribute. For example, introduction of a network transparency layer (as discussed in Section 4.6) will positively impact modifiability (with respect to a particular scenario) but will likely negatively impact performance.

Figure 11 shows a representation of the example mapping in Brie. Our abstract scenario, `Periodic_Latency_Performance_Scenario`, is shown on the left side of the figure. This scenario is modeled by the `Periodic_Latency_Performance_Model`, which has two associated sets of assumptions, namely: `WCPL_Performance_Assumptions` and `ACPL_Performance_Assumptions`, representing the worst case distribution and average case distribution derived scenarios respectively. Note that these assumptions depend on a set of generic performance assumptions, `Performance_Assumptions`, which simply capture assumptions common to both derived scenarios. The model also depends on a value, `num_servers`, that represents the computation of a global attribute of the architecture, the number of servers. This explicit dependency allows recomputation of appropriate analytic models when the architectural model changes: when the number of servers is changed, the `num_servers` value is first recomputed by Brie, and then dependent models are recomputed.

Associated with each element in the system model is a textual annotation describing the element. For example, the abstract scenario description is captured in an annotation on `Periodic_Latency_Performance_Scenario`. The assumptions that realize derived scenarios comprise a set of parameters that specify the values of attributes in the architectural model as well as other values that specify the scenario. For example, associated with `Performance_Assumptions` are values that give the server and network latencies that were specified as attributes of the architectural model; associated with `ACPL_Performance_Assumptions` is the specific distribution of temperature updates that will realize the average case scenario. All of these are descriptions of, and static attributes of, architectural elements. To support understanding of and comparison of these models, and the tradeoffs that they harbor, Brie can associate an executable script with each model that calculates the values of various attributes of interest.

For example, associated with the `Periodic_Latency_Performance_Model` is a script written in a combination of SQL and Perl. This script executes in three phases: first, it computes the assumption and value dependencies for the model; second, it extracts from the architectural model (as stored in the repository) the attributes required for computation of the analytic model and maps them to values specified by the assumptions; and third, it computes the formula specified by the model for each set of assumptions and stores the results. Assumptions and values are made available to the formula, written in Perl, as variables in the Perl environment; this environment is maintained by another Perl script that, in addition, manages the process of executing the model script phases. In the case of our example, the model execution proceeds as follows:

- the assumptions and values for the analytic model as specified within the analysis model are identified, extracted from the repository, and added to the environment. In the case of our example, this would include the `num_servers` value, the network and server latencies taken from `Performance_Assumptions` and the update distributions as specified by the derived scenario assumptions.
- the relevant attributes of the architectural model are extracted and mapped to the assumptions. In our example the network and server latencies specified in the architecture would take on the values from the previous step.

