A Quantitative Approach to Analyzing Architectures in the Presence of Uncertainty

Harrison Strowd David Garlan

July 2009
CMU-CS-09-120

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The concepts and techniques that underlie architectural analysis are well understood in the research community. However, many of these approaches do not incorporate uncertainty into their techniques. This forces the architect to specify precisely properties that are not well defined or to analyze the system qualitatively, providing an imprecise representation of the properties the system will exhibit. This paper presents a technique for analyzing architectural models that exhibit probabilistic behavior and discusses the systemic properties that can be identified through this form of analysis.

This work is supported in part by the Office of Naval Research (ONR), United States Navy, N000140811223 as part of the HSCB project under OSD, by the US Army Research Office (ARO) to CyLab under grant numbers DAAD19-02-1-0389 and DAAD19-01-1-0485, and by the National Science Foundation under Grants No. CNS-084701 and IIS-0534656. 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 the ONR, the ARO, the NSF, the US government, or any other entity.

1 Introduction

Over the past two decades a variety of techniques have been developed to specify and analyze the behavior of a system at an architectural level of abstraction. Such analyses are 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.

One of the important challenges for architecture analysis is appropriately handling uncertainty. Uncertainty can arise in a variety of ways. Parts of the system or its environment may be outside the control of the system designer. The exact behavior or properties of architectural elements may not be known at the time of architectural design since they depend on detailed design decisions that will occur later. Or, to manage complexity the architect may use a high level of modeling within which the precise behavior must be abstracted.

Today there are typically two general-purpose alternatives for dealing with uncertainty in architectural models. One, you can pick a representative model and behavior and hope that later elaborations do not violate those assumptions. For example, you might posit a certain specific reliability of a server and perform analysis based on that assumption. Two, you can use nondeterminism to retain flexibility (and abstraction). By leaving the behavior unconstrained you allow for later refinement. For example, your model might specify that a server can fail nondeterministically.

Unfortunately, neither of these alternatives is adequate. For the first approach, invalid assumptions can render further analysis useless or require expensive re-analysis and design at a later date. For the second approach, analyses must necessarily be conservative (since behavior is largely unconstrained), requiring the architect to address problems that may not actually occur or that will occur so infrequently that they need not occupy center stage.

In this paper we explore a third alternative: make uncertainty explicit and quantitative. That is, we associate specific numerical measures of uncertainty with behavior. This is done through the use of an existing probabilistic behavior specification language, coupled with tools to check properties of such specifications. In particular, we address three questions: How can we incorporate probabilistic behavior into architectural specifications? What kind of architecturally-relevant properties can be checked over these specifications? How can tools be leveraged to provide automated assistance with this analysis?

In the remainder of this paper we summarize our initial results in answering these questions and illustrate the ideas in terms of a small case study. We begin by addressing the shortcomings of current architectural analysis techniques, highlight the key characteristics of an appropriate analysis technique in light of the three questions above, and discuss our approach and the broader conclusions to be drawn from this research.

2 Motivation

To illustrate the issues associated with modeling uncertainty at an architectural level of abstraction, consider a simplified version of a web search engine server cluster, consisting of a load balancing machine and a number of web sever machines (The basis for this model is the Google server cluster architecture as described in [BDH03].). The architecture of the system is illustrated in Figure 1. Following a client-server architectural style, the load balancer receives search requests from the users and assigns them to the web servers to be processed. However, in this system web servers are known to be unreliable and can fail unexpectedly.

The example architecture includes two types of components and two types of connectors.[1] The load balancing component receives a search request via the external connector to the user. It then forwards this request to a web server component via the internal connector to the appropriate device. For this example, as an architect we would like to be able to reason about the overall reliability of the system in the face of individual web server failures, the latency of requests in the system, and the average queue length for each web server.

Using existing architectural analysis techniques, we can employ one of the two traditional tactics discussed above. We may attempt to specify the explicit conditions under which a web server will fail, or even simply the failure rate of those servers. Unfortunately, the conditions that cause a web server to fail are based on a diverse set of low-level or environmental events. At the time of developing the architecture of the system, we would likely not know the details that influence these conditions, nor be able to predict their occurrence. This forces us to specify a possibly unrealistic set of conditions or values related to the failure of a web server, rendering our analysis less than useful.

On the other hand, we may choose to specify only the most abstract conditions that result in a web server failure and allow the model to fail nondeterministically. This forces us to neglect information such as the fact that we may know a priori that web server failures are relatively rare, even if we don't know their exact failure rate. Under these assumption of unbounded nondeterminism, we would also have to consider the possibility that all web servers may fail "simultaneously" as a reasonable outcome, even though the likelihood of this happening is extremely rare. In turn, this might lead the architect to design in special mechanisms to account for this possibility – likely with high development cost implications.

[1]For the purposes of this paper we focus on the dynamic architectural perspective, in which an architecture is represented as a set of components and the connectors among those components [SG96]. This perspective is primarily used to reason about the runtime qualities of the system, such as performance, reliability, etc. [CBB$^+$03].

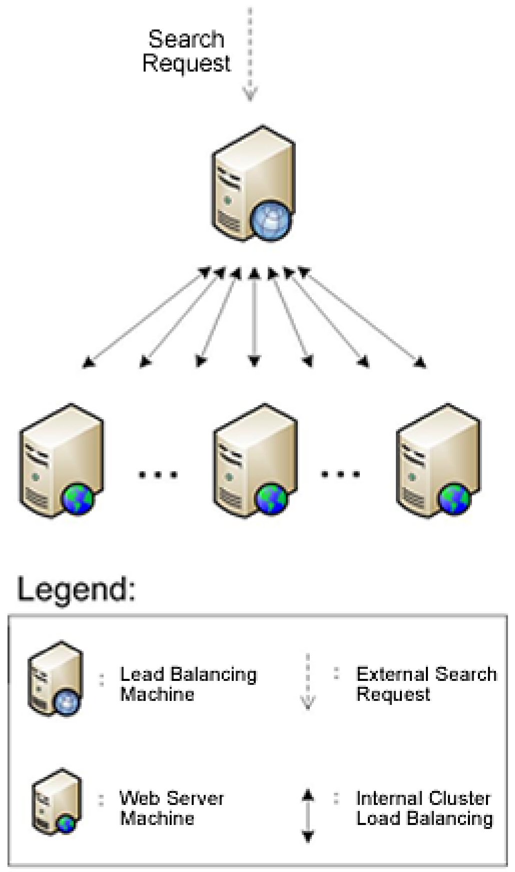


Figure 1: Architecture of the simplified web server example.

3 Requirements

Given the limitations of current approaches, outlined above, we argue that a better solution should include at least the following elements: First, the approach must support modeling at an architectural level of abstraction in order to understand the overall system behavior in terms of the key architectural design decisions (such as how to replicate servers, what protocols to use to detect failure, etc.). Second, to analyze the qualities of the system, we must be able to specify and verify architectural properties that explicitly capture the uncertainty inherent in the model. Finally, the analysis task is often too complex to be performed manually and hence requires appropriate tool support.

3.1 Architecture Modeling

To facilitate this form of modeling, the modeling language must allow the architect to capture the architectural elements of the system, to define its abstract behavior in terms of architecturally-significant events, to assign probabilities to those events, and to associate cost/reward values with the states and events of the model.

The key architectural elements to be modeled are the components and connectors of the system. Such elements can be modeled in a variety of ways depending on the level of detail and specificity required. In an architectural model many implementation the details are abstracted away in order to focus on the key characteristics to be analyzed about the system. The modeling language must also support the description of abstract behavior. This can be done using any number of behavior specification languages, including process algebras, state machines, relational algebras, etc.

Beyond raw behavior, the modeling language must support the explicit description of uncertainty. Specifically, by allowing for probabilities to be quantitatively associated with the conditions under which architecturally-relevant events occur in the system, the architect has far greater control over specifying the behavior of the model. In turn, the increased control leads to improved accuracy of the analysis.

