Automated Quantitative Attributes Prediction from Architectural Description Language

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, Non Functional Attributes, Architectural Description Language, Acme, Bottleneck Analysis, Quality Attributes, Performance, Reliability, Maintainability, Availability.

Abstract: Software architecture has become an increasingly important research topic in recent years. Concurrently much more attention has been paid to methods of evaluating non functional attributes of these architectures. However, in current architectural description languages (ADLs) based on a formal and abstract model of system behavior, there is a notable lack of support for representing and reasoning about non functional attributes. In this paper, we propose ACME+ ADL as an extension of ACME ADL and discuss our quantitative model for formal analysis of software architectures. This paper gives an overview of our formal approach for describing software architectures and analyzing their performance, reliability, maintainability and availability. The proposed model is supported by an automated tool that transforms an architecture described in ACME+ into a set of inequalities characterizing system non functional attributes. These inequalities are then solved using Mathematica in order to obtain system properties as function of its components and connectors properties.

1 INTRODUCTION

Software systems are characterized by both their functional requirements (what the system does) and by their non functional requirements (how the system behaves with respect to some observable attributes like performance, reusability, reliability, etc.). Specification of non-functional properties at the architectural level of a software system can be used to describe a system, analyze it, or direct decisions to make refinements. Therefore, the representation and analysis of non-functional attributes are taken into account early in the software life cycle allowing not only early problems detection and cost benefits but also assuring that the chosen architecture will meet both functional and non-functional quality requirements.

In current practice, very few of the non-functional requirements are automatically checked. This manual checking activity is prone to errors and time-consuming due to the complex designs, as a result of a high number of components, connectors and the interconnections between them, as well as the possible architectural decisions. Therefore, an automated verification, validation and testing of the quality attributes of a software architecture is becoming more needed for practitioners. Also, there is a notable lack of support for non-functional attributes in existing Architecture Description Languages(ADLs). Acme (Garlan et al., 1997), Aesop (Garlan et al., 1994), Weaves (Gorlick and Razouk, 1991) 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 work, we propose an ADL based formal method for representing and reasoning about system non-functional attributes at the architectural level. We are interested in performance attributes (response time and throughput), reliability (failure probability), maintainability and availability. We aim to translate an architectural description into a set of inequalities that characterize non functional attributes of software architectures. These inequalities are then optimized using Mathematica (©Wolfram Research) in order to obtain system properties as function of components and connector properties. The architect can analyze a software system by varying all the architecture constituents (e.g., components, connectors) and their non-functional attributes, test and compare it by obtaining the system non-functional attributes output given by that set. The architect can then compare the obtained results and inform the practitioner on what

Derbel I., Labed Jilani L. and Mili A..
Automated Quantitative Attributes Prediction from Architectural Description Language.
DOI: 10.5220/0005003700870094
In *Proceedings of the 9th International Conference on Software Paradigm Trends* (ICSOFT-PT-2014), pages 87-94
ISBN: 978-989-758-037-6

solution is the most reliable, efficient, available, etc. switch its attempts and preferences.

This paper is organized as follows. In section 2, we discuss requirements of our analysis model. In section 3, we present the main syntactic features that we have added to Acme. In section 4, we present the Mathematica inequalities produced using an inductive reasoning. In section 5, we discuss bottleneck analysis; in section 6, we show how we generate the proposed analysis tool. We give an illustrative example in section7. The paper concludes in section 8.

2 MODEL REQUIREMENTS

We submit the following requirements as a long term goals for our model:

- If we give the values of the non-functional properties of components and connectors, we want to derive the corresponding values for the whole architecture.
- If we are given a system architecture, and given components and connectors quantitative non functional attributes, we want the model to help us propagate inductively the requirements from components and connectors to the whole system.
- If we are given values of the non functional attributes of the components and connectors of an architecture, we want to determine the sensitivity of the system attributes with respect to component and connector attributes. In other words, if we want to enhance a given system attribute, which component or connector should we alter? or, which component or connector is the bottleneck to the current attribute value?

