Computing Attributes of Software Architectures
A Static Method and Its Validation

Imen Derbel[1], Lamia Labed Jilani[1] and Ali Mili[2]
[1]*Institut Superieur de Gestion, Bardo, Tunisia*
[2]*New Jersey Institute of Technology, Newark, NJ, 07102-1982, U.S.A.*

Keywords: Software Architecture, Architecture Description Language, Analysis, Quality Attributes, Acme, Response Time, Throughput, Reliability.

Abstract: During the last two past decades, software architecture has been a rising subject of software engineering. Since, researchers and practitioners have recognized that analyzing the architecture of a software system is an important part of the software development process. Architectural evaluation not only reduces software development efforts and costs but it also enhances the quality of the software by verifying the addressability of quality requirements and identifying potential risks. To this aim, several approaches have been recently proposed to analyze system non-functional attributes from its software architecture specification.
In this paper, we propose an ADL based formal method for representing and reasoning about system non-functional attributes at the architectural level. We are especially interested in analyzing performance and reliability quality attributes. We also propose to analyze the sensitivity of the system by identifying components that have the greatest impact on the system quality. The automation of our model was followed by a series of experiments that allowed us to validate our inductive reasoning to prove the capabilities of our model to represent and analyze software architectures.

1 INTRODUCTION

Software Architecture is a rising subject of software engineering that helps people to oversee a system in high level. It is defined as the system structure(s), which comprise software elements, the externally visible properties of those elements, and the relationships between them (Bass et al., 1998).

The architecture of a software product is traditionally modeled from the requirements specification, according to the needs expressed by one or more clients. The specification expresses customer needs in terms of service functions, constraints, quality, etc. From the specifications, the designers propose one or more solutions that meet the customer needs. At this stage, we must ensure that these proposed alternatives meet all requirements specification to pass the later stages of the life cycle (eg. implementation, integration, etc.). It is therefore recommended to develop methods and tools to analyze non-functional properties of software architectures to provide designers with a support in their activities. Analysis of the product early in the software life cycle, helps not only to discover the problems of the architecture but also to make decisions refinements of software products upstream. This reduces software development efforts and costs, and enhances the quality of the software by verifying the addressability of quality requirements and identifying potential risks.

In this context, several approaches have been proposed in order to analyze systems quality attributes at the architectural level. However, despite capacities of architecture description languages (ADLs) in formal description of software architectures, there is a notable lack of support for non-functional attributes in existing ADLs. Acme, Aesop, Weaves and others allow the specification of arbitrary component properties, but none of them interprets such properties nor do they make direct use of them (Medvidovic and Taylor, 2000).

In this paper, we propose an ADL based formal method for representing and reasoning about system non-functional attributes at the architectural level. We are interested in analyzing performance attributes (response time and throughput) and reliability (failure probability). We aim to compile an architectural description and obtain a set of equations that characterize non-functional attributes of software architec-

Derbel I., Labed Jilani L. and Mili A..
Computing Attributes of Software Architectures - A Static Method and Its Validation.
DOI: 10.5220/0005347300550066
In *Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2015), pages 55-66
ISBN: 978-989-758-100-7
Copyright © 2015 SCITEPRESS (Science and Technology Publications, Lda.)

tures, using an inductive reasoning. These equations are then solved using Mathematica (©Wolfram Research) in order to obtain system properties as function of components and connectors properties. We also propose to analyze the system performance sensitivity and the system reliability sensitivity by identifying architectural components and connectors that limit the system quality and that need an urgent attention to be improved. In this paper, we report on our experimentation of our approach on Aegis system (Allen and Garlan, 1996). The goal here is to validate the inductive reasoning and the model we have proposed. First, we analyzed the non-functional properties of Aegis using our ADL based approach. Then, we simulate the performance of the Aegis system to determine and calculate its response time, its throughput and its reliability.

This paper is organized as follows. Section 2 presents the background and the related work. Section 3 introduces ACME+ ADL and presents the main syntactic features that we have added to Acme. Section 4, describes the compiler that we propose for analyzing ACME+ descriptions. Section 5, discusses performance and reliability sensitivity analysis. Section 6, presents the generation of an automated tool for the proposed analysis model. Section 7 outlines experiments that we have conducted to validate our inductive reasoning and analysis model. Section 8 concludes this paper.

2 RELATED WORK

Several methods have been proposed for evaluating software architectures quality attributes. These methods can be categorized as either being informal methods (including experience-based, simulation-based, scenario-based approaches) or being formal ones (including mathematical modeling based approaches) (Bosch, 1999).

Some approaches are interested to the qualitative analysis (Giannakopoulou et al., 1999), (wr2, 2005), they verify structural and behavioral properties of the software product such properties are: vivacity, liveness, coherence, blocking free, etc. While some other approaches focus on quantitative analysis, they calculate the values of software product measurable properties such as performance, reliability, availability, maintainability, etc. Our approach falls into formal analysis of quantitative quality attributes. In this context, a number of mathematical modeling-based software architecture evaluation methods have been developed. These methods model software architectures using well-known formalisms and models. Then, these models are used to estimate operational quality attributes. However, each approach analyzes only one attribute, either reliability or performance.

In the following, we first discuss different approaches for assessing performance of a software architecture. Then, we discuss the approaches for predicting reliability at the architectural level.

Several studies address the performance assessment from a software architecture description. Each approach is based on a certain type of performance model and specification language. The latter includes specification formalisms such as ADL descriptions, Chemical abstract machine, and UML based specification. Performance models include Stochastic Process Algebras, queueing networks (QN) and their extensions called Extended Queueing Networks (EQN) and Layered Queueing Networks (LQN), etc.

QN is one of the best known performance models. Aquilani et al., in (Aquilani et al., 2001), proposed the derivation of QN models from Labeled Transition Systems (LTS) describing the dynamic behavior of SAs. Spitznagel and Garlan, in (Spitznagel and Garlan, 1998), proposed the transformation of Acme descriptions to QN models using ”distributed message passing” style defined in Aesop ADL. However, they propose only performance analysis of client/server systems. Bernardo et al. in (Balsamo et al., 2002) proposed Æmilia, an architectural description language based on stochastic Process Algebra that allows to solve performance indices using Timed Markov Models.

Several studies address the reliability assessment from a software architecture description. Goseva-Popstojanova and Trivedi (Goseva-Popstojanova and Trivedi, 2001) classify these approaches into three categories: state-based, path-based and additive. We discuss the path-based and the state-based approaches while ignoring the additive approach as it is not directly related to software architecture.

