# *Compiling Software Architectures*

**Imen Derbel, Lamia Labed Jilani, Ali Mili**

**Institut Superieur de Gestion, Bardo, Tunisia** imen.derbel@yahoo.fr, lamia.labed@isg.rnu.tn

**NJIT, Newark NJ, USA** mili@cis.njit.edu

## Abstract

***The concept of software architecture emerged in the eighties as an abstraction of all the design decisions pertaining to broad system structure, component coordination, system deployment, system operation, etc. As such, software architecture deals less with functional attributes than with operational attributes of a software system. So much so that a sound discipline of software architecture consists in identifying and prioritizing important non functional attributes that we want to optimize in the software system, and using them as a guide in making architectural decisions. In light of this situation, we find it paradoxical that no architectural description language in use nowadays has any means to automatically analyze the non functional attributes of software architectures. In this paper, we present a modified version of ACME, and present a compiler of this language that allows us to analyze and reason about non functional attributes of software systems.***

## 1. Introduction: Compiling Architectures

The concept of *software architecture* has emerged in the eighties as an abstraction of the design decisions that precede functional design, and pertain to such aspects as broad system structure, system topology in terms of components and connectors, coordination between system components, system deployment, system operation, etc [Garlan and Shaw, 1996; Bass et al, 2003; Clements et al, 2010]. This concept has gained further traction through the nineties and the first decade of the millennium, by virtue of its role in many modern software engineering paradigms, such as domain engineering, product line engineering, component based software engineering, and COTS based software development [Frakes and Kang, 2007; Kang et al, 2004; Mili et al, 2002]. Whereas functional design and programming determine the functional attributes of a software product, the architecture of a software product determines its non-functional attributes, i.e. properties such as: response time, throughput, reliability, buffer capacity, availability, security, safety, etc; we refer to these as *quality attributes* of the software product.

A number of architecture description languages (ADL's) have emerged in the past two decades, including ACME (CMU), Wright (CMU), Rapide (Stanford University), SADL (SRI), Aesop (CMU), MetaH (Honeywell), C2 (UC Irvine), Lileanna (IBM/ Loral), PADL (Urbino), Unicon (CMU). Even though many of these languages embody state of the art ideas about software architectures, and despite the importance of non functional attributes in the characterization of software architectures, to the best of our knowledge none of these languages offer automated support for analyzing quality attributes of software architectures. In this paper we propose to fill this gap by proposing an ADL which is a modified version of ACME, and building a compiler for this language, with the following characteristics:

- The language is based on ACME's architecture ontology, in that it represents architectures in terms of components, connectors, ports and roles.

- It uses ACME's *Property* construct to represent the quality attributes of components and connectors; but while ACME considers the data entered under Property as a mere comment, which it does not analyze, we give it a precise syntax and use it in our analysis.

- Whereas ACME lists the ports of a component and the roles of a connector, and does not specify any relation between the ports of a component or the roles of a connector, we introduce special purpose constructs that specify these relations, and use them in our analysis.

- Whereas programming language compilers generate executable code, that represents the functional attributes of a software product, our compiler generates equations that characterize the non-functional attributes of the product. These equations are written as Mathematica (© Wolfram Research) equations.

We use Mathematica to analyze and solve these equations. Among the questions that we envision to address/ answer, we cite the following:

- Given a set of values for the quality attributes of components and connectors, what are the values of the quality attributes of the overall system?

- How do the system-wide attribute values depend on component-level and connector-level values?

- How sensitive are system-wide attribute values to variations in component-level and connector-level values?

- Which component-level and connector-level attribute values are causing a bottleneck in system-wide attribute values?

In section 2, we briefly present and motivate the main syntactic features that we have added to ACME; in section 3, we discuss the semantics of these constructs, in terms of Mathematica equations that we associate to them. In section 4 we discuss the generation of a compiler that reads product architectures written in the proposed language and supports the analysis of the architecture through a user interface. This system is illustrated in section 5 through a sample example; the paper concludes in section 6 by a discussion of our current results and of our prospects for future research.

## 2. A Language for Software Architectures: Syntax

In order to enable us to represent and reason about non functional properties of software architectures, an architectural description language must fulfill the following conditions:

1. Support ACME's ontology of components, connectors, ports and roles.

2. Support the ability to represent non functional attributes of components and connectors.

3. Provide constructs that enable us to represent operational information that impacts the non functional attributes. 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?