Finally, the language must support the architect in defining extra-functional attributes of the behavior and structure, such as latencies, power consumption, reliability, etc. These can be done, for example, by associating cost/reward values with events and states of the model. In terms of the search engine example, such attributes include the number of active web servers in a given state, the latency of requests in the system, and the power consumption of each web server in the system.

3.2 Property Specification

The property specification language determines the kinds of emergent system behavior that can be analyzed. In considering languages for this purpose, it is important that they can (a) capture the steady-state behavior of the model, (b) allow for the analysis of some property in the context of a specific state or condition, (c) support analysis of the overall costs/rewards, and (d) support the specification and analysis of both boolean and probabilistic properties.

Steady-state behavior: For many of the dynamic properties being analyzed, there is often an important difference between the behavior of the system during start-up and the behavior of the system in its steady state. In the search engine example, the average number of requests assigned to a web server will be significantly less during the initial steps of the simulation than after it has been running for a while. For many systems it is often more important to understand the behavior of the system without considering the effects of the initialization period. To handle this, it is important that the property language employed be able to capture the steady-state properties of the system.

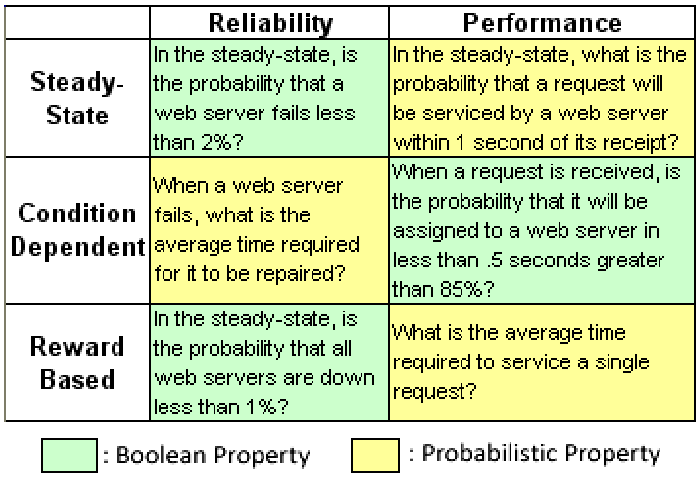
Context-specific behavior: On the other hand, it is often important to analyze the behavior of the system under certain specified conditions. In the web server example it may be important to know, "when a web server fails, what is the probability that it will be repaired before any other server fails?" Such properties attempt to hone in on particular circumstances under which a given property is significant.

Costs and rewards: When analyzing the dynamic properties of a system, as we are in the search engine example, it is necessary to be able to examine properties as they change over time. To facilitate this, it is necessary for the modeling language to allow the architect to assign a cost or reward value to each state or transition, but it is the property specification language that makes it possible for the architect to reason about these properties over the entire execution of the system. For the search engine example, the modeling language allows us to assign a cost to each state in which a web server is down, and the property specification language allows the architect to analyze the cumulative or average cost over an entire simulation or set of simulations. This form of analysis provides the architect insight into the global properties of the system that may not be directly reflected in its behavior.

Boolean versus probabilistic properties: In probabilistic modeling there are two types of properties that can be verified: boolean and probabilistic properties. Boolean properties are binary (true or false): they allow the architect to check whether a given property does or does not hold. For the search engine example, a relevant boolean property would be, "at any given time, the probability that all web servers are down is less than 1.00%." While boolean properties are useful in many scenarios, in other circumstances we may need to know the specific probability associated with a given outcome. In reasoning about the search engine, for example, it may be important to know the probability that all web servers will be down simultaneously.

Figure 2 lists some examples of the kinds of above properties that can be used to express the quality attributes of the search engine.[2] Both boolean and probabilistic properties are illustrated in these examples. For the reward-based properties, the rewards (i.e., extra-functional properties) defined in the model would be that all web servers are down and the amount of time that each request spends in the system. The reward-based properties for analysis allow the architect to understand how the value of these rewards change over the life of the system.

[2]Section 10.5 of the Appendix provides a more complete mapping of the quality attributes to the properties that can be defined by PRISM.



	Reliability	Performance
Steady-State	In the steady-state, is the probability that a web server fails less than 2%?	In the steady-state, what is the probability that a request will be serviced by a web server within 1 second of its receipt?
Condition Dependent	When a web server fails, what is the average time required for it to be repaired?	When a request is received, is the probability that it will be assigned to a web server in less than .5 seconds greater than 85%?
Reward Based	In the steady-state, is the probability that all web servers are down less than 1%?	What is the average time required to service a single request?

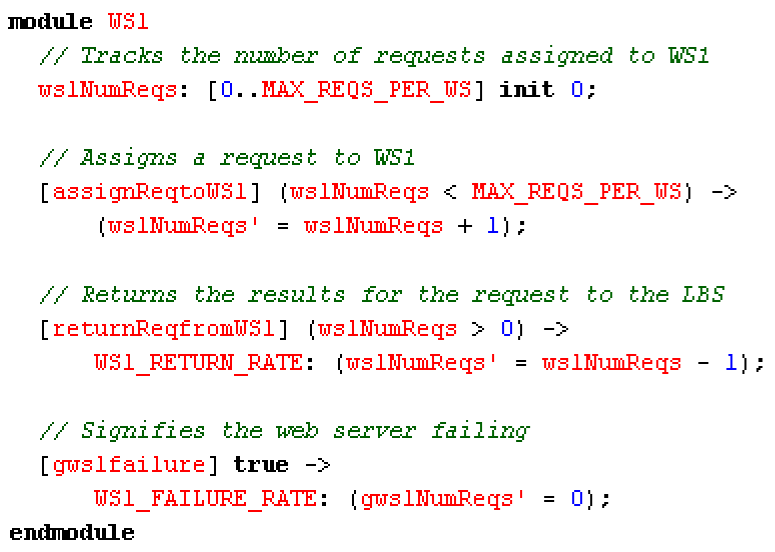
🟩 : Boolean Property 🟨 : Probabilistic Property

Figure 2: Examples for each type of property and the quality attribute that they relate to.

3.3 Automated Analysis

An appropriate tool to support this kind of analysis must allow the architect to evaluate the properties described above, scale sufficiently to handle realistic systems, and support the architect in analyzing the tradeoffs made in the architecture. Automated verification of property specifications is the minimal criteria for a tool to be applied to this form of analysis. The size of the model will vary widely, based on the level of detail included in the model and the size of the system being modeled. To make such analysis tractable, abstractions are usually required, but such abstractions should not alter the results or detract from the effectiveness of the analysis. The ability for the tool to model and analyze complex systems will dictate the effectiveness and accuracy of the results obtained.

In reasoning about a given architecture, it is important for the architect to be aware of the sensitivity points and tradeoffs being made. In turn, the tool should facilitate identifying these aspects of the architecture. In the search engine example, a key sensitivity point is the number of web servers used in the system. A system with too few web servers will not be able to service all the requests it receives, but a system with too many web servers will incur excessive costs. It is important for the architect to identify what the optimal number of web servers would be in light of the properties of the system. Ideally a tool would allow the architect to specify a range for the number of web servers in the system and to check properties for a representative sample of values in that range. The architect can then choose the value that optimizes business objectives.



```
module WS1
  // Tracks the number of requests assigned to WS1
  ws1NumReqs: [0..MAX_REQS_PER_WS] init 0;

  // Assigns a request to WS1
  [assignReqtoWS1] (ws1NumReqs < MAX_REQS_PER_WS) ->
      (ws1NumReqs' = ws1NumReqs + 1);

  // Returns the results for the request to the LBS
  [returnReqfromWS1] (ws1NumReqs > 0) ->
      WS1_RETURN_RATE: (ws1NumReqs' = ws1NumReqs - 1);

  // Signifies the web server failing
  [gws1failure] true ->
      WS1_FAILURE_RATE: (gws1NumReqs' = 0);
endmodule
```

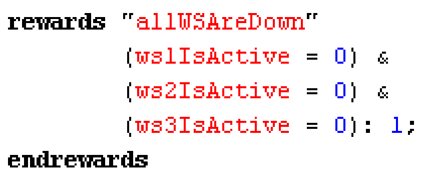
Figure 3: A simplified portion of the web server module.

