Augmenting Architectural Modeling to Cope with Uncertainty

Orieta Celiku
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15313
+1 412 268 3501
orietac@cs.cmu.edu

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15313
+1 412 268 5056
garlan@cs.cmu.edu

Bradley Schmerl
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15313
+1 412 268 5889
schmerl@cs.cmu.edu

ABSTRACT
Notations and techniques for architectural modeling and analysis have matured considerably over the past two decades. However, to date these approaches have primarily focused on architectural properties and behavior that can be precisely defined. In this paper we argue that it is possible to augment existing architecture description languages (ADLs) to support reasoning and analysis in the presence of uncertainty. Specifically, we outline two basic extensions to formal architecture descriptions that take advantage of probabilistic specifications to support architecture-based analyses such as simulation, detection of behavioral drift, and reasoning about the expected outcomes of uncertain behavior. An important property of these specifications is that they allow incremental refinement – as more is known about the behavior of the system, specifications can be extended without invalidating previous analyses.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: *Languages (e.g., description, interconnection, definition)*

General Terms
Design, Languages

Keywords
Software Architecture, probabilistic properties, simulation, analysis, behavior.

1. INTRODUCTION
Architectural description languages (ADLs) have come a long way since their introduction in the 1990s. Numerous ADLs have been developed [6], several are in industrial use, and several have been incorporated into international standards [2][7]. They support analyses as diverse as identifying potential deadlocks between components, analyzing real-time behavior, detecting violations in security flow policies, discovering performance bottlenecks, and guaranteeing conformance of a system's architecture to an architectural style or product line framework.

However, most approaches to modeling systems at the architectural level adopt formalisms that require precise specification of the properties and behavior of the system under construction. For instance, we might analyze specifications of the timing properties of components and connectors to determine system throughputs and latencies. While a few of these, such as queuing-theoretic analyses, incorporate stochastic specifications, the use of probabilistic or randomized behavior is limited to a very specific kind of analysis, rather than being generally applicable to architectural analysis as a whole.

As a result, it is difficult to use architectural specifications to reason about system properties in the absence of precise knowledge about properties of the system elements. This in turn limits the usefulness of architectural description in representing and reasoning about systems where there is considerable uncertainty, but where we would like to understand the *expected* behavior even if we cannot know the exact outcome. For example, in a service-oriented architecture the actual service provided for a client request may depend on a number of dynamic factors that cannot be known at design time. Nonetheless, if we have some information about the possible services and their properties, as well as the likelihoods of their being selected, we should be able to reason about some aspects of the system. As another example, at an early design stage we may have only a vague idea of the behavior of some components. Instead of postponing analysis until we are sure about that behavior, we might want to do initial analysis that incorporates what we *do* know, and refine that later as more information becomes available.

In order to handle examples like those just mentioned two things are needed. First, we need ways to *represent uncertainty at the architectural level*. Second, we need ways to *take advantage of that information in support of useful, general-purpose analyses.*

In this paper we summarize our initial results in doing this. Focusing specifically on the problem of characterizing behavior with probabilistic outcomes, we describe two techniques. The first is to augment property specification in ADLs with the ability to incorporate uncertainty in those properties. This allows us to talk formally about a range of possible behaviors for a given property. The second is to augment behavior descriptions to explicitly

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
International Workshop on Living with Uncertainty, Nov. 5, 2007, Atlanta, GA, USA.
Copyright 2007 ACM 1-58113-000-0/00/0004…$5.00.

account for probabilistic behavior. This will allow us to attach probabilities to system traces and state changes so we can reason about expected behavior as a random variable (in the probabilistic sense). We briefly illustrate the utility of these extensions by showing how they can form the basis for architecture-based simulation, run-time detection of behavioral drift, and formal reasoning about the expected outcomes of uncertain behavior.

2. ARCHITECTURAL UNCERTAINTY

As indicated above, there are two aspects to our approach to dealing with uncertainty: allowing uncertainty to be assigned to properties and using those values in analysis and monitoring; and augmenting behavior descriptions to explicitly account for probabilistic behavior.

2.1 Architectural properties as distributions

For the purpose of many architectural simulations and analyses we can represent uncertainty as probability distributions. For example, the arrival rate of requests into a server might be represented as a normal distribution with some mean and variance. In past work with analysis [11], it was assumed that the values specified represented the mean in the normal distribution, and the variance was assumed to be some unspecified constant value. Clearly, a more accurate specification of the distribution would make for more-meaningful analyses.