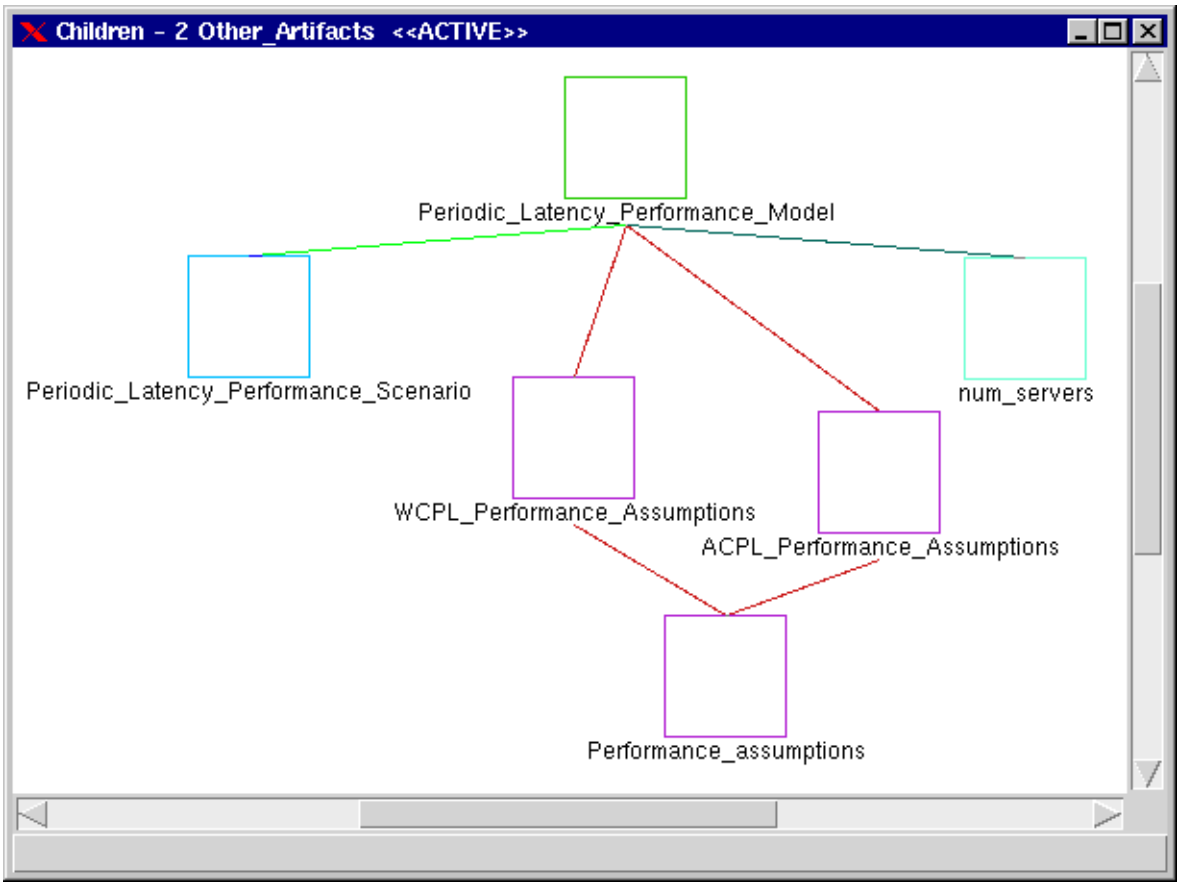

Figure 11: The Periodic Latency Model

- the model is computed for *each set* of assumptions. In this case, the model computes the formula from Figure 10 from the assumed temperature update distribution, the number of servers and the assumed latencies. The results are stored in the repository as attributes of the `Periodic_ Latency_Performance_Model`.

Although not currently supported by the tool, the next step in the scenario-driven architectural engineering process would be validation of the results of the analytic models against requirements. We envision modeling requirements as first-class entities within the tool and automating the process of verifying computed values against them. Similarly, we envision the explicit capture of constraints among various aspects of the model, such as structural constraints on the architectural representation and value constraints among the sets of assumptions representing derived scenarios.

In addition, to support a real world architectural engineering process, we need to include facilities for managing architectural evolution. Such facilities would include benchmarking sets of scenarios and their associated analyses and then using these benchmarks to assess the architecture as it evolves.

5.2 Using Brie to Model Availability Scenarios

Figure 12 shows the mapping of another generic system quality, availability, to an abstract scenario representing an aspect of this quality, “System components fail and are repaired”. This scenario embodies our concern regarding the overall availability of the system, which would typically be specified as a requirement in terms of yearly downtime. We then realize this abstract scenario as two specific derived scenarios, each

related to a particular mode of system component failure: a server power supply failure and a server software failure. Of course, there are many other possible failure modes, such as client failures, which would be represented by additional derived scenarios in a complete model of the system.