4 Our Approach

Based on the criteria enumerated above, we identified the PRISM probabilistic model checker as a prime candidate for modeling architectures in the face of uncertainty. The language used to specify models in PRISM is similar to the Reactive Modules formalism as described by [AH99]. In this language, there are three primary types of probabilistic models: discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs), and Markov decision processes (MDPs). DTMCs behave according to the discrete probabilities of the enabled events in the system at discrete time steps. CTMCs assign rates to the events of the system and model how these events will occur over time. MDPs focus on modeling concurrency by combining discrete probabilities with nondeterministic behavior [KNP04]. All three types of model are based on the parallel composition of modules, and allow for the specification of finite-ranging variables and a set of events over those variables. As we will illustrate later, events are specified in terms of a name, pre-condition, post-condition, and probability [HKNP06].

To understand the runtime properties of the search engine example, we use a CTMC model. Figure 3 shows a portion of the web server module from this example.[3] This module keeps track of the number of requests that have been assigned to the web server, updating it as requests are assigned to or returned from a server. The three events of the module represent a request being assigned to a server, a request being returned from a server, and a server failing, respectively. Rates have been assigned to the later two events, but not to

[3]Section 10.1 of the Appendix shows the full version of the model.



```
rewards "allWSAreDown"
        (ws1IsActive = 0) &
        (ws2IsActive = 0) &
        (ws3IsActive = 0): 1;
endrewards
```

Figure 4: A sample reward from the search engine model, identifying the amount of time that all web servers are down simultaneously.

the first, in which case a default rate of 1 is assumed. The time units associated with the model are not explicitly defined by PRISM, allowing the architect to equate a single time-step with an appropriate amount of time in light of the system's context.

The load balancer module of the example is defined in a similar way. The interaction between these two modules is handled by synchronous events, such as the assignment and receipt of requests for a given web server. For these events to occur, the precondition provided in each module must be satisfied and the effective rate of the event is the product of the rates of the event in each module.

To capture the extra-functional qualities of the system we employ the reward structure mechanism available in PRISM. Figure 4 shows a sample reward from the search engine model.[4] This reward associates a value of 1 with every state of the model in which all three web servers are simultaneously not active (i.e., all have failed). In a CTMC model the reward value of a state is multiplied by the time spent in that state. Thus if the model spends three consecutive time-steps in a state in which all web servers have failed the cumulative reward during that period is 3, allowing the architect to analyze properties about the total or the average amount of time spent in various states of the model.

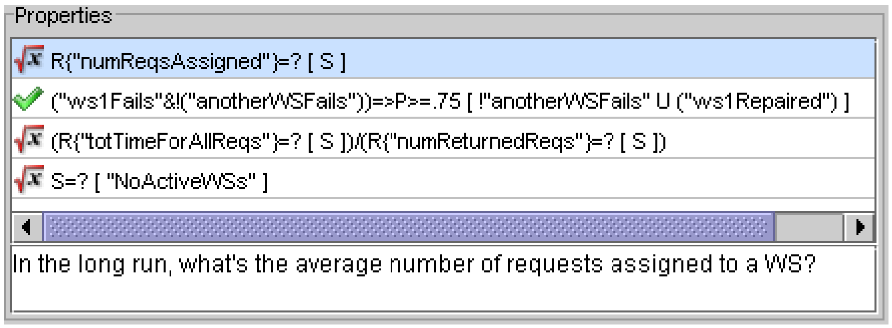
PRISM allows for temporal logic properties to be verified against the model. These properties can be specified using PCTL or CSL,[5] extensions of CTL that account for probabilistic behavior [KNP02]. Using CSL we were able to analyze the dynamic properties of the search engine example, including reliability and performance.

To illustrate, Figure 5 shows a representative set of the properties that can be verified against the search engine model just described.[6] The first of these properties analyzes the steady-state behavior of the model to determine the average number of requests that will be assigned to a web server. The second property verifies that when web server 1 fails, the probability that it will be repaired before another web server fails is at least 75%. The third property uses two rewards defined in the model: one tracks the cumulative time spent servicing requests, and the other tracks the total number of requests serviced,

[4]Section 10.2 of the Appendix shows the complete set of rewards defined for the model.

[5]PCTL properties are applied to DTMCs and MDPs, while CSL properties are applied to CTMCs.

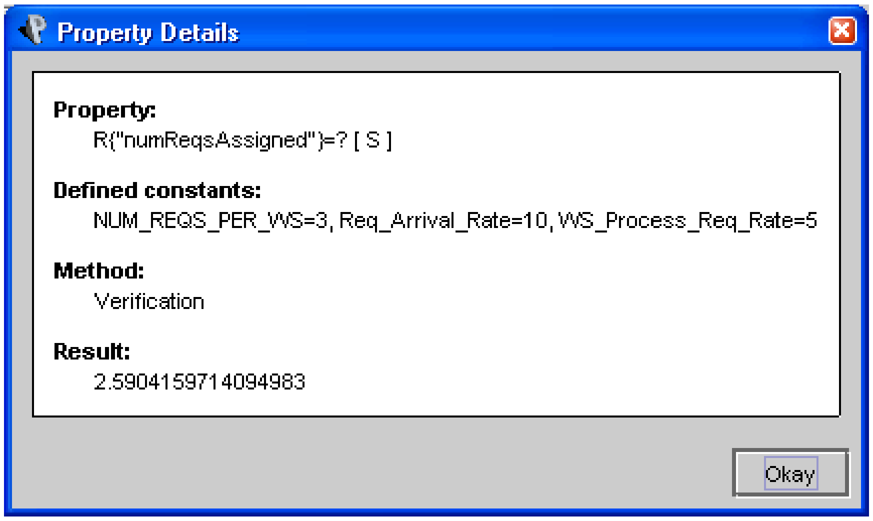
[6]Section 10.3 of the Appendix shows the full set of properties defined for the model.



```
R{"numReqsAssigned"}=? [ S ]
("ws1Fails"&!("anotherWSFails"))=>P>=.75 [ !"anotherWSFails" U ("ws1Repaired") ]
(R{"totTimeForAllReqs"}=? [ S ])/(R{"numReturnedReqs"}=? [ S ])
S=? [ "NoActiveWSs" ]
```

In the long run, what's the average number of requests assigned to a WS?

Figure 5: A sample of the properties used to analyze the web server model.



Property:
R{"numReqsAssigned"}=? [S]

Defined constants:
NUM_REQS_PER_WS=3, Req_Arrival_Rate=10, WS_Process_Req_Rate=5

Method:
Verification

Result:
2.5904159714094983

Figure 6: A sample of the output provided when verifying a property.

to determine the average latency of requests in the system. Finally, the last property analyzes the model to determine the percentage of time that all web servers in the system will be down simultaneously.[7]

For each of the properties discussed above PRISM provides feedback in the form shown in Figure 6. This figure shows the results of the analysis used to determine the average number of requests assigned to a web server in the system. The output identifies that the specified property was verified on the model with the maximum number of requests per web servers set to 3, the request arrival rate set to 10, and the rate at which web servers process a request set to 5. Using these parameters, the average number of requests assigned to a web server was found to be approximately 2.59.

[7]Section 10.4 of the Appendix provides a more complete range of the properties that can be specified.