In modern ADLs it is often possible to define new property types. In the Acme ADL [3], for example, we are able to define property types that can capture these distributions, and then use them in the same way as built-in property types. To take advantage of this, we define a family (called DistributionFamily) that specifies the new distribution property types; architectural designs that make use of this family can use these new property types by incorporating that family into their definitions. For example, Figure 1 provides the definition of a property type for a normal distribution property and an example of its use to specify the property arrival-rate. In our definition of DistributionFamily, we have included three common types of distributions: Normal, Exponential, and Weibull. The family can of course be extended with other types of distributions.

```
Property type NormalDistribution = Record [
    mean : float;
    stddev : float;
]
...
property arrival-rate : NormalDistribution =
    [mean=100; stddev=10;];
```

Figure 1. Specifying Normal Distribution Type in Acme.

To make the entry of distributions easy to use, we provide an extension point in our architecture development environment [9], that allows clients to add custom user interfaces for entering property values of certain types. Figure 2 shows how this looks. The user, once the NormalDistribution property type has been selected, can input the mean and standard deviation. Once the value is entered in this fashion, the custom UI will generate the description shown in Figure 1 for the arrival-rate property.

Now that we can enter probability distributions for properties in an architectural design, there are several things that we can do with them, including:

1. Compare observed values with predicted distributions; and

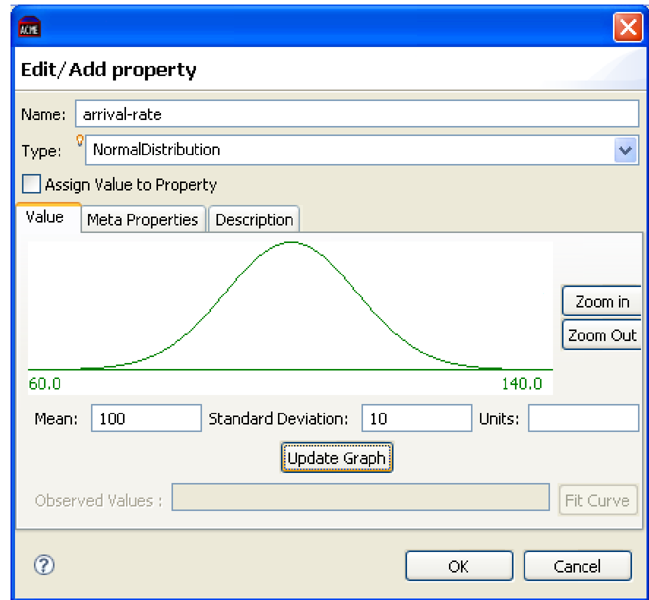

Figure 2. Interface for defining a normal distribution.

2. Use the values in probabilistic simulations and analyses to give more accurate results.

In previous work on architecture-based dynamic adaptation [1][10], we have shown how to connect observations of running systems to property values in an architectural description to determine whether changes should be made to executing systems. We can use this technology and the probability distribution extensions to compare the distribution of the observed results with the expected results, and can use standard statistical techniques to calculate the deviation between the expected and observed values (see Figure 3). Such calculations would be useful in deciding whether to repair the system under observation or report errors in the architecture.

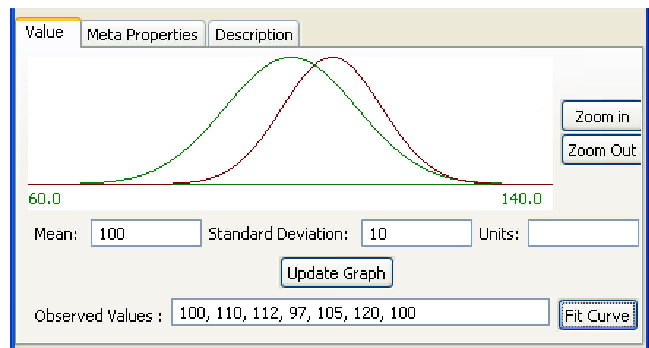

Figure 3. Comparing expected and observed distributions.

Without probabilistic properties, our repairs tend to be *reactive*: once a certain value is observed, a repair will be initiated.[1] With probabilistic properties, we have a way of smoothing out these

[1] In actuality, the values were smoothed by hard-coding probability distributions into *gauges* – elements that report values to the architecture.