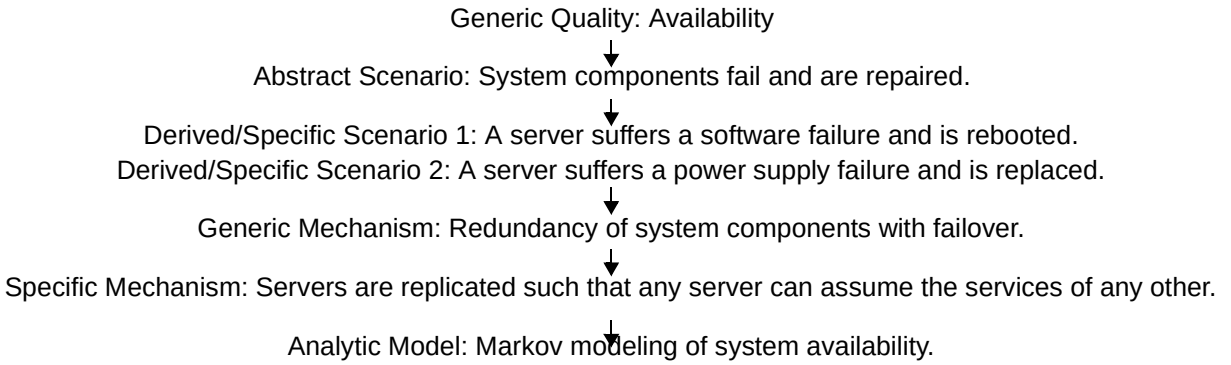

Figure 12: Mapping an Availability Scenario

The generic mechanism by which these scenarios are admitted to the design of the system is redundancy of system components with failover. This generic mechanism is realized in our architecture by server replication (not shown in Figure 9) such that, upon failure of a server, another server may begin distributing temperature updates in place of the failed server.

Based on the abstract scenario and the chosen mechanism, an analytic model may now be constructed to measure system availability in the presence of component failure. A common way to model availability—failures and repairs in the terms of the abstract scenario—is to use a Markov model. Figure 13(a) shows the Markov model for the single-server architecture of Figure 9 (this is, of course, a degenerate case of the server replication mechanism). The system begins operation in state **S**, with the server operating, and transitions into the **F** (failure) state with probability λ_s, based on the mean time to failure of the server. The transition from the **F** state to the **S** state occurs with probability μ_s, based on the mean time to repair of the server. The server availability is computed by solving the system of differential equations defined by the Markov model. This analytic model realizes the two derived scenarios given in Figure 12.

It is important to note that the structure of the Markov model will change when servers are added; Figure 13(b) shows the Markov model for a two-server architecture. Although in this case the analytic model could have been made sufficiently general to accommodate variation in the number of servers, the example illustrates an important point: there are occasions when an analytic model will apply only within the bounds of certain architectural constraints. These could also include constraints on attributes such as “the model only applies when the server latency is less than the network latency”. Because models sufficiently generic to handle every circumstance are not realistic, it is important to record these constraints as *ranges of applicability* for all analytic models. Ideally, a support environment would enforce these constraints.

Figure 14 shows the representation of our availability mapping in Brie. The assumptions specifying the software failure derived scenario are shown as `Software_Failure_Availability_Assumptions` while the power supply failure scenario is shown as `Power_Supply_Failure_Availability_Assumptions`. Currently our availability analytic model is not sufficiently general to support variability in the number of servers; this explains the absence of a dependency on the `num_servers` value, as was present in the performance ex-

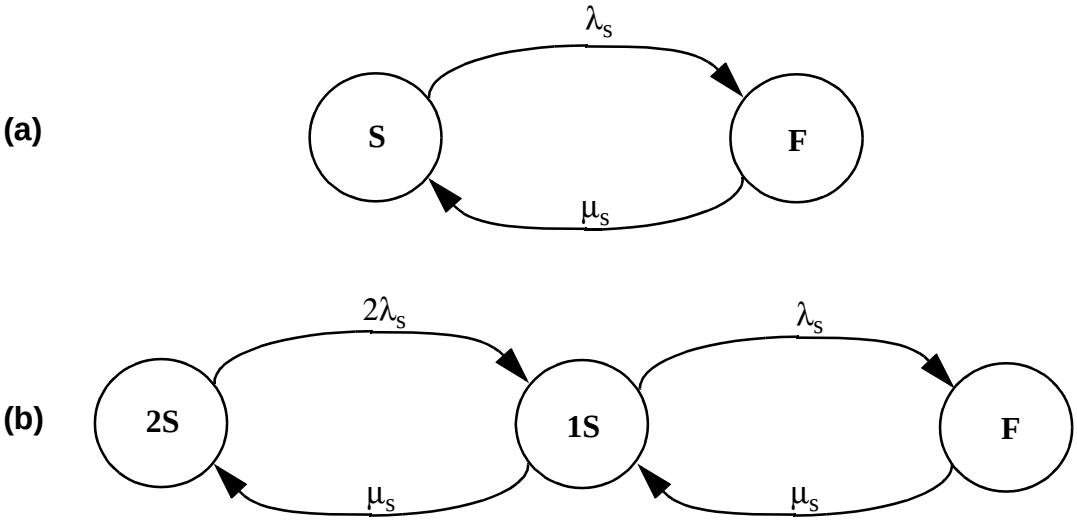

Figure 13: Markov Models for One and Two Server Architectures

ample. The assumptions representing the derived scenarios specify values for the mean time to failure and mean time to repair for the server.