Path-based approaches (Shooman, 1976), (Krishnamurthy and Mathur, 1997) assess the reliability of the system according to the possible execution paths of the program, which can be obtained experimentally, by testing or algorithmically. In other words, the reliability of each path is obtained as a product of the reliabilities of the components along that path. Then, the system reliability is calculated by averaging the reliability of all the paths. One of the major problems with the path-based approaches is that they provide only an approximate estimate of application reliability (Franco et al., 2012).

State-based approaches (Cheung, 1980), (Franco et al., 2012), (Gokhale, 2007) assume that the transitions between states have a Markov property, meaning that at any time the future behaviour of components or transitions between them is conditionally independent of the past behaviour. These models consider software architectures as a discrete Markov chain (DTMC) or a continuous time Markov chain (CTMC) or a semi-Markov process (SMP) which are solved using probabilistic model checking tools such as Prism (Kwiatkowska et al., 2009). However, Markov models face a common problem, the combinatorial growth of the statespace. This occurs when the model has a large number of states and a great number of transitions between those states exceeding the memory available.

3 ACME+: AN ARCHITECTURAL DESCRIPTION LANGAGE

In order to support the automated derivation of synthesized attributes from the attributes of building components and connectors, an ADL needs to have two important features: constructs to represent relevant attributes and constructs to represent *functional dependencies* between components and connectors. These two constructs are needed to reason about how attributes are synthesized throughout the architecture.

For the purposes of our study, we define an ADL called ACME+ as an extension of Acme ADL.

Acme was selected for extension because it is an interchange language offering benefits from the complementary capabilities of ADLs. Also, it is supported by AcmeStudio tool which enables users to edit architectures via a graphical user interface. In addition, it offers a complete ontology to describe software architectures, distinguishing between various architectural elements: components, connectors, and configuration. The construct of `functional dependency` arises from the observation that the topological information represented by Acme is not sufficient to derive synthesis rules for the various attributes, and consists primarily in defining relationships between the various ports of an Acme component and the various roles of an Acme connector. To fix our ideas, we focus on functional dependencies within a component, which represent the relationships between the ports of a component. At a minimum, the `functional dependency` must specify which ports are used for `input` and which ports are used for `output`. In addition, for input ports, we must specify whether the component may proceed with data from any one of the ports (`AllOf`) or the component needs data from all ports before it proceeds (`AnyOf`); also, there are cases where we may need a majority of input ports to proceed (`MostOf`), such as in a modular redundancy scheme (for example, we have three input ports providing duplicate information, and we proceed as soon as two out of the three produce the same input data).

As an example, let a component *C* has, say five ports, P1, P2, P3, P4, P5 and we wish to record that P1, P2, P3 are the input ports, then, depending on which configuration we want to represent, we write:

```
input(AllOf(P1,P2,P3)),
input(AnyOf(P1,P2,P3)),
input(MostOf(P1,P2,P3)).
```

In the latter two cases, we must also specify whether the input ports must deliver their inputs `synchronously` or `asynchronously`. Hence we could say, for example:

```
  input(AllOf(asynch(P1,P2,P3)),
  input(MostOf(synchro(P1,P2,P3)).
```

As for output ports, in case we have more than one for a given component, we may represent two aspects: the degree of overlap between the data on the various ports (`duplicate`, `exclusive`, `overlap`), and the synchronization between the output ports (`simultaneous`, `asavailable`). Pursuing the example discussed above, if P4 and P5 are output ports, then we can write, depending on the situation:

```
output(overlap(asavailable(P4,P5)),
output(exclusive(simultaneous(P4,P5)),
output(exclusive(asavailable(P4,P5)).
```

In ACME+, a functional dependency is written at the end of a component description, after the declaration of all the ports, or at the end of a connector description, after the declaration of all the roles. A declaration of a `functional dependency` has the following fields:

- a name, to identify the dependency,
- a declaration of the input relation (how input ports are coordinated),
- a declaration of the output relation (how output ports are coordinated),
- a declaration of relevant properties (e.g. processing time for components, transmission time for connectors, etc).

As an example, we may write:

```
FunDep {  Name_of_functional_dependency
input (MostOf(synchro(P1,P2,P3))),
output(exclusive(asavailable(P4,P5))),
properties(processingTime=0.02,
throughput = 45,failureProbability = 0.03)}
```

4 A COMPILER FOR ACME+ ARCHITECTURE

We have developed a compiler for ACME+; while programming language compilers map source code onto executable code, our compiler maps ACME+ source code onto a set of Mathematica equations that characterize the non functional attributes of the architecture of interest. To this effect, we assume that: all ports of components are labeled for `input` or for `output` (`input` ports feed data or control information to the component, and `output` ports receive data or control information from the component); all roles of connectors are labeled as `origin` or as `destination` (connectors carry data or control information from their `origin` roles to their `destination` roles); there is a single component without input port and with a single output port, called the `source`; there is a single component without output port and with a single input port, called the `sink`; we assume that the `source` and `sink` components are both dummy components, that are used solely for the purposes of our model (if the architecture happens to have a real component without input port and with a single output port, we provide it an input port and we attach a dummy source component and a dummy connector upstream of it; likewise for the sink component). We define an attribute grammar on such architectures, as follows:

- Each port of each component has an attribute for each property of interest (response time, throughput, failure probability); hence each port has three attributes, labeled RT (response time), TP (throughput), FP (failure probability).
- Likewise, each role of each connector has an attribute for each property of interest, labeled the same way.
- The output port of the source component has trivial values for all the attributes, namely:

```
source.inpPort.RT = 0,
source.inpPort.TP = infinity,
source.inpPort.FP = 0.
```

- The system inherits the attributes associated to the input port of the sink component, namely:

```
System.ResponseTime = sink.outPort.RT,
System.Throughput = sink.outPort.TP,
System.FailureProbability = sink.outPort.FP.
```

The question that we must address now is, of course, how do we compute the attributes of the sink from the properties of components and connectors. We do so by propagating attributes from the `source` to the `sink` in a stepwise manner, by considering the following information:

- The functional dependency of each component, as a relation between its input ports and output ports.
- The functional dependency of each connector, as a relation between its origin roles and its destination roles.
- The relevant properties of each component. For example, each component has a property called Processing Time, that may come in handy when we want to compute the value of the RT (response time) attribute of its output ports as a function of the value of the RT attribute of its input ports.
- The relevant properties of each connector. For example, each connector has a property called Transmission Time, that may come in handy when we want to compute the value of the RT (response time) attribute of its destination roles as a function of the value of the RT attribute of its origin roles.
- Whenever a port is attached to a role, the attribute values are passed forward from the output port to the source role. For each attachment of the form:

  `C.outPort to N.originRole.`

  We write:

```
C.outPort.RT = N.originRole.RT,
C.outPort.TP = N.originRole.TP,
C.outPort.FP = N.originRole.FP.
```

  What remains to explain now is how the attributes are propagated from input ports to output ports within a component and from origin roles to destination roles within a connector.