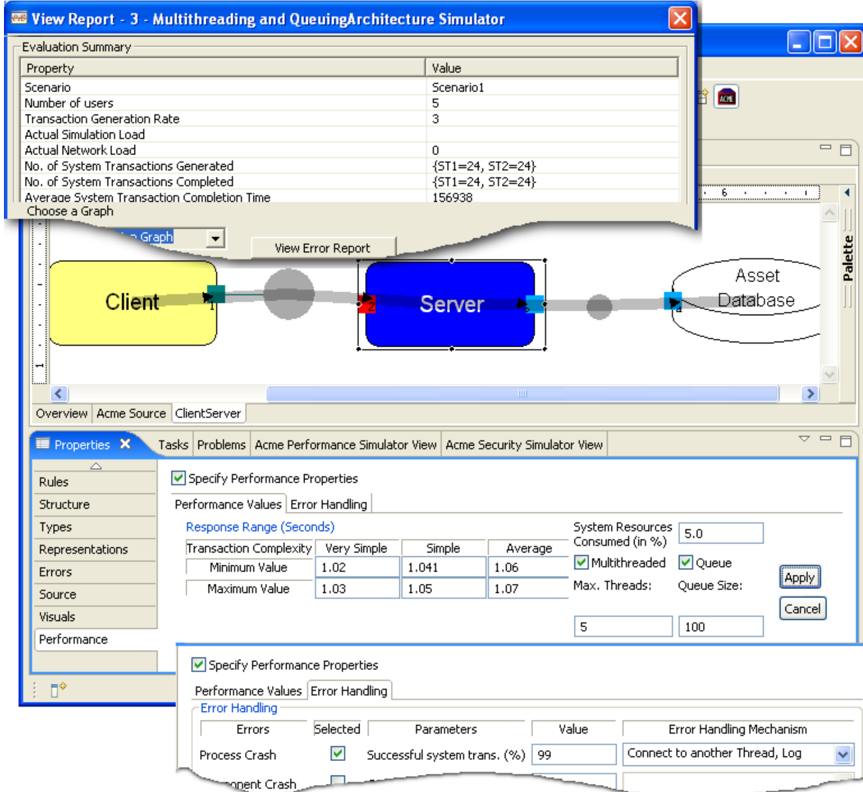

Figure 4. Specifying Performance Attributes.

values by instead reacting when the error in the distribution is above a certain threshold.

2.2 Using distributions in analyses

We can take advantage of probabilistic properties in various simulations and analyses. For example, we have developed a performance simulation tool (illustrated in Figure 4) that allows one to define parameters such as the processing times for components, the transmission rates for connectors, and error rates as probability distributions and then use these distributions as the basis for a Monte Carlo-style simulation. In such a simulation the system is symbolically executed repeatedly using inputs and property choices chosen in accordance with the distributions. The final result is a report indicating how many requests were generated by the simulation, and how many of those requests were processed or failed. It also provides feedback on which parts of the architecture are overloaded.

In Figure 4, the response range on properties can be of various categories (e.g., Very Simple, Simple, Average, Complex), each with a minimum and maximum response time. Currently, a normal distribution is calculated based on these ranges, but these could be easily specified directly as probability distributions. The Monte Carlo simulation would then use these distributions instead of the hardwired normal distribution.

We use a similar approach in security analysis simulations. Architects define paths through the architecture that particular threats (such as viruses, DoS attacks) can take. In addition, the architect indicates which components are assets and countermeasures. Value is assigned to assets and effectiveness against threats are assigned to countermeasures as probabilistic properties. Furthermore, probabilities are assigned to paths in the architecture. Monte Carlo simulation is then performed to determine the most probable damage to each of the assets in the transactions.

2.3 Probabilistic uncertainty in architectural behavior

We now describe how architectural descriptions can be augmented to explicitly incorporate probabilistic uncertainty, the kind of properties one may reason about in this context, and how incremental refinement of specifications can be used to make specifications more precise as we learn more about them.

We adopt the formal framework of *probabilistic action systems* developed by McIver [4]. The essential idea behind this framework is that, in addition to standard choices in behavior, we are also given access to probabilistic choices, which represent the frequency with which branches of a choice are executed. Specifications in this framework are written as collections of guarded actions that execute in parallel. The semantics of the language extends standard assertion-based reasoning with probabilistic analysis over a suitable probability space on computation paths [5]. Specifications can thus be seen as mapping initial states to (sets of) probability distributions over final states.

The main benefit of McIver's formalism is that it supports the notion of incremental refinement, which we illustrate below. Moreover, reasoning in the framework can be automated, using both theorem provers and model checkers.

As an example consider a simple client-server system, where a broker is used to connect a client with one of two possible servers.

Client ==

  var *c*: {*wait,sent,received*}
  initially *c* := *wait*
  request: (*c* = *wait*) → *c* := *sent*
  receive: (*c* = *sent*) → *c* := *received*

The client can execute two actions: request a service (when it is waiting), and receive a service once a request is made.