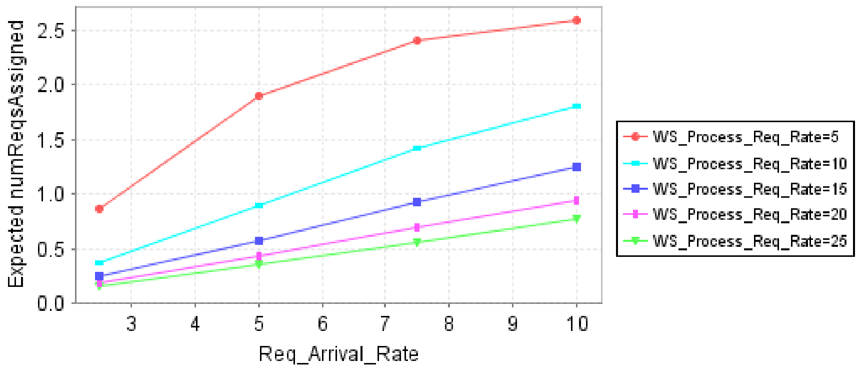


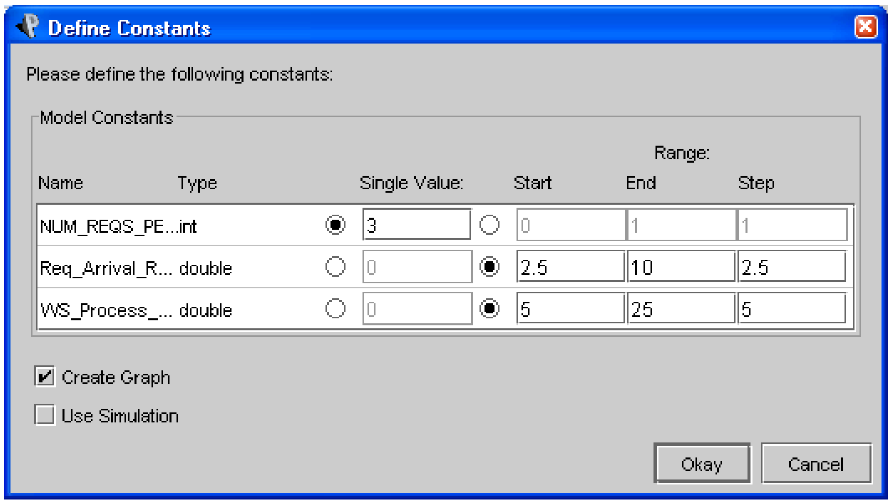
Figure 7: A sample of the output provided when running a PRISM experiment.

This indicates that for the majority of the time, each web server has 3 requests assigned to it, the maximum number possible. We can therefore conclude that the arrival rate is overwhelming the available web servers. This problem may be the result of a number of factors, including the fact that the load balancer does not have a sufficient number of web servers at its disposal, the web servers are not able to process requests fast enough, or the requests are simply arriving at too high a rate for the system to handle. Determining which applies, is the job of the architect who, armed with this kind of analysis, can make rational tradeoffs about ways to address such problems.

However, to gain a better understanding of the options available for decreasing the average number of requests assigned to each web server, the architect must understand the effect that each variable has on the other variables and the system as a whole. To gain this understanding, the architect will need to create a graph similar to Figure 7, detailing the effects of varying the web server processing rate and the request arrival rate on the average number of requests assigned to web servers in the system.

Assuming the number of available web servers and the number of requests that each server can handle is fixed, the architect can manipulate the two rates to achieve a desired property in the system. For instance, if the system must ensure that on average less than 2 requests are assigned to a web server, the architect will need to find a way to decrease the request arrival rate to at most 5 requests per time-step, or increase the rate at which web servers process requests to at least 10 requests per time-step.

PRISM facilitates the architect in performing this type of analysis by allowing the architect to run experiments. An experiment is a series of property verifications in which the variables of the model are incremented over a set range. PRISM runs every permutation of the values in the provided ranges and outputs the results in a chart, such as the one shown in Figure 7. PRISM allows the architect to specify the ranges for the variables of the experiment using the form shown in Figure 8.



Name	Type		Single Value:		Range: Start	End	Step
NUM_REQS_PE...	int	●	3	○	0	1	1
Req_Arrival_R...	double	○	0	●	2.5	10	2.5
WS_Process_...	double	○	0	●	5	25	5

[✓] Create Graph

[ ] Use Simulation

Figure 8: The form used by PRISM to allow the architect to specify the variables of an experiment.

5 Performance Considerations

One significant concern for this form of architectural analysis is how well it scales to meet the needs of realistic systems. For the purposes of this paper, we have used an extremely simplified web search engine cluster. The question that then needs to be answered is whether or not PRISM is able to model a realistic system. To understand the scalability concerns for PRISM, we used the four properties mentioned above to determine if the model checker is capable of verifying properties on a more complex model.

To gain this understanding, we varied the number of web servers in the model, the maximum number of requests assigned to a web server, and the arrival rate of requests in the system, and verified each property against the model, tracking the amount of time required for PRISM to verify the property. We chose to experiment on these three variables because each has a unique effect on model. Varying the number of web servers affects the number of modules contained in the model. Varying the maximum number of requests assigned to a web server affects the number of states required for the web server module. Varying the arrival rate of requests affects probabilities used to simulate and verify properties against the model. The results of this experiment are shown in Figures 9 and 10.

Figure 9 points out that PRISM is not capable of handling the scalability required to model a realistic web search engine cluster. The average amount of time required to verify properties against the model increases exponentially with both the number of web servers and the maximum number of requests per web server. An actual web search engine cluster would typically have hundreds of servers and each would service many requests simultaneously. This suggests a

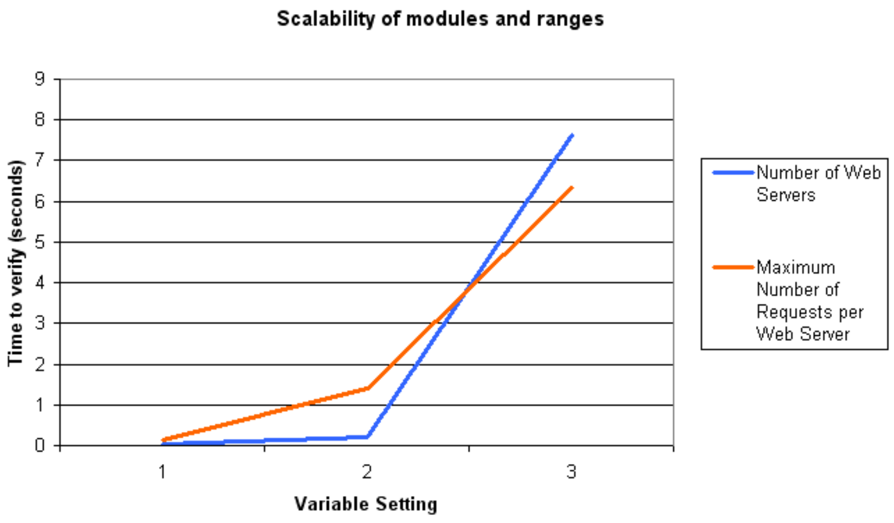


Figure 9: Scalability of PRISM in terms of the number of web servers and requests per web server.