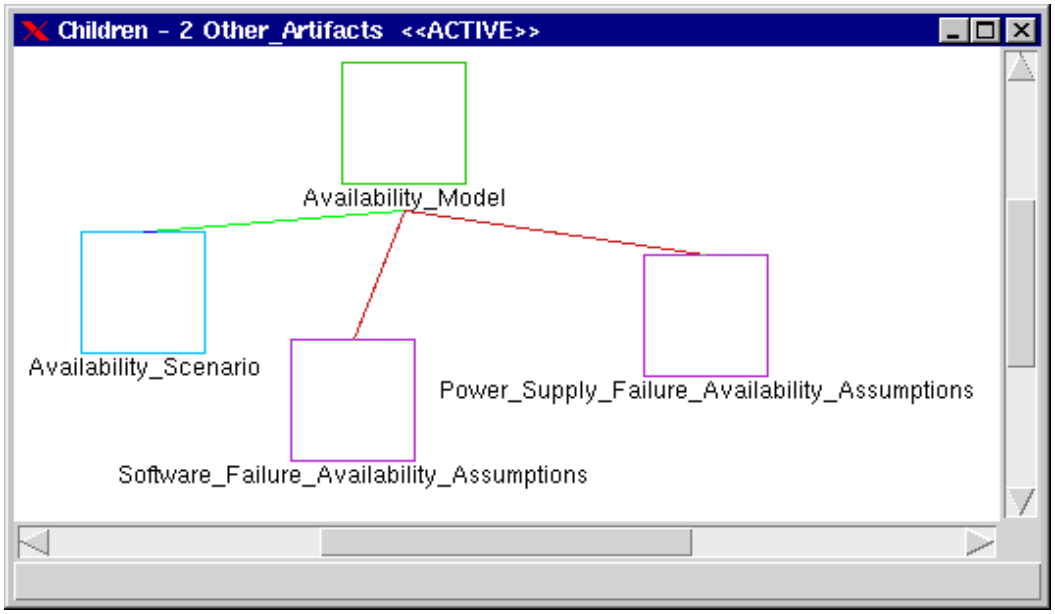

Figure 14: The Availability Latency Model

In the previous example, the analytic model was sufficiently simple to be computed directly by the model script. However, there are cases for which an external computation engine, such as a pre-existing tool or mathematics package, is more appropriate. In these cases, the first and second phases of model computation proceed as described above and the third phase changes as follows: rather than computing the analytic model directly, the model script exports the appropriate data to some external representation, invokes an external tool to carry out the computation, and imports the results. The results are then saved to the re-

pository as before. In the case of the `Availability_Model` for our example, we used Mathematica to solve the differential equations from the Markov model [2].

5.3 Building a More Complete Analytic Model of a System

Figure 15 depicts a more complete model of the remote temperature sensor system that includes our two previous examples, periodic latency and system availability. Also shown are two additional performance-related scenarios with their models: control latency (how long the server takes to respond to a client's control request) and jitter (the maximum delay between consecutive temperature updates). Note that a relationship is shown between `Jitter_Perf_Model` and `Periodic_Latency_Performance_Model`; this relationship indicates a direct dependency between the two *models*, in much the same way as a model depends on a set of assumptions. In this case, the jitter model is computed from the results of the periodic latency performance model. In general, a complete model of a given system will comprise a web of assumptions, scenarios and analytic models.