Let C designates a component, whose input ports are called $inPort_1;...;inPort_n$ and output ports are called $outPort_1;...;outPort_k$. We suppose that these input and output ports are related with a `functional dependency` relation R expressed as follows:

```
R(
Input(InSelection(InSynchronisation
              (inPort1; ..; inPortn)));
Output(OutSelection(OutSynchronisation
            (outPort1; ..; outPortk)));
Properties(procTime=0.7;thruPut=0.2;failProb=0.2)
 )
```

We review in turn the three attributes of interest.

4.1 Response Time

For each output port $outputP_i$ expressed in the relation R, we write:

$$C.outPort_i.RT = function(C.inPort1.RT; \\ ...;C.inPort_n.RT) + C.R.procTime. \quad (1)$$

where *function* depends on the construct `InSelection`, expressing the nature of the relation between input ports.

If **InSelection** is **AllOf**, then *function* is the maximum, we write:

$$C.outPort_i.RT = Max(C.inPort_1.RT;...;C.inPort_n.RT) + C.R.procTime. \quad (2)$$

If **InSelection** is **AnyOf**, then *function* is the minimum, we write:

$$C.outPort_i.RT = Min(C.inPort_1.RT;...;C.inPort_n.RT) + C.R.procTime. \quad (3)$$

If **InSelection** is **MostOf**, then *function* is the median, we write:

$$C.outPort_i.RT = Med(C.inPort_1.RT;...;C.inPort_n.RT) + C.R.procTime. \quad (4)$$

4.2 Throughput

For each output port $outPort_i$ of the component C expressed in the relation R, we write an equation relating the component's throughput and $inPort_i.TP$. This rule depends on whether all of inputs are needed, or any one of them. Consequently if **InSelection** is **AllOf**, and since the slowest channel will impose its throughput, keeping all others waiting, we write:

$$C.outPort_i.TP = Min(C.R.thruPut; (C.inPort_1.TP + ... + C.inPort_n.TP)). \quad (5)$$

Alternatively, if **InSelection** is **AnyOf**, since the fastest channel will impose its throughput, we write:

$$C.outPort_i.TP = Max(Min[C.R.thruPut; C.inPort_1.TP]; ...; Min[C.R.thruPut; C.inPort_n.TP]). \quad (6)$$

If **InSelection** is **MostOf**, then we write:

$$C.outPort_i.TP = Min(C.R.thruPut; (C.inPort_1.TP + ... + C.inPort_n.TP) \div n). \quad (7)$$

4.3 Failure Probability

For each output port $outPort_i$ of the component C expressed in the relation R, we write an equation relating component's failure probability and failure probabilities of its input ports. This rule depends on whether all of inputs are needed, or any one of them. We first consider that $inPort_i$, $i = 0..n$, provides complementary information (**InSelection** is **AllOf**). A computation initiated at $C.outPort_i$ will succeed if the component C succeeds, and all the computations initiated at the input ports of C succeed. Assuming statistical independence, the probability of these simultaneous events is the product of probabilities. Hence we write:

$$C.outPort_i.FP = 1 - (1 - C.inPort_1.FP \times ... \times C.inPort_n.FP)(1 - C.R.FailProb). \quad (8)$$

Second we consider that $inPort_i$ provide interchangeable information (**InSelection** is **AnyOf**). A computation initiated at $C.outputP_i$ will succeed if component C succeeds, and one of the computations initiated at input ports $C.inPort_i$ succeeds. Whence we write:

$$C.outPort_i.FP = 1 - (1 - C.inPort_1.FP) \times ... \times (1 - C.inPort_n.FP)(1 - C.R.FailProb). \quad (9)$$

If **InSelection** is **MostOf**,then we write:

$$C.outPort_i.FP = 1 - (1 - C.R.FailProb) \times (1 - (C.inPort_1.FP + ... + C.inPort_n.FP) \div n).$$

The ACME+ compiler generates all the equations that we have discussed in this section in Mathematica format, to enable us to reason about the non functional attributes of the overall system, as a function of the relevant properties of its components and connectors, its functional dependencies, and its topology.

5 SENSITIVITY ANALYSIS

Sensitivity analysis informs architects about what are the architectural constituents that need an urgent attention to be improved. So they can improve a software architecture, test and validate it based on its components and connectors properties, architectural style, etc. Therefore, we developed a performance and reliability sensitivity analysis able to help architects identify architectural components that require changes.

5.1 Performance Sensitivity Analysis

Performance sensitivity analysis consists in identifying component bottleneck that limits the system performance. Hence, we have used queueing networks laws. We present below the most important two laws (equations 10 and 11) that we have used. A more detailed explanation can be found in (Denning and Buzen, 1978). Let D_i be the total service demand on the constituent i (component or connector). D_i is defined by :

$$D_i = \frac{X_i}{X} \times S_i \quad (10)$$

where X_i and S_i are respectively the processing time and the throughput of constituent i. The system throughput X verifies the following inequality:

$$X \leq \frac{1}{D_i} \qquad (11)$$

Therefore, the constituent with largest D_i limits the system throughput and is the bottleneck. Since in our model, each constituent C is described by one or more `functional dependency` relations and each relation R is characterized by a processing time, we propose to calculate service demand $D_{C^{R_i}}$ of constituent C relative to each relation R_i. $D_{C^{R_i}}$ is defined by:

$$D_{C^{R_i}} = \frac{(C.R_i.thruPut \times C.R_i.procTime)}{System.Throughput} \qquad (12)$$

The constituent having the largest value of $D_{C^{R_i}}$, is the bottleneck of the system.