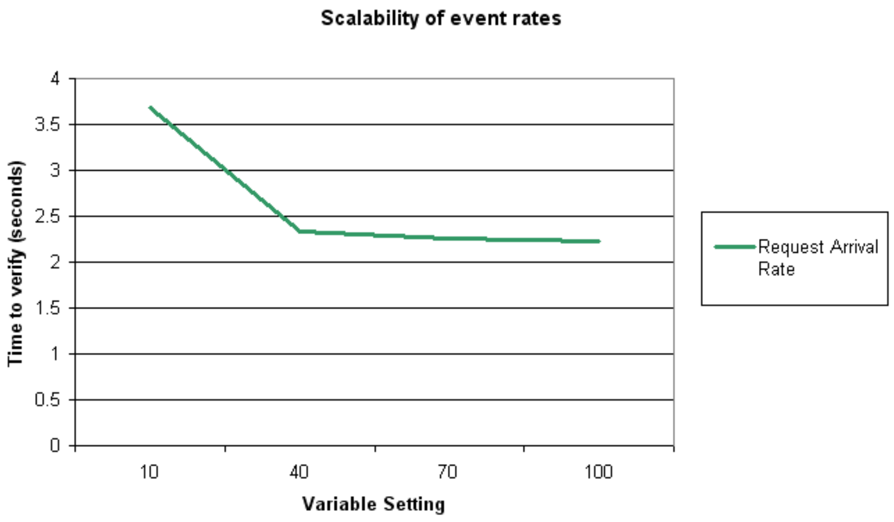


Figure 10: Scalability of PRISM in terms of the request arrival rate.

significant limitation of applying the PRISM model checker to the architectural analysis of complex systems with uncertainty.

On the other hand, we found an exponential decrease in the time required to verify properties as the arrival rate of requests increased, as shown in Figure 10. This is consistent with our expectations of the model, as cannot handle more than three web servers or three requests per web server. Thus, as the request arrival rate increases the queues will be filled more quickly and the system will reach a steady state in which all queues are full. In turn, the analysis will finish faster because the behavior of the model converges faster.

6 Discussion

As we have indicated above, the modeling language, property specification language, and tool support used in the architectural modeling of systems with uncertainty will have a large influence on the effectiveness of the analysis performed on the model. We feel that many of the features provided by the PRISM probabilistic model checker can be effectively used to capture and analyze such models. The ability to quantitatively specify the probabilities of the events in the system and to perform detailed analysis and experiments based on these probabilities provides the architect with better understanding of their affect on the underlying properties of the system.

While PRISM has proven to be an effective solution to some of the problems faced in attempting to analyze such models, it does have its own set of drawbacks and limitations. As noted above, scalability is a key issue. Even for our simplified example model, we were forced to keep our variables within small ranges to prevent state explosion. This is a serious barrier that will need to be handled in order for this solution to prove useful in industrial settings.

Another limitation imposed by PRISM is the inability to specify the distribution to be used during the simulation of the model. For CTMC models, PRISM uses an exponential distribution to simulate variability in the rates of events. In many cases, the realistic behavior of an event will not be captured by this distribution, inhibiting the accuracy of the model, and in turn the analysis. As shown in [CGS07], the ability to control the probabilistic distributions assigned to events in the system provides meaningful insight into the properties of the system.

The PRISM modeling language was extremely effective in handling variability in the rates and ranges defined for the model, but failed to support variability of the modules or events defined in the model. In our example, we would like to be able to vary the number of web servers with relative ease and to see the resulting effect on the properties of the system. Unfortunately, the constructs provided in the PRISM modeling language force us to hard code this information into the model. A preprocessor is available to allow the architect to specify such variables of the system and automatically generate the model. However, these variables are not accessible from the automated analysis provided by the PRISM model checker, preventing the architect from easily understanding the effects of varying these characteristics of the system.

In PRISM we found an effective means for specifying and analyzing architectural models with uncertainty. A set of limitations was identified in the approach, but it is our belief that these limitations do not outweigh the significant benefits obtained from modeling and reasoning about architectural models in this way.

7 Related Work

Three areas of existing work are closely related to the research presented in this paper: architectural modeling and analysis, probabilistic verification, and tools for analysis of system qualities.

7.1 Architectural Modeling and Analysis

Over the past decade architectural design has become an accepted component of most software development processes and formal models of software architecture allow for the analysis of system-level behavior. Unfortunately, research has not produced general techniques to model or reason about uncertainty. While there exist examples of architec-ture-based analyses for specific quality attributes in certain styles (e.g., queuing-theoretic analysis in message-queue systems), these are not generally applicable to the architectural design problem in general.

Some architectural modeling formalisms (such as [AG97a, MEK95]) do, however, support nondeterminism in behavioral specifications in order to achieve a high level of abstraction, and to permit later refinement of behavior. However, such specifications leave uncertainty completely unconstrained, and hence any analyses are unnecessarily conservative. Moreover, they are incapable of answering questions related to the expected outcome of the system-wide behavior, given knowledge about the expected behavior of the components.

7.2 Probabilistic Verification

The last two decades have seen significant development in formal specification languages, logics, and tools for reasoning about probabilistic systems. Specification languages based on probabilistic process algebras (e.g., PEPA [Hil96] and the stochastic π-calculus [Pri95]), probabilistic automata [Seg95], and probabilistic extensions of guarded commands [MM04] make it possible to explicitly associate probabilities with system transitions. Some specification languages also retain standard nondeterministic choice, making it possible to abstract over unknown probabilities. Probabilistic extensions of temporal logics (e.g., PCTL [HJ94], CSL [BKH99], and QPTL [MM04]) have enabled specification of rich properties for such system.

Probabilistic model-checkers (such as Prism [KNP02], MRMC [KKZ05], E-MC2 [HKMKS00], and PEPA Workbench [GH94]) have improved considerably both with respect to their power (as measured by the size of the models they can handle) and expressivity of logics they support. These tools have been successfully used to verify randomized distributed algorithms (e.g., Byzantine

Agreement, and randomized leader election protocols), communication and multimedia protocols (e.g., the Firework contention resolution protocol), security protocols (e.g., probabilistic contract signing), biological processes, etc. Recent work on theorem-proving-based verification of probabilistic systems includes formalization of expectation transformers [MM04] in HOL [HMM05, Cel06] and B [Hoa05].

7.3 Tools for Analysis of System Properties

A large number of special-purpose tools have been developed for analysis of system properties such as reliability, performance, and security. Some of these incorporate uncertainty into their analyses. Typically each of these tools requires its own kind of model. For example, the Mbius suite of tools supports a variety of analyses including reliability, availability, security, and performance based on a multi-model approach [DCC$^+$02]. However, such efforts do not specifically address analysis of software architectures (although many of them could potentially be adapted to architectural models). The advantage of centering analysis on architectural models is that it provides a single locus for design decisions and tradeoffs, as well as providing a high level of abstraction with which to understand complex systems.

8 Conclusion and Future Work

In this paper, we outlined an approach to architectural modeling in which uncertainty appears as a first-class specification mechanism. Such uncertainty goes beyond traditional nondeterminism insofar as it provides quantifiable measures that can be analyzed to determine a wide variety of stochastic properties of a system. The PRISM model checker automates many of the tasks required by this form of analysis and provides an effective means for reasoning about these systems and their behavior.

This work represents a starting point for further research into probabilistic-based architectural specification and analysis. However, to bring this line of research into practice, a number of missing ingredients will need to be investigated.

First, although one can map an architectural model into PRISM as a set of interacting components, where communication is determined by synchronized events (as we have illustrated above), it is not clear how best to model the richer vocabulary of software architecture found in modern architecture description languages [MT00], such as [GMW00, SAE], and standard modeling languages, such as UML [OMG]. In particular, how should one represent connectors, ports, hierarchical descriptions, and variability points? For example, in [AG97b] checking for compatibility between the ports of a component and the expectations of a connector to which it is attached is done via a kind of refinement check. What is the analog in a probabilistic setting?

Second, it is not clear how to specify models of dynamically changing architectures. In many modern systems architectures must adapt their structure based on environmental conditions, faults, or opportunities for run-time optimization. Such systems require a specialized form of analysis and reasoning, not directly supported by the PRISM model checker.

Finally, as noted earlier, finding good ways to manage the tractability of models for model checking is key to scaling the technology to realistic systems.

9 Acknowledgements

This work is supported in part by the Office of Naval Research (ONR), United States Navy, N000140811223 as part of the HSCB project under OSD, by the US Army Research Office (ARO) to CyLab under grant numbers DAAD19-02-1-0389 and DAAD19-01-1-0485, and by the National Science Foundation under Grants No. CNS-084701 and IIS-0534656. 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 the ONR, the ARO, the NSF, the US government, or any other entity.