For the purposes of this paper, we content ourselves with the first and second goals. In order to enable us to represent and reason about non functional properties of software architectures, we need an architectural model characterized by:

- The ability to represent the architecture of components, their ports and connectors, their roles.
- The ability to represent quantitative non functional properties of components and connectors.
- The ability to represent operational properties of the architecture, in addition to its topological properties. For example, if we have two components A and B connected in parallel between a shared source and a shared sink, we want to determine whether they play complementary roles (in which both are needed for normal operation) or alternative roles (if any one is operational, the system is operational). While these two configurations have the same topology, they represent radically different architectures, with different operational properties values. Thus, at a minimum, we must be able to identify, among ports of a component (and roles of a connector) which ports are used for input and which ports are used for output. Furthermore, if we have more than one input port or more than one output port, we need to represent the relation between the ports: are they mutually synchronous or asynchronous? Do they carry duplicate information? or disjoint/ complementary information? or overlapping information?
- The ability to represent for each component (connector) more than one relation between input ports (source roles) and output ports (destination roles). The reason we need this provision is that often the same component (or connector) may be involved in more than one operation, where each operation involves a different configuration of ports (roles), and have different values for its non functional attributes.
- The availability of automated support.

None of the existing ADLs meets all the requirements defined above. However, the Acme ADL is the most suitable for our purposes but after including some extensions. Acme ADL was chosen for many reasons. First, unlike many ADLs(e.g. Darwin and Rapide), it provides accurate textual notations to describe software architectures, distinguishing between the different architectural elements: components, connectors, and configuration. Second, it is supported by AcmeStudio, a modeling, parsing and code generation tool. Furthermore, Acme is a pivot language that allows to switch from one ADL to another to benefit from the complementary capabilities of ADLs. Hence, to cater to the five requirements we have presented above, we adopt Acme's basic syntax and ontology of architecture description, and add to it the concept of `Functional Dependency`. We refer to the extended Acme ADL as ACME+.

3 ACME+ SYNTAX

In order to represent and reason about non functional attributes at the architectural level, we propose the extension of Acme ADL with a new concept of `Functional Dependency`. The new construct of Functional Dependency defined in a component (connector) description aims to represent relations between component ports (connector roles)

and specifies its quantitative non functional properties. Each component (connector) has only one `Functional Dependency` construct described by one or many dependency relations. Each relation expresses that the component (connector) needs input data received by its input ports (source roles) to proceed and produce data to its output ports (destination roles). This process is characterized by a set of non-functional attributes. Consequently, each relation description connects input ports (source roles) and output ports (destination roles) and is characterized by specific non functional attributes values. A `Functional Dependency` description is written at the end of a component specification, 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` consists of:

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

As an example, let's take a `Functional Dependency` construct that includes only one dependency relation. We may write:

```
fundep {Relation Name //name of the dependency relation
input (Input Ports relations),
output(Output Ports relations),
properties(ProcTime=0.02 s, Thruput = 45 trans/s,
           FailProb =0.001, MTTR= 120 h , MTBF= 2300 h)}
```

Identification of Input and Output ports:
At a minimum, the dependency must specify which ports are used for input and which ones are used for output. If a component has, say five ports, P1, P2, P3, P4, P5 and we wish to record that P1, P2, P3 are the input ports and P4,P5 are the output ports, we write:

```
input(P1,P2,P3),
output(P4,P5).
```

Relations between Input ports:
For input ports, we must specify whether they provide alternative/duplicate information (the component may proceed with data from any one of the ports) or complementary information (component needs data from all input ports before it proceeds); also, there are cases where we may need a majority of input ports to proceed. We write respectively:

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

Relations between Output ports: 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)).
```

4 ACME+ SEMANTIC

We want to map ACME+ source code onto a set of Mathematica inequalities that characterize the non functional attributes of the architecture of interest. To this effect, we assume that: (1) all ports of components are labeled for `input` or for `output`; all roles of connectors are labeled as `fromRole` or as `toRole`; (2) each port of each component and each role of each connector has an attribute for each property of interest labeled RT (response time), TP (throughput), FP (failure probability), MTTR and MTBF; (3) there is a single component without input port and 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. The system non functional properties are computed from the properties attached to the components and connectors (ProcTime, TransTime, Thruput, FailProb, MTTR, MTBF) and represent attributes associated to the input port of the `sink` component, namely:

```
System.ResponseTime = sink.input.RT,
System.Throughput = sink.input.TP,
System.FailureProbability = sink.input.FP,
System.MTTR = sink.input.MTTR,
System.Availability= sink.input.MTBF /
      (sink.input.MTBF + sink.input.MTTR)
```

In order to obtain the values of these attributes, we decide to maximize or minimize attribute values attached to the input port of the `sink` component taking into account inequalities generated inductively.

These inequalities are obtained by propagating attributes from the source component to the sink component in a stepwise manner. They are written as $\leq$ for the maximum problem and $\geq$ for the minimum problem. Thus, as we want to maximize system throughput and MTBF, we write $\leq$ inequalities and try to maximize *sink.input.TP*, *sink.input.MTBF*, subject to the corresponding inequalities. Also, as we aim to minimize system response time, failure probability and maintainability, we write $\geq$ inequalities and try to minimize *sink.input.RT*, *sink.input.FP*, *sink.input.MTTR*, subject to the corresponding inequalities.

4.1 Basis of Induction

The output port of the source component has trivial values for all the attributes, namely:

```
source.output.RT >= 0,
source.output.FP >= 0,
source.output.MTTR >= 0,
source.output.TP <= infinity,
source.output.MTBF <= infinity.
```

4.2 Inductives Rules between Components and Connectors

Whenever a port is attached to a role, the attribute values are passed forward from the output port to the source role. As an example, we present inequalities generated for Response time and throughput quality attributes:

```
C.output to N.originRole
```

We write:

```
C.output.TP = N.originRole.TP
C.output.RT = N.originRole.RT
```

4.3 Inductive Rules within Components

We present inequalities of the components and leave it to the reader to see how the inequalities of the connectors can be derived by analogy.

4.3.1 Single Input/ Single Output

Given a component *C*, we write an inequality that links the attributes of the input port(*input*), the attributes of the output port(*output*), and the properties of the component. We write the inequalities:

```
C.output.RT >= C.input.RT + C.ProcTime.
C.output.FP >= 1-(1-C.input.FP)(1-C.failProb).
C.output.MTTR >= C.input.MTTR + C.MTTR.
C.output.TP <= Min(C.input.TP;C.thruPut).
C.output.MTBF <= (C.input.MTBF x C.MTBF)/
                 (C.input.MTBF + C.MTBF).
```

4.3.2 Multiple Inputs and Outputs

These equations depend on the nature of the functional dependency relations. Let *C* designate a component, whose input ports are called $input1;...;inputn$ and output ports are called $output1;...;outputk$. We suppose that these input and output ports are related with a functional dependency relation *R* expressed as follows:

```
R(
Input(InSelection(InSynchronisation
(input1; ..; inputn)));
Output(OutSelection(OutSynchronisation
(output1; ..; outputk)));
Properties(procTime=0.7;thruPut=0.2;failProb=0.2)
)
```

We review in turn the five attributes of interest.

- **Response Time**

For each output port $outputi$ expressed in the relation *R*, inequalities depend on the construct *InSelection*, expressing the nature of the relation between input ports.
If InSelection is **AllOf**, then *C* needs all information from its input ports to proceed and generate an output. We write:

```
C.outputi.RT>=Max(C.input1.RT;..;C.inputn.RT)
              + C.R.procTime.
```

If InSelection is **AnyOf**, then the receipt of the first input information from any input port, triggers *C* running. We write:

```
C.outputi.RT>=Min(C.input1.RT;..;C.inputn.RT)
               + C.R.procTime.
```

- **Throughput**

For each output port $outputi$ of the component *C* expressed in the relation *R*, we write an inequality relating the component's throughput and $inputiP$. 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.outputi.TP<=Min(C.R.thruPut;(C.input1.TP+..+
                               C.inputn.TP)).
```

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

```
C.outputi.TP<=Max(Min[C.R.thruPut;C.input1.TP];..;
             Min[C.R.thruPut; C.inputn.TP]).
```

- **Failure Probability**

For each output port $outputi$ of the component *C* expressed in the relation *R*, we write an equation relating component's failure probability and input ports failure probability. This rule depends on whether all of

inputs are needed, or any one of them. We first consider that *inputi* provide complementary information (InSelection is AllOf). A computation initiated at *C.outputi* 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. Whence we write:

```
C.outputi.FP>=1-(1-C.input1.FPx..xC.inputn.FP)
                (1 - C.R.FailProb).
```

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

```
C.outputi.FP >= 1-(1-C.input1.FP)x..x
    (1 - C.inputn.FP)(1- C.R.FailProb).
```

- **Maintainability**

For each output port *outputPi* expressed in the relation *R*, inequalities depend on the construct InSelection , expressing the nature of the relation between input ports.
If InSelection is **AllOf**, then *C* needs all information from its input ports to proceed and generate an output. Let's recall that the maintainability of the system is expressed in terms of MTTR quality attribute. Hence we propagate MTTR through the architecture and write:

```
C.outputi.MTTR>=C.R.MTTR+C.input1.MTTR+..+
                C.inputn.MTTR.
```

If InSelection is **AnyOf**, then the receipt of the first input information from any input port, triggers *C* running. We write:

```
C.outputi.MTTR>=C.R.MTTR+(C.input1.MTTRx..x
                C.inputn.MTTR)/
               (C.input1.MTTR+..+C.inputn.MTTR)
```

- **Availability**

For each output port *outputi* expressed in the relation *R*, inequalities depend on the construct InSelection, expressing the nature of the relation between input ports. The system availability is expressed in terms of system MTTR and system MTBF as explained in section 3. MTTR inductive inequalities have been already discussed. Hence, we present MTBF related inequalities.
If InSelection is **AllOf**, then *C* needs all information from its input ports to proceed and generate an output, we write:

```
C.outputi.MTBF<=(C.R.MTBFxC.input1.MTBFx...x
                C.inputn.MTBF)x
  (C.R.MTBF+C.input1.MTBF+...+C.inputn.MTBF)
```

If InSelection is **AnyOf**, then the receipt of the first input information from any input port, triggers *C* running. We write:

```
C.outputi.MTBF<=(C.R.MTBFxC.input1.MTBF x...x
                C.inputn.MTBF)/
   (C.R.MTBF+C.input1.MTBFx...xC.inputn.MTBF)
```

5 BOTTLENECK ANALYSIS

Bottleneck analysis forms the core of system performance analysis. In our tool we have decided to apply bottleneck analysis laws of queueing networks. In the following we present these laws. A more detailed explanation can be found in (Denning, 2008): The utilization of the i^{th} device(component or connector) is defined by:

$$U_i = X \times D_i \tag{1}$$

where X is the system throughput and D_i is the total service demand on the i^{th} device for all visits V_i of a task with processing time S_i. D_i is defined by the following law:

$$D_i = V_i \times S_i \tag{2}$$

where $V_i = X_i/X$, X_i is the i^{th} device throughput. Since utilization U_i cannot exceed 1, then $X \times D_i \leq 1$. Consequently, for each device *i*, the system throughput X checks the following law:

$$X \leq \frac{1}{D_i} \tag{3}$$

Therefore, the component or connector with largest D_i limits the system throughput and is the bottleneck. In other words, we must compute D_i for each device in the system. Let us assume that $D_j = Max\{D_1, D_2, ..., D_j, ..., D_n\}$; device j is the bottleneck. In order to make these results into practice, our tool, automatically computes the demand property of each component and connector defined by the constituents properties:

$$D = \frac{(C.ThruPut \times C.ProcTime)}{System.Throughput} \tag{4}$$

Then it picks up the device having the maximum value of *D*. There are several ways to deal with a bottleneck component: speed it up or reduce its demand in the system. After applying one of these options, we can re-compute the new quality attributes of the software and compare the improvements that will result. The main advantage of quality attributes analysis at architectural level is to detect such problems at an early stage. Since considering such changes at a late stage can be expense, difficult or even unfeasible.

6 AUTOMATED TOOL FOR SOFTWARE ANALYSIS

In order to put the proposed model into practice, we have resolved to proceed as follows:

- We adopt ACME+ as the architectural description language on which we attach our analysis model. ACME+ is based on Acme's ontology to which we add `functional dependency` construct. The new construct expresses operational information and represents components and connectors non functional properties.
- We define an attribute grammar on top of ACME+ syntax, which assigns attributes such as response time, throughput, failure probability, etc to all the ports and all the roles of the architecture. This attribute grammar can in principle be used as a synthesized grammar, propagating actual attributes up the syntax tree, or as an inherited grammar, propagating required / hypothetical attributes down the syntax tree.
- We define semantic rules in the form of inequalities that involve these attributes, and attach them to various reductions of ACME+ BNF. The rules in question are nothing but the inductive inequalities we have discussed in the previous section, along with associated operations (symbol table operations).
- We use compiler generation technology to generate a compiler for the augmented ACME+ language. The purpose of this compiler is to generate inequalities that involve the attributes associated to the ports and roles of the architecture.
- To compute the system wide properties of the architecture (such as response time, throughout, failure probability, etc), all we have to do is solve the inequalities derived by the compiler. The resolution is achieved by Mathematica (©Wolfram Research) by maximizing (sink.input.TP, sink.input.MTBF) and minimizing (sink.input.RT, sink.input.FP, sink.input.MTTR). Mathematica can solve the inequalities either symbolically (by keeping component properties and connector properties unspecified, and producing an expression of the overall system attributes as a function of the component and connector properties) or numerically (by assigning actual values to component properties and connector properties and producing numerical values for the overall system).

In order to facilitate the use of the compiler for analyzing system architecture, we have developed a graphical user interface (GUI). In its current version, the compiler generates inequalities pertaining to response time, throughput, failure probability, maintainability and availability; each of these attributes corresponds to a tab in the GUI. Once we select a tab, we can perform the following operations:

- Compute the system level attribute as a function of component level properties. The GUI does so by merely solving the system of inequalities for the unknown
  sink.inpPort.AT, for attribute AT (where AT is the attribute identified by the selected tab). When a tab is selected, the GUI posts this value automatically.
- The GUI allows the user to update the value of a property of a component or connector, and will re-compute and post the updated value of the selected system level attribute.
- Once a tab is selected, the GUI also generates, and posts in a special purpose window, the symbolic expression of the corresponding attribute as a function of relevant properties of components and connectors.
- To enable a user to assess the sensitivity of the system level attribute with respect to component level properties, the GUI shows a curve that plots the system level attribute on the Y axis and the component level property on the X axis.
- Finally, for some attributes, the GUI can also identify the component or connector that is the bottleneck of system performance for the selected attribute. Once the bottleneck of the architecture is identified, the user can change the value of its relevant property and check for the new (possibly distinct) bottleneck.

Space limitations preclude us from illustrating our analysis model with examples of system analysis. A demo of the tool that we developed, which includes the compiler and the user interface, is available online at: http://web.njit.edu/∼ mili/granada.exe. In this demo, we present the use of the analysis tool in order to analyze Aegis Weapons System (Allen and Garlan, 1996).