Broker ==

  var *r*: {*listen,serve,serving,served*}
  initially *r* := *listen*
  request: (*r* = *listen*) → *r* := *serve*
  serviceArequest: (*r* = *serve*) → *r* := *serving*
  serviceBrequest: (*r* = *serve*) → *r* := *serving*
  serviceAreceive: (*r* = *serving*) → *r* := *served*
  serviceBreceive: (*r* = *serving*) → *r* := *served*
  receive: (*r* = *served*) → *r* := *listen*

The broker listens on variable *r*, and chooses to serve the client by executing one of the actions *serviceArequest, serviceBrequest*. Since both these actions may be enabled simultaneously (when *r*

= *serving* and both servers are waiting) either of them may execute, and we assume that we have no information about how this kind of nondeterminism is resolved.

ServerA ==
  var s_a: {*wait,serving,served*}
  initially s_a := *wait*
  serviceArequest: (s_a = *wait*) → s_a := *serving*
  serviceA: (s_a = *serving*) → s_a := *served* $_{pA}\oplus$ *skip*
  serviceAreceive: (s_a = *served*) → s_a := *wait*

The statement s_a := *served* $_{pA}\oplus$ *skip* is an example of an explicit probabilistic choice and is interpreted as follows: s_a := *served* is executed with probability *pA* and the statement *skip* (which leaves all the variables unchanged) is executed with probability 1-*pA*. In this situation the probabilistic choice expresses that *ServerA* chooses to delay serving the client with probability 1-*pA*. *ServerB* is specified analogously, and chooses to delay serving the client with probability 1-*pB*.

The specified components are called action systems [4]. The entire client-server system is expressed as the parallel composition of the specified action systems:

*Client-Server*1 == *Client* || *Broker* || *ServerA* || *ServerB*

We assume that the components of this parallel composition synchronize on the entire set of action labels; for example, for a client to execute *request*, the broker must also simultaneously execute its *request* action. The declared variables are assumed to be global. Parallel composition of action systems is made under standard noninterference assumptions (e.g., two synchronizing actions do not update the same variable simultaneously).

What properties can we reason about given our specification? Can we guarantee with *absolute* certainty that the client will be eventually served? When probabilities are in the picture we cannot guarantee properties with absolute certainty. However, we can reason about the probability with which properties hold (or more generally the expected value of some random variable of interest). For example, the semantics of the action systems above [4] allows us to prove that when *pA* and *pB* are positive constants (or in case they are state functions they are bounded away from 0) the client will eventually be served with probability 1; in other words the client will not be served with vanishing probability. (Note that if we were to use standard choice, instead of the probabilistic one, to specify the *service* action (written *s* := *served* [] *skip*) we would have no information at all about how the choice would be resolved.) Other properties that we can check our specification against include various probabilistic versions of temporal properties. The guarantees on such properties are usually expressed as "the least probability with which the property holds."

Now let us assume that the servers are augmented with clocks (t_a, and t_b respectively) that count the number of times that the server "stutters" on action *service* before sending back a response.

*ServerA*1 ==
  var s_a: {*wait,serving,served*},
    t_a: *Nat*
  initially s_a := *wait;* t_a := 0
  serviceArequest: (s_a = *wait*) → s_a := *serving*
  serviceA: (s_a = *serving*) → s_a := *served* $_{pA}\oplus$ $t_a := t_a + 1$
  serviceAreceive: (s_a = *served*) → s_a := *wait*

Now the entire system can be expressed as

*Client-Server*2 == *Client* || *Broker* || *ServerA*1 || *ServerB*1

What we would like to do is to be able to answer questions such as: What is the probability that the client will receive an answer within the first tick? The semantics of our system would guarantee an answer within the first tick with probability at least *pA* **min** *pB*. To understand why this is, note that we interpret standard nondeterminsm as follows: when two or more actions are enabled any of them can be chosen for execution. In other words, we can only guarantee that the system will do at least as well as the worst server.

Let us now briefly illustrate how the broker specification may be refined. Even without knowing how *pA* and *pB* are related we notice that we can do better than in the original specification by replacing the nondeterministic choice over which service to request by a (fair) coin flip. This would indeed improve the least probability of getting an answer within the first tick to (1/2)*pA* + (1/2)*pB*, since as expected, probabilistic choices are interpreted as "averaging" the expected results according to the specified probabilities. (In fact, a nondeterministic choice *S*1 [] *S*2 is always refined by a probabilistic choice *S*1 $_p\oplus$ *S*2 since a valid implementation of arbitrary choice is one in which a (biased) coin is used to decide which branch to execute.) Crucially, such notions of refinement allow us to reuse analyses: if property *P* holds with probability *p* in a specification *S*, then *P* will hold with probability *at least p* for any refinement *S'* of *S*.