References

[AG97a] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[AG97b] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[AH99] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.

[BDH03] Luis Andre Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, March-April 2003.

[BKH99] Christel Baier, Joost-Pieter Katoen, and Holger Hermanns. *Approximative symbolic model checking of continuous-time Markov chains.* CONCUR99 Concurrency Theory. Springer Berlin / Heidelberg, 1999.

[CBB$^+$03] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond.* Addison-Wesley, 2003.

[Cel06] Orieta Celiku. *Mechanized Reasoning for Dually-Nondeterministic and Probabilistic Programs.* PhD thesis, Turku Centre for Computer Science Technical Report 77, 2006.

[CGS07] Orieta Celiku, David Garlan, and Bradley Schmerl. Augmenting architectural modeling to cope with uncertainty. *International Workshop on Living with Uncertainty*, November 2007.

[DCC$^+$02] Daniel D. Deavours, Graham Clark, Tod Courtney, David Daly, Salem Derisavi, Jay M. Doyle, William H. Sanders, and Patrick G. Webster. The möbius framework and its implementation. *IEEE Trans. Softw. Eng.*, 28(10):956–969, 2002.

[GH94] Stephen Gilmore and Jane Hillston. The PEPA workbench: a tool to support a process algebra-based approach to performance modelling. In *Proceedings of the 7th international conference on Computer performance evaluation : modelling techniques and tools*, pages 353–368, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, page 47. Cambridge University Press, 2000.

[Hil96] Jane Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, 1996.

[HJ94] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6(5):512–535, September 1994.

[HKMKS00] Holger Hermanns, Joost-Pieter Katoen, Joachim Meyer-Kayser, and Markus Siegle. A markov chain model checker. In *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 347–362, London, UK, 2000. Springer-Verlag.

[HKNP06] Andrew Hinton, Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: A tool for automatic verification of probabilistic systems. *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, 3920:441–444, March 2006.

[HMM05] Joe Hurd, Annabelle McIver, and Carroll Morgan. Probabilistic guarded commands mechanized in HOL. *Theor. Comput. Sci.*, 346(1):96–112, 2005.

[Hoa05] Thai Son Hoang. *The Development of a Probabilistic B-Method and a Supporting Toolkit.* PhD thesis, The University of New South Wales, 2005.

[KKZ05] Joost-Pieter Katoen, Maneesh Khattri, and Ivan S. Zapreev. A markov reward model checker. In *QEST '05: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, page 243, Washington, DC, USA, 2005. IEEE Computer Society.

[KNP02] Marta Kwiatkowska, Gethin Norman, and David Parker. *PRISM: Probabilistic Symbolic Model Checker.* Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002.

[KNP04] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 2.0: A tool for probabilistic model checking. *Quantitative Evaluation of Systems*, pages 322–323, September 2004.

[MEK95] Jeff Magee, Susan Eisenbach, and Jeff Kramer. Modelling darwin in the π-calculus. *Theory and Practice in Distributed Systems*, 938:133–152, June 1995.

[MM04] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science).* SpringerVerlag, 2004.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[OMG] OMG. Unified modeling language. URL: http://www.uml.info/.

[Pri95] Corrado Priami. Stochastic π-calculus. *Computer Journal*, 38(7):578–589, 1995.

[SAE] SAE. SAE AADL information site. URL: http://www.aadl.info/.

[Seg95] Roberto Segala. *Modelling and Verification of Randomized Distributed RealTime Systems.* PhD thesis, Massachusetts Institute of Technology, 1995.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