5.2 Reliability Sensitivity Analysis

Reliability sensitivity analysis consists in identifying component bottleneck that limits the system reliability. Let's recall that the equations generated by our compiler will be resolved by Mathematica numerically and symbolically. The symbolic resolution is to keep components and connectors properties unspecified and use Mathematica to produce a system property expression based on components and connectors properties. This form of resolution helps in analyzing the sensitivity with respect to reliability. Note that our model allows the analysis of system failure probability according to components and connectors failure probabilities. To determine which component/connector that most affects system reliability, we calculate the derivative of the system failure probability with respect to its components/connectors failure probabilities. The derivative of the formula is defined by the equation 13:

$$\frac{\partial System.FailProb}{\partial C_i.FailProb}; i = 1, 2, ..., n \qquad (13)$$

where C_i is the component/connector i. The component/connector having the highest value of the derivative is the reliability bottleneck.

6 AN AUTOMATED TOOL FOR ARCHITECTURE ANALYSIS

We have developed an automated tool that analyzes architectures according to the pattern discussed in this paper. This tool uses a compiler to map the architecture written in ACME+ onto Mathematica equations, then it invokes Mathematica to analyze and solve the resulting system of equations. The analysis process is illustrated by figure 1. It can be observed that our analysis tool takes as input a file containing a given system architecture description written in our enriched ACME+ ADL. The compiler then translates this file into mathematical equations that characterize the system's non-functional attributes. Then, the tool invokes Mathematica to compute actual values of the system's attributes or to highlight functional dependencies between the attributes of the system and the attributes of the system's components and connectors.

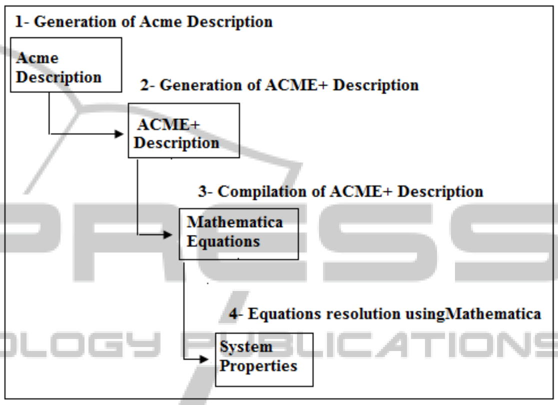

1- Generation of Acme Description
Acme Description
2- Generation of ACME+ Description
ACME+ Description
3- Compilation of ACME+ Description
Mathematica Equations
4- Equations resolution usingMathematica
System Properties

Figure 1: Analysis process workflow.

7 EXPERIMENTS

Our work consists in analyzing quantitative non-functional requirements that can inductively be derived such as response time, throughput, failure probability, etc. In order to validate our inductive approach, we propose to analyze the Aegis system (Allen and Garlan, 1996) using our ACME+ based method and verify the obtained results by simulating the system architecture.

7.1 ACME+ Analysis of Aegis System

Aegis Weapons System (Allen and Garlan, 1996) is designed to defend a battle group against air, surface and subsurface threats. These weapons are controlled through a large number of control consoles, which provide a wide variety of tactical decision aids to the crew. Figure 2 depicts the basic architecture of Aegis represented in AcmeStudio (Kompanek, 1998). The system consists of seven components: Geo_Server, Doctrine_Reasoning, Doctrine_Authoring, Track_Server, Doctrine_Validation, Display_Server and Experiment_Control. To this configuration, we add, for the sake of illustration, two dummy components Sink and Source and their associated connectors.

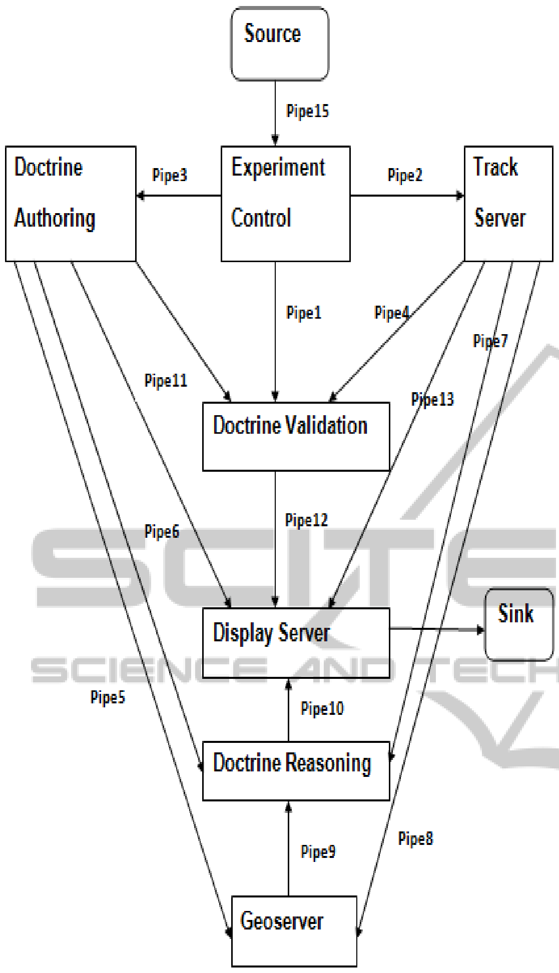


Figure 2: Aegis system architecture represented in ACME Studio.

- **ACME+ Description**

Using our proposed constructs of *functional dependency*, we give below examples of ACME+ descriptions. For the sake of brevity, we content ourselves with giving ACME+ descriptions of only two components of Aegis system. The overall architecture description of the Aegis Weapon System in ACME+ is available online at:
http://web.njit.edu/∼mili/AegisArch.txt.
We present ACME+ descriptions of the two components Display_Server and Doctrine_Authoring.

1. Display_Server operates in a single process (*R*) requiring all data from its input ports to display the results on its output port.

```
Component Display_Server {
Port inPort0; Port inPort1;
Port inPort2;Port inPort3;
FunDep = {R(
Input(AllOf(Synchronous(inPort0; inPort1;
                        inPort2; inPort3)));
Output(outPort);
Properties(procTime=1;thruPut=0.4;failProb=0.3)
)}};
```

2. The second example concerns the component Doctrine_Authoring which operates in a single task (*R*). Its output ports send simultaneously duplicate information.

```
Component Doctrine_Authoring {
Port inPort; Port outPort0;
Port outPort1; Port outPort2; Port outPort3;
FunDep= {R(
Input(inPort);
Output(Duplicate(Simultaneous(outPort1;
       outPort0;outPort2; outPort3)));
Properties(procTime= 0.7;thruPut=0.2;
        failProb=0.2))}};
```

- **Inductive Rules**

The compiler generates the following Mathematica equations for Display_Server:

Within Component Display_Server. Display_Server input ports are related with **AllOf** relation. Then, if we refer to equation 2, we can write:

$$\begin{aligned} DisplayServer.outPort.RT = Max(&\\ DisplayServer.inPort0.RT;&\\ DisplayServer.inPort1.RT;&\\ DisplayServer.inPort2.RT;&\\ DisplayServer.inPort.RT)+&\\ DisplayServer.R1.procTime&\end{aligned} \tag{14}$$

With reference to equation 5, we can write:

$$\begin{aligned} DisplayServer.outPort.TP = Min(&\\ DisplayServer.R.thruPut;&\\ \sum_{i=0}^{3}(C.inPort_i.TP))&\end{aligned} \tag{15}$$

With reference to equation 8, we can write:

$$\begin{aligned} DisplayServer.outPort.FP = 1-&\\ (1-DisplayServer.R.failProb)\times&\\ (1-\prod_{i=0}^{3} DisplayServer.inPort_i.FP)&\end{aligned} \tag{16}$$

Between Display_Server and Connectors. Whenever a component port is attached to a connector role, the attribute values are passed forward from the output port to the source role. Hence, we write:

$$DisplayServer.inPort0.RT = Pipe13.toRole.RT \tag{17}$$

$$DisplayServer.inPort1.RT = Pipe10.toRole.RT \tag{18}$$

$$DisplayServer.inPort2.RT = Pipe12.toRole.RT \tag{19}$$

$$DisplayServer.inPort3.RT = Pipe11.toRole.RT \tag{20}$$

$$DisplayServer.outPort.RT = Pipe14.fromRole.RT \tag{21}$$

The compiler generates the following Mathematica equations for Doctrine_Authoring :

Within Component Doctrine_Authoring. Doctrine_Authoring has only one input port, then the response time of each $outPort_i$, $i = 0..3$, is equal to the sum of its processing time and the response time of its input port. We write:

$$\begin{gathered} DoctrineAuthoring.outPort_i.RT = \\ (DoctrineAuthoring.R1.procTime+ \\ DoctrineAuthoring.inPort.RT); i = 0..3 \end{gathered} \tag{22}$$

The throughput of each $outPort_i$, $i = 0..3$, is equal to the minimum between its throughput and the throughput of its input port. We write:

$$\begin{gathered} DoctrineAuthoring.outPort_i.TP = \\ Min(DoctrineAuthoring.R.thruPut; \\ C.inPort.TP); i = 0..3 \end{gathered} \tag{23}$$

In order for a computation that is initiated at $DoctrineAuthoring.outPort_i$ $i = 0..3$, to succeed, component Doctrine_Authoring has to succeed, and the computation initiated at its input port has to succeed. Assuming statistical independence, the probability of these simultaneous events is the product of probabilities. Hence, we write:

$$\begin{gathered} DoctrineAuthoring.outPort_i.FP = \\ 1-(1-DoctrineAuthoring.R.failProb)\times \\ (1-DoctrineAuthoring.inPort.FP); i = 0..3 \end{gathered} \tag{24}$$

Between Doctrine_Authoring and Connectors. Whenever a component port is attached to a connector role, the attribute values are passed forward from the output port to the source role. Hence, we write:

$$DoctrineAuthoring.inPort.RT = Pipe0.toRole.RT \tag{25}$$

$$DoctrineAuthoring.outPort0.RT = Pipe5.fromRole.RT \tag{26}$$

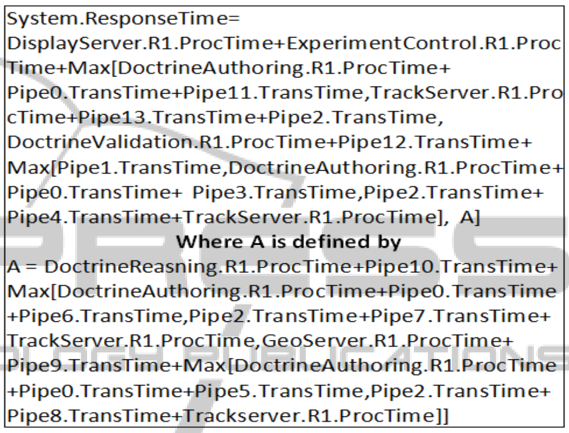
$$DoctrineAuthoring.outPort1.RT = Pipe3.fromRole.RT \tag{27}$$

$$DoctrineAuthoring.outPort2.RT = Pipe6.fromRole.RT \tag{28}$$

$$DoctrineAuthoring.outPort3.RT = Pipe11.fromRole.RT \tag{29}$$

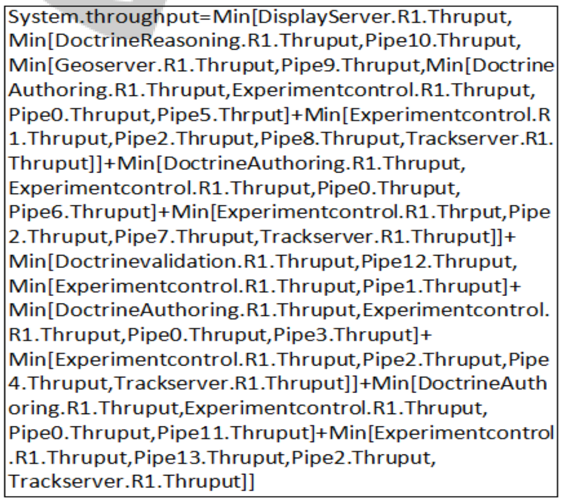
- **System Properties**

To determine the response time, the throughput and the failure probability of the system, we use Mathematica to solve the system of equations derived by the compiler taking as unknown *Datasink.input.RT*, *Datasink.input.TP* and *Datasink.input.FP* respectively. Figure 3 depicts the system response time, figure 4 depicts the system throughput and figure 5 depicts the system failure probability.



```
System.ResponseTime=
DisplayServer.R1.ProcTime+ExperimentControl.R1.Proc
Time+Max[DoctrineAuthoring.R1.ProcTime+
Pipe0.TransTime+Pipe11.TransTime,TrackServer.R1.Pro
cTime+Pipe13.TransTime+Pipe2.TransTime,
DoctrineValidation.R1.ProcTime+Pipe12.TransTime+
Max[Pipe1.TransTime,DoctrineAuthoring.R1.ProcTime+
Pipe0.TransTime+ Pipe3.TransTime,Pipe2.TransTime+
Pipe4.TransTime+TrackServer.R1.ProcTime], A]
                Where A is defined by
A = DoctrineReasning.R1.ProcTime+Pipe10.TransTime+
Max[DoctrineAuthoring.R1.ProcTime+Pipe0.TransTime
+Pipe6.TransTime,Pipe2.TransTime+Pipe7.TransTime+
TrackServer.R1.ProcTime,GeoServer.R1.ProcTime+
Pipe9.TransTime+Max[DoctrineAuthoring.R1.ProcTime
+Pipe0.TransTime+Pipe5.TransTime,Pipe2.TransTime+
Pipe8.TransTime+Trackserver.R1.ProcTime]]
```