7 EXAMPLE

7.1 System Description

Let's consider a system consisting of three interacting components: a web server(S), a web client(C), and a database(DB). We assume that the client makes

requests to the server and receives corresponding responses. The server makes a request of the database in the process of filling the client's request. For the sake of brevity, we content ourselves with giving ACME+ descriptions of only server component and we analyze performance (response time, throughput) and reliability (failure probability) quality attributes.
We have noted that the server does two different processes, each process is described by a dependency relation which links an output port to an input port and is characterized by a different quality attribute values:

- Following the receipt of the customer's request by its input port `input`, the server generates a request to the database by its output port `output`. This process requires a service time equal to 20 ms. We refer to the dependency relation describing this process as `Request`.
- Once the server receives a response from the database by its input port `input1`, it sends a response to the client by its output port `output`. This process requires a service time equal to 20 ms. We refer to the dependency relation describing this process as `Response`.

Server's ACME+ Description is given by:

```
component S ={port input,port output,
port input1,port output1,
FunDep = {
 Requete (Input( input);
          Output(output) ;
          Properties_values( ProcTime=20 ms,
                        Thruput= 600 tr/s,
                        FailProb=0.0002))
 Reponse (Input( input1);
          Output(output1) ;
          Properties_values (ProcTime=20 ms,
                        Thruput= 700 tr/s,
                        FailProb=0.0002))};
```

Several questions concerning the overall quality attributes the system should be answered, such as:

- What are system quality attributes: response time, throughput, failure probability, etc.?
- Which component is the bottleneck? Should it be upgraded or replicated?

7.2 Quality Attributes Analysis

In the following, we present the basis of induction and inequalities generated relatively to the server(S).
Basis of Induction. We write:

```
Datasource.output.RT >= 0,
Datasource.output.FP >= 0,
Datasource.output.TP <= infinity,
```

Inequalities between Ports of *S* and Roles of Connectors. Let's take the example of the link between *S* and the connector *Cn*1:

```
S.input.RT >= Cn1.toRole.RT
S.input.FP >= Cn1.toRole.FP
S.input.TP <= Cn1.toRole.TP
```

Inequalities within Connectors. Let's take the example of the connector *Cn*1 connecting *C* to *S*.

```
Cn1.toRole.RT >= Cn1.fromRole.RT+Cn1.transTime
Cn1.toRole.FP >= 1-(1-Cn1.fromRole.FP)x
                  (1-Cn1.transTime)
Cn1.toRole.TP <= Min(Cn1.fromRole.TP;Cn1.transTime)
```

Inequalities within *S*. With respect to the two dependency relations, we write:

```
S.output.RT >= S.input.RT+S.Request.ProcTime
S.output1.RT >= S.input1.RT+S.Response.ProcTime
S.output.FP >= 1-(1- S.input.FP)x
               (1-S.Request.FailProb)
S.output1.FP >= 1-(1-S.input1.FP)x
                 (1-S.Response.FailProb)
S.output.TP<= Min(S.input.TP;S.Request.thruput)
S.output1.TP<=Min(S.input1.TP;S.Response.thruput)
```

After using Mathematica to solve the inequalities, we find the system attributes presented in table 1:

Table 1: Values of systems properties.

System Attribute	Response Time (s)	Throughput (tr/sec)	Failure Probab
exp-ression	Datasink. outputP.RT	Datasink. outputP.TP	Datasink. outputP.FP
value	265	300	0.001

7.3 Bottleneck Analysis

We propose to apply bottleneck analysis to determine the component that limits system performance. For each component: client, server and database, we calculate the `request` $D_{Cp} = \frac{(Cp.thruPut \times Cp.procTime)}{System.Throughput}$, then we select the component having the largest value. Client and server components are described by two relations of functional dependency (`Request` (*Req*)and `Response` (*Resp*)). Therefore, for each component, we compute two `demand` values with respect to each dependency relation (Table 2). For example, for the server component, we compute two values $D_{S^{Requete}}$ and $D_{S^{Reponse}}$ respectively to the relations *Request* and *Response*. They are defined by:

$D_{S^{Request}} = \frac{(S.Requete.thruPut \times S.Requete.procTime)}{System.Throughput}$;

$D_{S^{Response}} = \frac{(S.Reponse.thruPut \times S.Reponse.procTime)}{System.Throughput}$

Table 2: Components demand values.