10 Appendix

10.1 Full PRISM Model

```
// Web Sever Cluster
// Author: Harrison Strowd

// This model depicts a Web Server Cluster for web searches. It
//   consists of a Master Load Balancing Server (MLBS module),
//   and a set of Web Servers (WSx modules). Search queries are
//   recognized by the Master Load Balancing Server (MLBS),
//   which delegates them to an arbitrary available Web Server
//   (WS). The WS then processes the request and returns the
//   results to the MLBS. In actuality there is a lower level
//   structure that exists below the WS that enables it to
//   handle the query in a timely manner, but for the purposes
//   of this model, we have chosen to abstract that complexity
//   out of the model. This model allows WSs to fail at a
//   specified rate. This model also simulates the repair of WSs
//   that failed.

ctmc
// For the purposes of this model, 1 time unit will equate to 1
//   second


// -------------------------
//  Model Properties
// -------------------------

// The number of web servers at the disposal of the master load
//   balancing server
const int NUM_WS = 3;

// The maximum number of requests that can be queued in any stage
//   (accepted, being serviced, or service completed) of the load
//   balancing server.
const int NUM_MLB_REQ = 8;

// The maximum number of requests that can be queued in any stage
//   (assigned or service completed) of the web server.
const int NUM_REQS_PER_WS = 3;


// -------------------------
//  Action Rates
// -------------------------

// The rate at which requests are received by the load balancing
//   server.
//   An action with a rate of 50 occurs every .02 seconds
```

```
const double GetReq_RATE = 50;

// The rate at which requests will be returned to the user, once
//   they have been processed by a WS.
//   An action with a rate of 33 occurs every .03 seconds
const int ReturnReq_RATE = 33;

// The rates at which requests are assigned to or returned from
//   a WS.
//   An action with a rate of 25 occurs every .04 seconds
//   An action with a rate of 50 occurs every .02 seconds
const double assignReqtoWS1_RATE = 25;
const double returnReqfromWS1_RATE = 50;

const double assignReqtoWS2_RATE = 25;
const double returnReqfromWS2_RATE = 50;

const double assignReqtoWS3_RATE = 25;
const double returnReqfromWS3_RATE = 50;


// NOTE: The probability of failure and repair are far higher
//   than reality, but allow us to see these actions in
//   simulation and understand how they effect the properties
//   of the model.

// The probability that a WS will fail in any given time unit
//   An action with a rate of .01 occurs every 100 seconds
const double wsFailure_RATE = .01;

// The probability that a WS will be repaired at any given time
//   unit.
//   An action with a rate of .2 occurs every 5 seconds
const double wsRepair_RATE = .2;

// The rates at which the WSs can process a user's request
//   An action with a rate of 5 occurs every .2 seconds
const double WS1ProcessRequest_RATE = 5;
const double WS2ProcessRequest_RATE = 5;
const double WS3ProcessRequest_RATE = 5;



// ------------------------------------------
//  The Master Load Balancing Server
// ------------------------------------------
```

```
module MLBS

// -------------------------------------------
//  State keeping track of the WSs
// -------------------------------------------

// Tracks the activity of each of the web servers:
//   - a value of 0 indicates that the specified web server is
//     not available
//   - a value of 1 indicates that the specified web server is
//     available
ws1IsActive: [0..1] init 1;
ws2IsActive: [0..1] init 1;
ws3IsActive: [0..1] init 1;

// Tracks the number of request that have been assigned to each
//   WS. This information is required to ensure that when a WS
//   fails the requests that were assigned to it are reassigned
//   to an active WS.
ws1Reqs: [0..NUM_REQS_PER_WS] init 0;
ws2Reqs: [0..NUM_REQS_PER_WS] init 0;
ws3Reqs: [0..NUM_REQS_PER_WS] init 0;


// -------------------------------------------
//  State keeping track of the users' requests
// -------------------------------------------

// Tracks the number of requests that have been submitted to
//   the MLBS. The MLBS does not keep track of where a request
//   comes from or where it is returned to. In actuality it
//   would need to keep track of which users submitted which
//   queries, but this is a detail that I have abstracted away
//   from this model.
reqs: [0..NUM_MLB_REQ] init 0;

// Tracks the number of requests that have been assigned to web
//   servers.
servicing: [0..NUM_MLB_REQ] init 0;

// Tracks the number of requests that have been serviced and
//   are waiting to be returned to the user.
serviced: [0..NUM_MLB_REQ] init 0;
```

```
// --------------------------------
// Handling a request
// --------------------------------

// Gets a search request from the user
[getReq] (reqs < NUM_MLB_REQ) ->
  GetReq_RATE: (reqs' = reqs + 1);

  // Assigns the user's request to WS1, if available.
  [assignReqtoWS1]
    (servicing < NUM_MLB_REQ) &
    (reqs > servicing) &
    (ws1Reqs < NUM_REQS_PER_WS) &
    (ws1IsActive = 1) ->
    assignReqtoWS1_RATE :
      (servicing' = servicing + 1) &
      (ws1Reqs' = ws1Reqs + 1);

  // Gets a user's request back from WS1 after it has been
  //   serviced.
  [returnReqfromWS1]
    (serviced < NUM_MLB_REQ) &
    (servicing > serviced) &
    (ws1Reqs > 0) &
    (ws1IsActive = 1) ->
    (serviced' = serviced + 1) &
    (ws1Reqs' = ws1Reqs - 1);

  // Assigns the user's request to WS2, if available.
  [assignReqtoWS2]
    (servicing < NUM_MLB_REQ) &
    (reqs > servicing) &
    (ws2Reqs < NUM_REQS_PER_WS) &
    (ws2IsActive = 1) ->
    assignReqtoWS2_RATE :
      (servicing' = servicing + 1) &
      (ws2Reqs' = ws2Reqs + 1);

  // Gets a user's request back from WS2 after it has been
  //   serviced.
  [returnReqfromWS2]
    (serviced < NUM_MLB_REQ) &
    (servicing > serviced) &
    (ws2Reqs > 0) &
    (ws2IsActive = 1) ->
    (serviced' = serviced + 1) &
    (ws2Reqs' = ws2Reqs - 1);

  // Assigns the user's request to WS3, if available.
  [assignReqtoWS3]
    (servicing < NUM_MLB_REQ) &
    (reqs > servicing) &
    (ws3Reqs < NUM_REQS_PER_WS) &
    (ws3IsActive = 1) ->
    assignReqtoWS3_RATE :
      (servicing' = servicing + 1) &
      (ws3Reqs' = ws3Reqs + 1);

  // Gets a user's request back from WS3 after it has been
  //   serviced.
  [returnReqfromWS3]
    (serviced < NUM_MLB_REQ) &
    (servicing > serviced) &
    (ws3Reqs > 0) &
    (ws3IsActive = 1) ->
    (serviced' = serviced + 1) &
    (ws3Reqs' = ws3Reqs - 1);


// Returns the results from the search request to the user.
[returnReq]
  (reqs > 0) &
  (servicing > 0) &
  (serviced > 0) ->
  ReturnReq_RATE: (serviced' = serviced - 1) &
    (servicing' = servicing - 1) &
    (reqs' = reqs - 1);


// ---------------------------------
// Web Server Failures
// ---------------------------------

// WS1 failure
[ws1failure]
  (servicing >= ws1Reqs) &
  (ws1IsActive = 1) ->
  (reqs' = reqs) &
  (servicing' = servicing - ws1Reqs) &
  (ws1Reqs' = 0) &
  (ws1IsActive' = 0);
```

```
// WS2 failure
[ws2failure]
  (servicing >= ws2Reqs) &
  (ws2IsActive = 1) ->
  (reqs' = reqs) &
  (servicing' = servicing - ws2Reqs) &
  (ws2Reqs' = 0) &
  (ws2IsActive' = 0);


// WS3 failure
[ws3failure]
  (servicing >= ws3Reqs) &
  (ws3IsActive = 1) ->
  (reqs' = reqs) &
  (servicing' = servicing - ws3Reqs) &
  (ws3Reqs' = 0) &
  (ws3IsActive' = 0);


// --------------------------------
// Web Server Repairs
// --------------------------------

// GWS1 repair
[ws1repair]
  (ws1IsActive = 0) ->
  (ws1IsActive' = 1);

// GWS2 repair
[ws2repair]
  (ws2IsActive = 0) ->
  (ws2IsActive' = 1);

// GWS3 repaire
[ws3repair]
  (ws3IsActive = 0) ->
  (ws3IsActive' = 1);

endmodule



// -----------------------------------
//  A Web Server
// -----------------------------------

module WS1

// Tracks of the number of requests currently assigned to WS1
ws1NumReqs: [0..NUM_REQS_PER_WS] init 0;

// Tracks of the number of requests that WS1 has finished
//   servicing, but not yet returned to the MLBS
ws1ReqsServiced: [0..NUM_REQS_PER_WS] init 0;


// ------------------------------------
// Handling a Request
// ------------------------------------

// Assigns a request to WS1. This can only occur if WS1 does
//   not have the maximum number of requests assigned to it.
[assignReqtoWS1]
  (ws1NumReqs < NUM_REQS_PER_WS) ->
  (ws1NumReqs' = ws1NumReqs + 1);

// Signifies WS1 processing a user's request
[ws1ProcessRequest]
  (ws1NumReqs > 0) &
  (ws1ReqsServiced < ws1NumReqs) ->
  WS1ProcessRequest_RATE:
    (ws1ReqsServiced' = ws1ReqsServiced + 1);

// Returns the result for a user's query to the MLBS
[returnReqfromWS1]
  (ws1NumReqs > 0) &
  (ws1ReqsServiced > 0)->
  returnReqfromWS1_RATE:
    (ws1NumReqs' = ws1NumReqs - 1) &
    (ws1ReqsServiced' = ws1ReqsServiced - 1);


// --------------------------------
// Web Server Failure
// --------------------------------

// WS1 failure
[ws1failure]
  true ->
  wsFailure_RATE: (ws1NumReqs' = 0) &
```

```
    (ws1ReqsServiced' = 0);


// --------------------------------
// Web Server Repair
// --------------------------------

// WS1 repair
[ws1repair]
  true ->
  wsRepair_RATE : true;

endmodule


// Replicating WS1's module to include more WSs
module WS2 = WS1
[
  ws1NumReqs = ws2NumReqs,
  ws1ReqsServiced = ws2ReqsServiced,
  ws1ProcessRequest = ws2ProcessRequest,
  assignReqtoWS1 = assignReqtoWS2,
  returnReqfromWS1 = returnReqfromWS2,
  returnReqfromWS1_RATE = returnReqfromWS2_RATE,
  ws1failure = ws2failure,
  ws1repair = ws2repair
]
endmodule

module WS3 = WS1
[
  ws1NumReqs = ws3NumReqs,
  ws1ReqsServiced = ws3ReqsServiced,
  ws1ProcessRequest = ws3ProcessRequest,
  assignReqtoWS1 = assignReqtoWS3,
  returnReqfromWS1 = returnReqfromWS3,
  returnReqfromWS1_RATE = returnReqfromWS3_RATE,
  ws1failure = ws3failure,
  ws1repair = ws3repair
]
endmodule
```