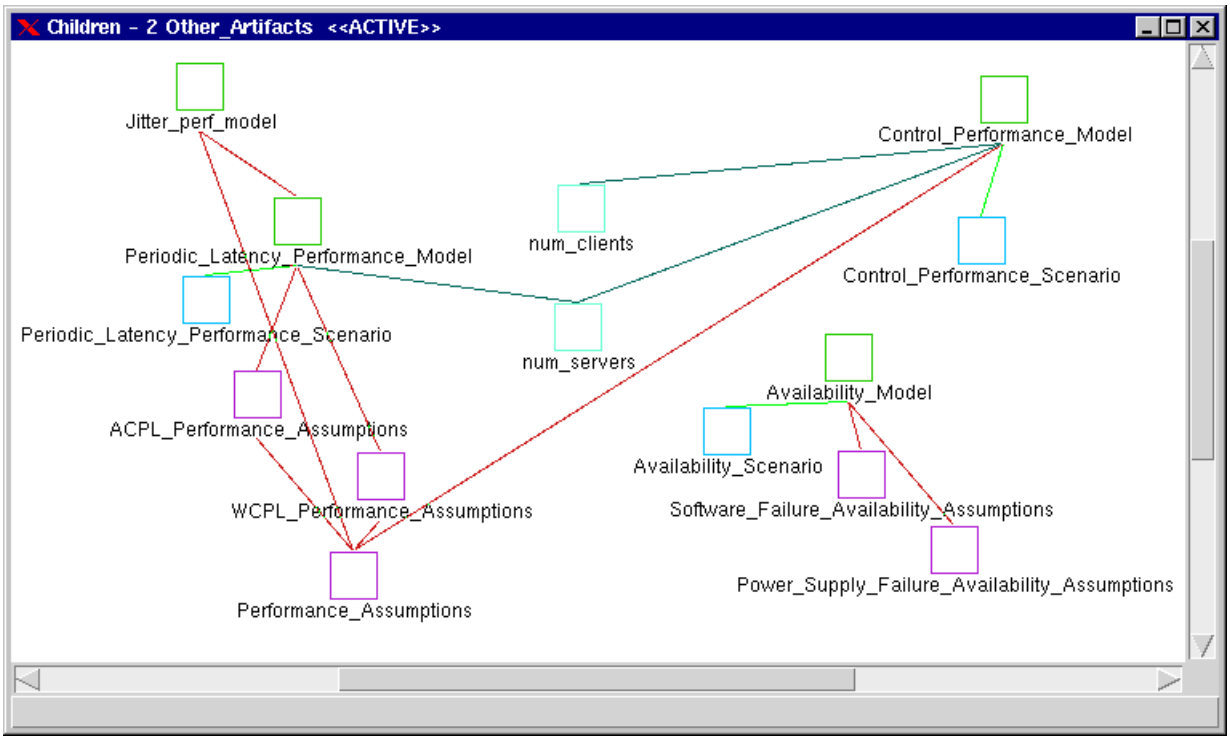

Figure 15: A More Complete Remote Temperature Sensor System Model

6 Related Work

While much has been written on the uses of scenarios and use cases in object-oriented analysis and design, these uses have been constrained almost entirely to a single goal: that of eliciting and validating functional requirements (e.g. [11]). By way of contrast, little has been written on the use of scenarios in eliciting, driving, and validating quality requirements, with the exception of the use of scenarios for analyzing a software architecture's modifiability [16].

Even less has been written on how scenarios shape the requirements for a family, or product line, of systems. While we have gathered much anecdotal evidence, through architectural reviews and risk assessments, that this is done informally in almost every significant system development, the lack of formality of the process limits its usefulness. Scenarios do not get created and circulated throughout a large group of stakeholders, and they seldom form the basis for formal reviews of the software architecture, as we have advocated ([3], [16]).

Domain analysis activities (e.g. [25]) do collect the uses of and requirements for sets of related components and architectures, but they do not connect these analyses to architectures. The process described in this paper is a bridge between domain analysis and architectures.

The most complete set of literature on using scenarios for motivating software design centers around the practice of object oriented design. The major uses of scenarios and relatives of scenarios in object oriented design are:

- scenarios as a sequence of related events that represent dialogs between the user and the system, as exemplified by the widely used OMT notation and its associated design paradigm [28];
- scenario as a “control flow diagram”, as promulgated by Firesmith *et al* [9];
- use cases, as employed by Jacobson *et al*, which include not only the notion of a scenario as a control flow, but also a set of pre- and post-conditions under which this use case applies [14];
- use case maps, as defined by Buhr and Casselman, which are a means of representing and reasoning about the flow of responsibility through a system [6].

Each of these uses of direct scenarios aids the architect in understanding how a software architecture supports its requirements; they are a means of mapping requirements onto an object oriented design in a way that lets stakeholders “walk through” their execution.

Kruchten has gone one step further in the “4+1” model, proposing scenarios as the glue that aids in understanding a software architecture, in driving the various views needed to represent it, and in binding the views together [21]. Kruchten’s work is most closely related to the architectural engineering process described here.