	Client	Server	Database
Demand values	$D_{C^{Req}} = 40$ $D_{C^{Resp}} = 62.5$	$D_{S^{Req}} = 30$ $D_{S^{Resp}} = 35$	$D_{DB^{Process}} =$ 77.25

D_{DB} is the maximum value, therefore the database component is the bottleneck of the system. To manage the problem of the bottleneck component, the designer can proceed in two ways:

- either replace the base data component by another component with better performance.
- or duplicate this component to distribute the workload to several other components.

In the following, we will discuss these two solutions assuming that the rest of the system remains unchanged and we will compare the new non-functional properties obtained for each solution.

- **First Solution: Component Substitution**

We assume that the database component is substituted by a single instance with improved performance: a service time of 90 s, a throughput of 350 $trans/s$. System ACME+ Description remains unchanged and the new values of system properties will be modified as shown in Table 3.

Table 3: Case of substitution: System properties.

System Attribute	Response Time (ms)	Throughput (tr/sec)	Failure Probab
expression	Datasink. outputP.RT	Datasink. outputP.TP	Datasink. outputP.FP
value	252	350	0.0002

- **Second Solution: Component Duplication**

We assume that the database component is replaced by two identical instances of databases: $DB1$ and $DB2$, each of which is characterized by a time service 103 s and a throughput 300 $trans/s$.
Suppose that the server S sends requests to $DB1$ and $DB2$ respectively via connectors $Ca1$, $Ca2$ and receives responses from databases by connectors $Cb1$ $Cb2$. We find the system attributes represented in the table 4.

Table 4: Case of duplication: System properties.

System Attribute	Response Time (ms)	Throughput (tr/sec)	Failure Probab
expression	Datasink. outputP.RT	Datasink. outputP.TP	Datasink. outputP.FP
value	265	500	0.0002

After comparing the results in tables 3 and 4, we note that the substitution of the database component by another component with better performance decreases the response time of the system, increases its throughput and decreases its probability of failure. However replication leads to an unchanged value of the response time of the system, increases the values of system throughput and reduces the probability of failure. The choice of the best solution between these two cases depends on customer preferences between getting a faster system or a system with the same response time and higher throughput. Other factors such as cost modifications can also be taken into account in the final decision.

8 CONCLUSION

In this paper, we have proposed ACME+ as an extension of Acme ADL, on which we attach our analysis model and discussed the development and operation of a compiler that compiles architectures written in this language to generate inequalities that characterize non functional attributes of software architectures. The tool that we have developed includes the compiler and the user interface, it permits the experimentation of ACME+ and its proof of concept. In case of interest it may be demonstrated at the conference.We are currently trying to compare the results obtained for some system non functional attributes with those supported by other ADLs based on a formal, abstract model of system behavior. The perspectives of our work are to add other quantitative non functional attributes and to investigate more profoundly architecture styles and dynamic architectures.

REFERENCES

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.

Denning, P. (2008). Throughput. *Wiley Encyclopedia of Computer Science and Engineering*.

Garlan, D., Allen, R., and Ockerbloom, J. (December 1994). Exploiting style in architectural design environments. *In Proceedings of SIGSOFT'94: Foundations of Software Engineering*, pages 175–188.

Garlan, D., Monroe, R., and Wile, D. (November 1997). Acme: An architecture description interchange language. *CASCON'97. Toronto, Ontario*, page 169183.

Gorlick, M. and Razouk, R. (1991). Using weaves for software construction and analysis. In *ICSE*, pages 23–34.

Medvidovic, N. and Taylor, R. (2000). A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93.