10.2 PRISM Rewards

```
// Tracks the total time units elapsed
```

```
rewards "numTimeUnits"
  true : 1;
endrewards

// Identifies the number of WSs failures that occur in the system
rewards "numWSFailures"
  [ws1failure] true : 1;
  [ws2failure] true : 1;
  [ws3failure] true : 1;
endrewards

// Identifies the total amount of time that any WS is not active
rewards "wsDownTime"
  (ws1IsActive = 0): 1;
  (ws2IsActive = 0): 1;
  (ws3IsActive = 0): 1;
endrewards

// Identifies the amount of time all WSs are down simultaneously
rewards "allWSAreDown"
  (ws1IsActive = 0) &
  (ws2IsActive = 0) &
  (ws3IsActive = 0): 1;
endrewards

// Rewards used for Quality Attribute properties:

// Tracks the number of requests that are currently queued at the
//   MLBS
rewards "numReqsQueued"
  (reqs >= (ws1Reqs + ws2Reqs + ws3Reqs)) :
    (reqs - (ws1Reqs + ws2Reqs + ws3Reqs));
endrewards

// Tracks the number of requests that have been assigned to WSs
rewards "numReqsAssigned"
  true : (ws1Reqs + ws2Reqs + ws3Reqs);
endrewards

// Tracks the number of WSs that have a request assigned to them
rewards "numWSsWorking"
  (ws1Reqs > 0) : 1;
  (ws2Reqs > 0) : 1;
  (ws3Reqs > 0) : 1;
endrewards
```

```
// Tracks the total amount of time that all requests have spent
//   in the system
rewards "totTimeForAllReqs"
  true : reqs + servicing + serviced;
endrewards

// Tracks the total number of requests that have been successfully
//   serviced by the system
rewards "numReturnedReqs"
  [returnReq] true : 1;
endrewards
```

10.3 PRISM Properties

```
label "NoActiveWSs" =
    (ws1IsActive = 0) &
    (ws2IsActive = 0) &
    (ws3IsActive = 0);
label "UnevenLoad" =
    ((ws1Reqs > 1) & ((ws2Reqs = 0) | (ws3Reqs = 0))) |
    ((ws2Reqs > 1) & ((ws1Reqs = 0) | (ws3Reqs = 0))) |
    ((ws3Reqs > 1) & ((ws1Reqs = 0) | (ws2Reqs = 0)));

// In the long run, what is the probability that at least 80% of
//   the servers are up?
S=? [((ws1IsActive + ws2IsActive + ws3IsActive) / NUM_WS > .8)]

// In the long run, what is the probability that all WSs are down?
S=? [ "NoActiveWSs" ]

// In the long run, what is the average number of requests that
//   are waiting to be
// assigned to a WS?
R{"numReqsQueued"}=? [ S ]

// In the long run, what is the average number of requests that
//   are assigned to a WS?
R{"numReqsAssigned"}=? [ S ]

// In the long run, what is the average number of WSs that are
//   servicing a request? (This would vary from the number of
//   requests if each WS could service more than one request at a
//   time.)
R{"numWSsWorking"}=? [ S ]

// In the long run, what is the probability that the MLBS's
//   request queue is full?
S=? [ reqs = NUM_MLB_REQ ]

// In the long run, what is the probability that one server has
//   more than 1 request while another server has 0 requests?
S=? [ "UnevenLoad" ]

// What is the average time required to process a single request?
(R{"totTimeForAllReqs"}=? [ S ]) / (R{"numReturnedReqs"}=? [ S ])
```

10.4 PRISM Example Property Analogies

The following are the sample PRISM properties found on the PRISM website [8] and their analogs in terms of the Web Server example:

[8]The full explanation of the sample properties can be found at http://www.prismmodelchecker.org/manual/PropertySpecification/Introduction.

PRISM Example:	Google Web Server analog:
P>=1 [F terminate]	P>=1 [F “allWSAreDown”]
The algorithm eventually terminates successfully with probability 1	Eventually all WSs in the system will be down simultaneously with probability 1
“init” => P<0.1 [F<=100 num_errors > 5]	“ws1Down” => P>.85 [F<=20 ws1IsActive=1]
From an initial state, the probability that more than 5 errors occur within the first 100 time units is less than 0.1	From the state in which WS1 is down, the probability that it will be repaired within the next 20 time units is greater than .85
“down” => P>0.75 [!“fail” U[1,2] “up”]	“ws1HasRequest” => P>.7 [!“ws1Down” U[1,5] ws1ReqServiced=1]
When a shutdown occurs, the probability of system recovery being completed in between 1 and 2 hours without further failures occurring is greater than 0.75	When WS1 is assigned a query, the probability of it servicing that query in between 1 and 5 time units without failing is greater than .7
S<0.01 [num_sensors < min_sensors]	S<.45 [(ws1NumReqs + ws2NumReqs + ws3NumReqs)>5]
In the long-run, the probability that an inadequate number of sensors are operational is less than 0.01	In the long run, the probability that at least 5 requests have been assigned to the WSs is greater that .45
P=? [!proc2_terminate U proc1_terminate]	P=? [(gws1NumReqs=0) U ”ws1Down”]
The probability that process 1 terminates before process 2 does	What is the probability that WS1 fails without ever receiving a request
Pmax=? [F<=T messages_lost > 10]	Pmax=? [F<=T “ws1HasQuery”]
The maximum probability that more than 10 messages have been lost by time T	What is the maximum probability that WS1 has been assigned a query by time T (because there is no non-determinism in our model, this does not provide a different result from P)
S=? [queue_size / max_size > 0.75]	S=? [(ws1Req + ws2Req + ws3Req)>=2]
The long-run probability that the queue is more than 75% full	What is the long run probability that two or more of the WSs have requests assigned to them

10.5 Mapping of Quality Attributes to PRISM Properties

The following is a more complete mapping of the relevant quality attributes, in terms of the properties to be analyzed about the model, to the PRISM structure used to analyze them. For those that are support by PRISM, we have provided the appropriate structure for analyzing them.

Quality Attribute:	Property:	PRISM Analysis:
Availability	In the long run, what is the probability that a request will not be serviced?	Depends on the circumstances that result in a request no being serviced.
	In the long run, what is the probability that the MLBS's request queue is full?	S=? [reqs = NUM_MLB_REQ]
	How do changes in the rate of requests effect the probability of a request being lost?	Depends on the circumstances that result in a request no being serviced.
Reliability	In the long run, what is the probability that all WSs are down?	S=? ["NoActiveWSs"]
	In the long run, what is the probability that at least 80% of the servers are up?	S=? [("NumActiveWSs" / NUM_WS > .8)]
	What is the highest rate of WS failure that will not result in more than 1% of the requests being lost?	Depends on what is means for a request to be lost.
	What is the highest rate of WS failure that will not result in all WSs being down more than 1% of the time?	Experiment: Varying wsFailure_RATE and checking [S=? ["NoActiveWSs"]]
Performance	What is the average time required to process a single request?	(R"totTimeForAllReqs"=? [S]) /(R"numReturnedReqs"=? [S])
	In the long run, what is the average number of requests that are waiting to be assigned to a WS?	R"numReqsQueued"=? [S]
	In the long run, what is the average number of requests that are assigned to a WS?	R"numReqsAssigned"=? [S]
	In the long run, what is the average number of WSs that are servicing a request? (This would vary from the number of requests if each WS could service more than one request at a time.)	R"numWSsWorking"=? [S]

Performance	In the long run, what is the probability that one server has more than 1 request while another server has 0 requests? (This is currently 0.0 because we restrict GWSs to a maximum of 1 requests at a time.)	S=? [“UnevenLoad”]
	How do changes in the rate of WS failures effect the latency of requests?	Experiment: Varying wsFailure_RATE and checking [(R“totTimeForAllReqs”=? [S]) /(R“numReturnedReqs”=? [S])]
	What effect does allowing servers with 0 requests to pull requests away from overloaded servers have on the latency of the system?	This would require an extension to the model
Cost	How many servers would you need to decrease the probabilty that all WSs are down to below 1%?	Experiment: Varying NUM_WS and checking [S=? [“NoActiveWSs”]]
	What is the average amount of power consumed by the system?	Outside of the scope of this model
	How do changes in the rate of requests effect the amount of power consumed by the system?	Outside of the scope of this model
Scalability	What is the maximum rate of requests the system can handle while keeping the average latency below 1 second?	Experiment: Varying GetReq_RATE and checking [(R“totTimeForAllReqs”=? [S]) /(R“numReturnedReqs”=? [S])
	What is the maximum rate of requests the system can handle without resulting in more than 1% of the requests being lost?	Depends on what is means for a request to be lost.