However, we may also opt to postpone the decision of how to refine server selection until we have more information about *pA* and *pB*. In fact, it is clear that weighing the choice towards the service that has a higher probability of returning a quick answer is better than flipping a fair coin. This is an example where we can make specifications more precise as we learn more about descriptions.

The framework that we have described enables reasoning about the mentioned refinements. Moreover, refinements of the state space are also possible (enabling more abstract data types to be refined into more concrete ones). However, in its current form the formalism is not tailored to expressing certain architectural notions and concerns. For example, one cannot differentiate between specifications of ports and roles. A related concern is how to express the conditions for port-role compatibility. More generally, although the framework is suitable for expressing hierarchies, further investigation is needed to determine how to best expose behavior at the interface level.

3. DISCUSSION

As we have argued above, considerable leverage can be obtained by incorporating uncertainty into architectural descriptions. The easiest way to do this is augment architectural representations so that one can specify properties in terms of probability distributions, thereby permitting one to describe a range of behaviors, while still permitting analysis of deviation from the expected patterns, and supporting Monte Carlo style simulation. More complex, but also relatively straightforward, is to adopt behavioral descriptions that permit one to include uncertainty – in our case we adopt the framework of [4], which supports probabilistic choice, and supports monotonic reasoning in the presence of refinement.

We view this work as a starting point, however. Additional aspects of uncertainty that we would like to characterize formally include dynamic evolution of the architectural structure under uncertainty, general specification of resource consumption in the presence of uncertainty, and a closer linkage between simple stochastic property specifications and probabilistic behavioral descriptions. We are also exploring the possibility of including properties that represent dynamically-updated predictions of future states of the system, such as anticipated performance [8]. Such predictions necessarily require one to characterize and reason about uncertainty.

4. ACKNOWLEDGMENTS

This research was supported by DARPA under grants N66001-99-2-8918 and F30602-00-2-0616, by the US Army Research Office (ARO) under grant numbers DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to Carnegie Mellon University's CyLab and DAAD19-01-1-0485, and the NASA High Dependability Computing Program under cooperative agreement NCC-2-1298. The views and conclusions described here are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, the ARO, NASA, the US government, or any other entity.

5. REFERENCES

[1] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl and P. Steenkiste. Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure. In *IEEE Computer*, Vol. 37(10), October 2004.

[2] P. Feiler, D. Gluch, J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Software Engineering Institute, Carnegie Mellon University Technical Report (CMU/SEI-2006-TN-011), Pittsburgh, PA, 2006.

[3] D. Garlan, R.T. Monroe and D. Wile. Acme: Architectural Description of Component-Based Systems. In Gary T. Leavens and Murali Sitaraman editors, *Foundations of Component-Based Systems*, Pages 47-68, Cambridge University Press, 2000.

[4] A. McIver. Quantitative Refinement *and* Model Checking for the Analysis of Probabilistic Systems. In *Formal Methods 2006*, Springer, LNCS 4085, pages 131-146, August 2006.

[5] A. McIver and C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems.* Springer, 2005.

[6] N. Medvidovic and R.N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Software Eng.*, vol. 26, no. 1, pp. 70-93, Jan. 2000.

[7] Object Management Group. *UML 2.0 Superstructure Specification: Final Adopted Specification.* http://www.omg.org/docs/ptc/03-08-02.pdf, August, 2003.

[8] V. Poladian, D. Garlan, M. Shaw, B. Schmerl, J.P. Sousa and M. Satyanarayanan. Leveraging Resource Prediction for Anticipatory Dynamic Configuration. In *Proceedings of the First IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, SASO-2007, Pages 214-223, 8-11 July 2007.

[9] B. Schmerl and D. Garlan. AcmeStudio: Supporting Style-Centered Architecture Development (Research Demonstration). In P*roceedings of the 26th International Conference on Software Engineering*, Edinburgh, Scotland, 23-28 May 2004.

[10] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman and H. Yan. Discovering Architectures from Running Systems. In *IEEE Transactions on Software Engineering*, Vol. 32(7), July 2006.

[11] B. Spitznagel and D. Garlan. Architecture-Based Performance Analysis. In *Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering (SEKE'98)*, June 1998.