Figure 3: Aegis response time as function of its components and connectors response time.



```
System.throughput=Min[DisplayServer.R1.Thruput,
Min[DoctrineReasoning.R1.Thruput,Pipe10.Thruput,
Min[Geoserver.R1.Thruput,Pipe9.Thruput,Min[Doctrine
Authoring.R1.Thruput,Experimentcontrol.R1.Thruput,
Pipe0.Thruput,Pipe5.Thrput]+Min[Experimentcontrol.R
1.Thruput,Pipe2.Thruput,Pipe8.Thruput,Trackserver.R1.
Thruput]]+Min[DoctrineAuthoring.R1.Thruput,
Experimentcontrol.R1.Thruput,Pipe0.Thruput,
Pipe6.Thruput]+Min[Experimentcontrol.R1.Thrput,Pipe
2.Thruput,Pipe7.Thruput,Trackserver.R1.Thruput]]+
Min[Doctrinevalidation.R1.Thruput,Pipe12.Thruput,
Min[Experimentcontrol.R1.Thruput,Pipe1.Thruput]+
Min[DoctrineAuthoring.R1.Thruput,Experimentcontrol.
R1.Thruput,Pipe0.Thruput,Pipe3.Thruput]+
Min[Experimentcontrol.R1.Thruput,Pipe2.Thruput,Pipe
4.Thruput,Trackserver.R1.Thruput]]+Min[DoctrineAuth
oring.R1.Thruput,Experimentcontrol.R1.Thruput,
Pipe0.Thruput,Pipe11.Thruput]+Min[Experimentcontrol
.R1.Thruput,Pipe13.Thruput,Pipe2.Thruput,
Trackserver.R1.Thruput]]
```

Figure 4: Aegis throughput as function of its components and connectors throughput.

- **Reliability Sensitivity Analysis**

We propose to analyze the sensitivity of the Aegis system relatively to the reliability. Thus, we propose to use Mathematica to calculate the derivative of the system failure probability with respect to the failure probability of each one its components / connectors. The derivative of the formula is defined by the equation 30, where C_i represents component i.

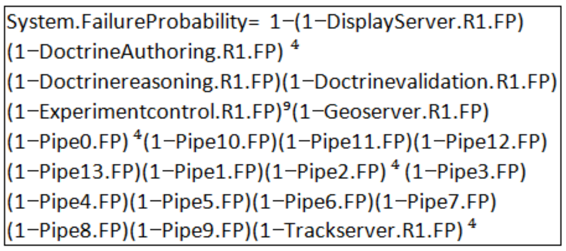

System.FailureProbability= 1−(1−DisplayServer.R1.FP)
(1−DoctrineAuthoring.R1.FP) ⁴
(1−Doctrinereasoning.R1.FP)(1−Doctrinevalidation.R1.FP)
(1−Experimentcontrol.R1.FP)⁹(1−Geoserver.R1.FP)
(1−Pipe0.FP) ⁴(1−Pipe10.FP)(1−Pipe11.FP)(1−Pipe12.FP)
(1−Pipe13.FP)(1−Pipe1.FP)(1−Pipe2.FP) ⁴ (1−Pipe3.FP)
(1−Pipe4.FP)(1−Pipe5.FP)(1−Pipe6.FP)(1−Pipe7.FP)
(1−Pipe8.FP)(1−Pipe9.FP)(1−Trackserver.R1.FP) ⁴

Figure 5: Aegis failure probability as function of its components and connectors failure probability.

In this example, we assume that all connectors have a low probability of failure equal to 0.001 and we propose to determine the component representing the reliability bottleneck. So, for each component of Aegis, we calculate the derivative defined in equation 30.

$$\frac{\partial System.FailProb}{\partial C_i.FailProb}; i = 1, 2, \ldots, n \qquad (30)$$

In table 1, the components are ordered by decreasing order of their derivatives values. The component having the largest value of the derivative is the one that most affects the reliability of the system, it is the bottleneck of the system. More specifically, the component *Experiment_control* is on the top of the list, showing that it has an impact on the overall system reliability.

Table 1: Reliability sensitivity analysis.

Composant C	C.FP	$\frac{\partial System.FailProb}{\partial C.FP}$
Experiment_Control(EC)	0.007	7.813
Doctrine_Authoring(DA)	0.004	3.462
Track_Server(TS)	0.006	3.469
Doctrine_Reasoning(DR)	0.008	0.869
Geoserver(GS)	0.009	0.869
Doctrine_Validation(DV)	0.005	0.866
Display_Server(DS)	0.003	0.864

To check on our approach of sensitivity analysis of Aegis system, we study the effect of variations of components reliabilities to identify points in the architecture where the variation has a higher impact on the property of the whole system. Thus, we vary the probability of failure of each component of the Aegis system (variation of 0.025) and calculate the failure probability of the system by keeping fixed the values of failure probability of other components. The same variations of failure probabilities values are performed for all components.

The graph in figure 6 depicts the failure probability of the overall system according to variations of 0.025 on components failure probabilities.

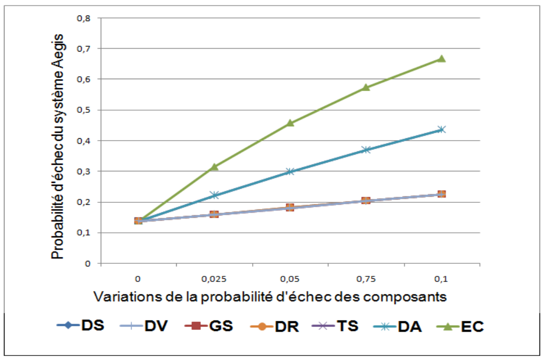


Figure 6: Reliability sensitivity analysis.

For the readability of the results, in the graphic's caption, from the left to the right, components are ordered from the lower to the higher increase of the impact on the overall system reliability. The graph clearly shows that the component *Experimentcontrol* has the highest impact, representing then, the bottleneck of the system. Hence, we note the compliance of conclusions drawn from the graph compared to those found by our approach.