Other work has focused on evaluating tradeoffs, the quality of tradeoffs and the perspective of those making the tradeoffs. Notable examples include Boehm’s Win-Win model [4] which seeks to reconcile customer and developer expectation and capability differences, and Hauser and Clausing’s “House of Quality” [12] work on Qualify Function Deployment intended to assure the design quality of a product while it is still in the design stage.

tradeoffs and qualities and tradeoffs .... ie end goal of analysis and a tool to support analysis ... win-win, house of quality stuff ... typically from perspective of requirements not design tradeoffs ...

The authors are firm believers in the utility of applying appropriate tools to the process of architectural analysis. As part of a larger research effort at the Software Engineering Insitute, we have been involved in the process of applying and maturing a method known as the Architectural Tradeoff Analysis Method (ATAM) [17]. ATAM seeks to clearly specify both a process and integrated tool support for the task of formally specifying the impact of software architectural decisions on system quality attributes such as survivability, security, performance, dependability, and maintainability. Key aspects of this work include the creation of an appropriate set of tools to support both the dynamic and static analysis of existing software and mapping to architectural representations, to accomodate the visualization of large systems, and to adequately represent the aspects of software systems most related to their particular quality attributes. In recent work we

have described in more detail the use of tools [19], representations [31], the role of tools in the large process of architectural evolution [18], and a predecessor ATAM process known as SAAM [3].

7 Conclusions

Five years of experience in performing architectural evaluations at the Software Engineering Institute have convinced us that scenarios are useful for understanding an architecture, for uncovering its flaws and limitations, for driving us to the appropriate architectural views and the right level of representation of these views, for mapping requirements onto an architecture, and for understanding the implications of anticipated changes on the architecture.

Although we have seen many examples of the kinds of reasoning described in this paper used in industry, it is invariably done in an *ad hoc*, experience based, “back of the envelope” fashion. While these techniques have often sufficed in past development efforts, they are becoming less adequate for the ever increasing size of software systems. Our goal has been to *systematize* an existing *ad hoc* practice; to move it from a craft to an engineering discipline. For example, our elicitation matrix promotes the consideration of a wide variety of scenarios and stakeholders’ concerns in a systematic fashion. Our mapping from scenarios to mechanisms to analytic models ensures that design decisions and their rationale are documented in such a fashion that they can be systematically explored, varied, and potentially traded off against each other. As systems evolve, the analytic models can be used to assess the impact of architectural changes, relative to the system’s quality goals. Constructing analytic models to reflect the various scenarios and mechanisms in the system provides a technique for doing so.The use of scenario driven analytic models has proven to be both necessary (to make the right design decisions) and tractable (because the process is risk driven; we only build analytic models in areas deemed to be of high risk).

Finally, the consideration of so many factors and the need for documenting the process has driven us to create tool support for architectural engineering. This is because the process is complex, with many requirements to trace, analyses to consider, and constraints to manage.

8 Acknowledgments

The following people contributed greatly to the ideas contained within this paper, and we are indebted to them: Mark Klein, Mario Barbacci, Len Bass, Joe Batman, Paul Clements, Craig Meyers, Dan Plakosh, Pat Place, Reed Little, and Linda Northrop.

9 References

[1] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop, A. Zaremski, “Recommended Best Industrial Practice for Software Architecture Evaluation”, Carnegie Mellon University, Software Engineering Institute Technical Report CMU/SEI-96-TR-25, 1996.

[2] M. Barbacci, J. Carrière, R. Kazman, M. Klein, H. Lipson, T. Longstaff, C. Weinstock, “Toward Architecture Tradeoff Analysis: Managing Attribute Conflicts and Interactions”, Software Engineering Institute, Carnegie Mellon University Technical Report CMU/SEI-97-TR-29, 1997.

[3] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Reading, MA: Addison-Wesley, 1997.

[4] B. Boehm and H. In, “Aids for Identifying COnflicts Among Quality Requirements”, IEEE Software, March 1996.

[5] L. Briand, J. Daly, J. Wuest, “A Unified Framework for Coupling Measurement in Object-Oriented Systems”, IEEE Transactions on Software Engineering, 25:1, Jan/Feb, 1999, 91-121.

[6] R. Buhr, R. Casselman, *Use Case Maps for Object-Oriented Systems*, Upper Saddle River, NJ: Prentice Hall, 1996.

[7] G. Cockton, C. Gram (eds.), *Design Principles for Interactive Software*, Chapman & Hall, 1996.