7.2 Analysis of Aegis System by Simulation

In order to simulate Aegis system, we have first codified the architecture of the whole system using an $(M \times N)$ matrix which we denote by *arch*. M represents the number of system connectors and N is referred to the number of system components. This matrix describes the links between components and connectors. The intersection of a line i with a column j in *arch* matrix indicates whether the connector i is connected by its *source* role to the component j, is connected by its *destination* role to the component j or is not connected to this component.
Functional dependency relations (*AllOf*, *AnyOf*, *MostOf*), expressing links between input ports of the component, are also codified using *relation* matrix. Each column j of the *relation* matrix describes which connectors have to complete their transmissions in order to let component j proceed.
In order to compare the results found by our analysis tool ACME+ and those found by the simulation method, we proceed as follows:

1. we generate random values of components and connectors non-functional properties,
2. we use the formula found by our ACME+ analysis tool to calculate the property of the whole system. Let's recall that this formula expresses the property of the overall system based on the properties

of components and connectors,

3. we execute the simulation using the values of components and connectors properties which are generated in step 1,
4. we store the results found by ACME+ tool and by simulation method,
5. Finally, we compare the obtained results.

The whole process is applied to each one of the quality attributes: response time, throughput and reliability. It is repeated a hundred times in order to validate our inductive reasoning.

- **Response Time Simulation**

To calculate the system response time by the simulation method, we have used an array of structure, which we denote by $Exec$. It stores for each component and each connector respectively his ”*execution time*” and his ”*transmission time*”, its ”$state$” describing the state of the component or the connector: ”$waiting$”, ”$finished$” or in ”$execution$”, ”$time$” expressing the remaining time for the component or connector to terminate and go to the state ”$finished$”. Initially, for every element of $Exec$ (corresponding to a component or connector), fields ”*execution time*” and ”*transmission time*” are initialized to corresponding values randomly generated, the field ”$state$” is set to ”$waiting$”, the value of the field ”$time$” is set to the value of the execution time / transmission time of the component / connector.
In order to simulate the performance of the Aegis system, we start by assuming that the component $Datasource$ starts execution and all other components are pending. We initialize the value of a $clock$ to 0. We iterate over the array of structure $Exec$. In each iteration, the value of $clock$ is incremented by 1 and at the same time:

- the field ”$time$” of components and connectors in execution are decremented by 1,
- components and connectors that have finished, have their states changed to ”$finished$” ,
- check which components or connectors must be triggered and change their states to ”$execution$”. This requires consulting matrices $arch$ and $relation$ to verify the links between components and connectors.

The system response time is the value of the $clock$ when the states of all components and connectors are changed to ”$finished$”.

- **Throughput Simulation**

To calculate the throughput of the system by the method of simulation, we have used an array of structure $Exec$ that stores for each component / connector their $throughput$ values randomly generated, their ”$state$” describing the state of the component / connector: ”$waiting$”, ”$finished$” or in ”$execution$”, ”$time$” representing the time remaining to complete and pass the state ”$finished$”, ”$amount$”, the amount of information processed or communicated by the component or the connector. The amount of an architectural element depends on the amount of information received and on its $throughput$. In order to simulate the execution of the Aegis system, we start by assuming that the component $Datasource$ starts execution and all other components are pending. Initially the ”$amount$” of each component and connector is set to 0, the ”$time$” of each component is set to its execution time and that of each connector is initialized to its transmission time. Except the first component who starts its execution, its ”$amount$” is set to its $throughput$.
We iterate over the array of structure $Exec$. In each iteration:

- ”$time$” of components and connectors in execution are decremented by 1,
- components and connectors that have finished, have their states changed to ”$finished$”,
- check which components or connectors must be triggered and change their states to ”$execution$” and change their fields ”$amount$” to the amount of data that they can treat or emit. This requires consulting matrices $arch$ and $relation$ to verify the links between components and connectors. The ”$amount$” of an architectural element is equal to the minimum between the ”$amount$” received and its $throughput$.

The system throughput will be the value of ”$amount$” emitted by the last component of the architecture that was executed.

- **Failure Probability Simulation**

To determine the reliability of the system by the method of simulation, we used an array of structure that stores for each component / connector his ”$probabilityoffailure$”, its ”$state$” (”$waiting$”, ”$finished$”, ”$execution$”), the ”$time$” remaining to finish and move on to the state ”$finished$” and ”run” which indicates whether the component / connector succeeds or fails. We generate random values of components and connectors probabilities of failure. We also randomly determine if the architectural element fails or succeeds the current execution, according to its probability of failure. The overall system succeeds or fails based on the failure or the success of its components and connectors. We iterate over the array of structure $Exec$. In each iteration:

- the field ”*time*” of components and connectors in execution are decremented by 1,
- components and connectors that have finished, have their states changed to ”*finished*”,
- check which components or connectors must be triggered and change their states to ”*execution*”. This requires consulting matrices *arch* and *relation* to verify the links between components and connectors.
- verify which components and which connectors have succeeded or have failed and update their corresponding values of ”*run*”.

The success or the failure of the overall system is determined by the success or the failure of the last executed component. The steps already described are repeated several times (100 or 200 times) and each time, we determine whether the system succeeds or fails. We then calculate the probability of failure of the system as the quotient of the number of failures by the total number of tests.

7.3 Comparison of Analysis Results

We have analyzed Aegis system using two approaches: simulation and ACME+ tool. We have run many tests, in each test we generate random values of components and connectors properties. Then, we run our analysis tool ACME+ on these values in order to calculate properties of the whole system. We also run the simulation and finally, we compare the results found by the two methods.

To visualize the difference between the obtained results, we propose to represent the analysis results in bar charts graphics which plot Aegis properties computed by ACME+ tool and by simulation in each test. Graphics shown by figures 7, 8, 9 depicts an example of 20 tests done to compare respectively response time, throughput and failure probability values found by simulation approach and by our ACME+ tool.

Figures 7, 8 show that the analysis of the Aegis system by simulation and ACME+ tool generates equal values of response time and throughput properties. However, figure 9, relative to failure probability property, shows close results. Hence, these results claim the validity of our inductive reasoning.

8 CONCLUSION

In this paper, we propose an ADL based formal method for representing and reasoning about system non-functional attributes at the architectural level. We aim to automatically analyze performance and reliability of system architecture from performance and reliability of its components and connectors. For these reasons, we propose ACME+ as an extension of Acme ADL, and discuss the development and operation of a compiler that compiles architectures written in ACME+ ADL to generate equations that characterize non functional attributes of software architectures. We then conduct a sensitivity analysis on the results, to determine existent bottlenecks that most affect the performance and the reliability of the system. This will help architects to identify components and connectors that need urgent attention to be improved in order to ameliorate system quality.

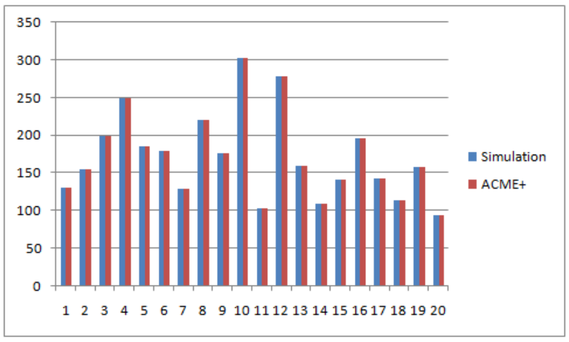


Figure 7: Aegis response time values found by simulation and ACME+.

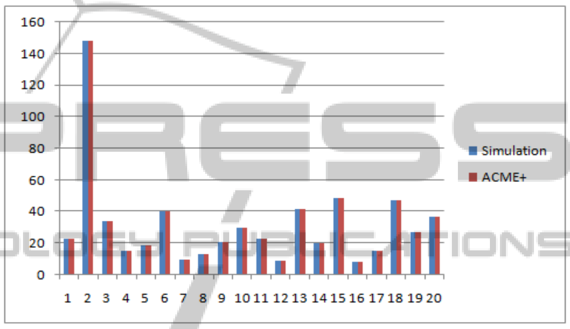


Figure 8: Aegis throughput values found by simulation and ACME+.

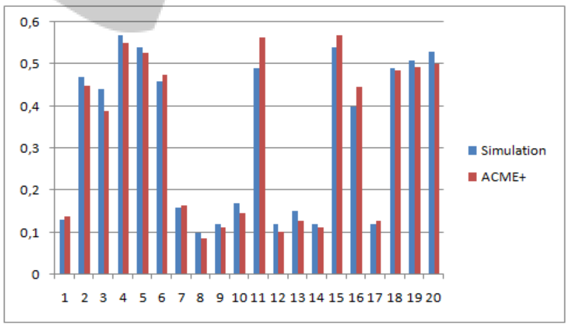


Figure 9: Aegis failure probability values found by simulation and ACME+.

Our work can be characterized by the following attributes, which set it apart from other work on architectural analysis.

- from a software architectural specification it is quite simple to derive an ACME+ textual description thanks to its expressiveness.
- we propose to estimate the non-functional properties directly from an architectural description to avoid problems occurred when using a mathematical models. In fact, the transformation of software architecture to a mathematical model, imposed by the analytical approaches, limits their capacities of analysis which depends on the used model.
- our analysis approach can be applied to any system that can be described by components and connectors for any architectural style (client/server, pipes and filters, etc.) unlike Acme-based approach, presented in (Spitznagel and Garlan, 1998), which is limited to client/server systems analysis.
- system analysis must deal with various quality attributes to enable a better understanding of the strengths and weaknesses of complex systems (Dobrica and Niemelae, 2002). Thus, unlike approaches found in the literature, which analyze either performance or reliability quality attributes, it is possible with the same ACME+ specification of a software system to analyze both performance and reliability. It is also possible to extend analysis to other quality attributes that can be inductively derived such as availability, maintainability, etc.
- our approach is supported by an automated tool. The user can perform many different experiments by analyzing the software architecture and modifying the components and connectors non-functional properties values or changing the software topology to get better performances before iterating the process again. It is then very easy to perform many ”what-if” experiments, changing parameters or structure of the model to see what the result is.

Among the extensions we envision for this work, we cite: the analysis of other quantitative non functional attributes and to investigate more profoundly architecture styles and dynamic architectures.

REFERENCES

(December 2005). Wr2fdr. *http://www.cs.cmu.edu/∼able/wright/wr2fdr_bin.tar.gz.*

Allen, R. and Garlan, D. (1996). A case study in architectural modeling: The aegis system. In *In Proceedings of the 8th International Workshop on Software Specification and Design*, pages 6–15.

Aquilani, F., Balsamo, S., and Inverardi, P. (2001). Performance analysis at the software architectural design level. *Perform. Eval.*, 45(2-3):147–178.

Balsamo, S., Bernardo, M., and Simeoni, M. (2002). Combining stochastic process algebras and queueing networks for software architecture analysis. In *Workshop on Software and Performance*, pages 190–202.

Bass, L., Clements, P., and Kazman, R. (1998). *Software Architecture in Practice*. Addison Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Bosch, J. (1999). Design and use of industrial software architectures. In *Proceedings of Technology of Object-Oriented Languages and Systems*.

Cheung, R. (1980). A user-oriented software reliability model. *IEEE Trans. on Software Engineering*, 6:118–125.

Denning, P. and Buzen, J. (1978). The operational analysis of queueing network models. *ACM Computing Surveys*, 10:225–261.

Dobrica, L. and Niemelae, E. (2002). A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, 28:638–653.

Franco, J., Barbosa, R., and Rela, M. (2012). Automated reliability prediction from formal architectural descriptions. In *WICSA/ECSA*, pages 302–309.

Giannakopoulou, D., Kramer, J., and Cheung, S. (1999). Behaviour analysis of distributed systems using the tracta approach. *Journal of Automated Software Engineering*, 6(1):7–35.

Gokhale, S. S. (2007). Architecture-based software reliability analysis: Overview and limitations. *IEEE Trans. Dependable Sec. Comput.*, 4(1):32–40.

Goseva-Popstojanova, K. and Trivedi, K. (2001). Architecture-based approach to reliability assessment of software systems. *Journal of Performance Evaluation*, 45(2-3):179–204.

Kompanek, A. (1998). Acmestudio user’s manual.

Krishnamurthy, S. and Mathur, A. P. (1997). On the estimation of reliability of a software system using reliabilities of its components. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, ISSRE ’97, pages 146–155.

Kwiatkowska, M., Norman, G., and Parker, D. (2009). Prism: Probabilistic model checking for performance and reliability analysis. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):40–45.

Medvidovic, N. and Taylor, R. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 11(1):70–93.

Shooman, M. (1976). Structural models for software reliability prediction. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE ’76, pages 268–280.

Spitznagel, B. and Garlan, D. (1998). Architecture-based performance analysis. In *Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering*, pages 146–151.