[8] A. Davis, “Requirements Engineering“, in (J. Marciniak, ed.), *Encyclopedia of Software Engineering*, Vol. 2, Wiley: New York, 1994, 1043-1054.

[9] D. Firesmith, *Object-Oriented Requirements Analysis and Logical Design*, New York: John Wiley & Sons, 1993.

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[11] P. Gough, F. Fodemski, S. Higgins, S. Ray, “Scenarios - an Industrial Case Study and Hypermedia Enhancements”, *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, York, England, March, 1995, 10-17.

[12] J.R. Hauser and D. Clausing, “The House of Quality”, Harvard Business Review, May/June 1998, 63-73.

[13] M. Jackson, *Software Requirements and Specification*, Addison-Wesley, 1995.

[14] I. Jacobson, M. Christeron, G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Reading, MA: Addison-Wesley, 1992.

[15] R. Kazman, G. Abowd, L. Bass, M. Webb, “SAAM: A Method for Analyzing the Properties of Software Architectures,'' in *Proceedings of the 16th International Conference on Software Engineering*, (Sorrento, Italy), May 1994, pp. 81-90.

[16] R. Kazman, G. Abowd, L. Bass, P. Clements, “Scenario-Based Analysis of Software Architecture”, *IEEE Software*, Nov. 1996, pp. 47-55.

[17] R. Kazman, M. Barbacci, M. Klein, S. J. Carrière, S. Woods, “Experience with Performing Architecture Tradeoff Analysis”, *Proceedings of the 21st International Conferene on Software Engineering,* Los Angeles, CA, May 1999, to appear.

[18] R. Kazman, S. Woods, S. J. Carrière, “Requirements for Integrating Software Architecture and Reengineering Models: CORUM II”, *Proceedings of the Fifth IEEE Working Conference on Reverse Engineering*, Honolulu, HI, 1998.

[19] R. Kazman, S. J. Carrière, “Playing Detective: Reconstructing Software Architecture from Available Evidence”, *Journal of Automated Software Engineering*, 6:2, April 1999, 107-138.

[20] M. Klein, T. Ralya, B. Pollak, R. Obenza, M. Gonzales Harbour, *A Practitioner’s Handbook for Real-Time Analysis*, Kluwer Academic, 1993.

[21] P. Kruchten, “The 4+1 View Model of Software Architecture”, *IEEE Software*, Nov. 1995, pp. 42-50.

[22] J. McCall, “Quality Factors”, in (J. Marciniak, ed.), *Encyclopedia of Software Engineering*, Vol. 2, Wiley: New York, 1994, 958-969.

[23] N. Medvidovic, “A Classification and Comparison Framework for Software Architecture Description Languages”, University of California at Irvine Technical Report UCI-ICS-97-02, 1997.

[24] B. C. Meyers, D. Plakosh, P. Place, M. Klein, R. Kazman, “Assessing Multiple Designs for the FAA En Route Center Architecture”, Software Engineering Institute, Carnegie Mellon University Technical Report CMU/SEI-98-SR-02, 1998.

[25] R. Nilson, P. Kogut, G. Jackelen, “Component Provider’s and Tool Developer’s Handbook Central Archive for Reusable Defense Software (CARDS)”, STARS Informal Technical Report STARS-VC-B017/001/00, Unisys Corporation , March 1994.

[26] Object Management Group, *CORBAservices: Common Object Services Specification*, 1997.

[27] J. Poulin, “Domains, Product LInes and Software Architectures: Choosing the Appropriate Level of Abstraction”, *Proceedings of the 8th International Workshop on Software Reuse*, (Columbus, OH), March 1997.

[28] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Pentice Hall, 1991.

[29] C. Smith, L. Williams, “Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives”, *IEEE Transactions on Software Engineering*, 19(7), pp. 720-741.

[30] K. Wong, S. Tilley, H. Müller, M. Storey. “Programmable Reverse Engineering”, International Journal of Software Engineering and Knowledge Engineering, 4(4), pp. 501-520, December 1994.

[31] S. Woods, L. O’Brien,T. Lin, K. Gallagher, A. Quilici, “An Architecture for Interoperable Program Understanding Tools“, Proceedings of the Sixth IEEE International Workshop on Program Comprehension”, Ischia, Italy, 1998.

[32] World Wide Web Consortium, “W3C Reference Library“, http://www.w3.org/Library.