4. Provide means for a component (or a connector) to represent more than one relation from input ports to output ports.

Among all the architecture description languages that we have considered, we have found none that meets these four requirements. Most languages focus primarily on representing the topology of the system; some languages, such as Wright [Allen, 1997] and PADL [Aldini et al, 2010] complement the topological information with operational information, but the latter is expressed in CSP [Hoare, 2004] which is too detailed for our purposes, and at the same time fails to always provide the information we need.

To cater to the four requirements we have presented above, we adopt ACME's basic syntax and ontology, and add to it the concept of *functional dependency*. We illustrate this concept with a simple/ artificial example: Imagine that we want to represent an Order Processing component (OP) in an online application; and imagine that OP has five ports, a port *Order* (that feeds orders), a port BankAuth (that feeds bank authorizations), a port *Virt* (that records virtual orders, i.e. orders that do not require anything to be shipped, such as subscriptions, memberships, etc), a port *Stock* (that maintains a database of merchandise), a port *Shipping* (that schedules shipments), and a port *Archive* (that maintains a database of customer orders). Imagine that the operation of this component depends on two orthogonal options: First, whether the order is a complimentary order (e.g. a promotional offer) or a paid order; second, whether the order involves a virtual service (membership, subscription, information, access right) or real merchandise. As a result, this component can operate in one of four modes, each with a distinct set of input ports and output ports, and each with possibly distinct quality attributes (in terms of response time, throughput, capacity, reliability, etc). To represent this situation, we write the following functional dependencies (keyword: FunDep):

```
FunDep  {   FreeVirt ();

            FreeReal ();

            PayVirt  ();

            PayReal  ();   }
```

This (yet incomplete) description fits at the end of the component description, after the declaration of all the ports; we refer to each item in the FunDep declaration as a *functional dependency relation* of the component. Each relation is defined by three terms, namely: an input, an output, and an aggregate of properties (reflecting quality attributes). If the input field refers to a single port, then we write **input(⟨port⟩).** But if the input field refers to multiple ports, then we need to specify:

- Whether all of them are needed (if they provide complementary information), or any one of them (if they are interchangeable), or most of them (in a fault tolerance scheme).

- Whether they have to make data available synchronously or asynchronously

For output ports, in case of multiple ports, we need to specify the following details:

- Whether the outputs posted on the different output ports are duplicates (i.e. identical to each other), exclusive (complementary and not overlapping) or overlapping (some data are shipped to more than one output port.
- Whether the data posted on output ports is posted simultaneously on all output ports, or is posted as available.

For the Property field of each relation, we post the names of the quality attributes (from a predefined vocabulary), along with values in predefined units (e.g. ms for response time, transactions/second for throughput, abstract number between 0 and 1 for failure probability, etc). Returning to the example above, we may write, for example:

```
FunDep  {FreeVirt (input(Order),
            output(Virt),
            properties (response_time=0.02,
                        throughput=20));
          FreeReal (input(Order),
             output(overlapping(asavailable(Stock, Shipping))),
             properties (response_time=0.035,
                         throughput=15));
          PayVirt  (input(allof(asynch(Order,BankAuth))),
             output(Virt),
             properties (response_time=0.04,
                         throughput=12));
          PayReal  (input(allof(asynch(Order,BankAuth))),
             output(overlapping(asavailable(Stock,Shipping,Archive))),
             properties (response_time=0.045,
                         throughput=10));  }
```

To test the adequacy of this language, we considered a number of sample architectures found in the literature, and we attempted to represent them using the proposed notation. The include: the AEGIS Weapons System [Allen, 1997]; The Video Animation Repainting System [Eduardo, 2008]; the e-Bay e-commerce Platform [Ahmed]; and the Rule Based System [Garlan and Shaw, 1996]. We found that: first,

all the information required by the language was readily available from the description of the systems in question; second, the information required by the language felt perfectly appropriate for an architecture-level description.

## 5. A Language for Software Architectures: Semantics

In order to use the information recorded in this notation for the purpose of analyzing software architectures, we take the following modeling decisions.

- Each port in a component is labeled for input or for output.
- Each role in a connector is labeled as an origin or a destination.
- Each architecture has a single component without input port, called the *source,* and a single component without output ports, called the *sink*.

Furthermore, in the context of this paper, we impose the following restrictions: Each component includes a single functional dependency relation; analyzing components with multiple dependency relations is not beyond the capability of our model, but is beyond the scope of this paper. It would require the introduction of a probability distribution between the various relations in a single component, and would require that we estimate lower bounds and upper bounds for the system attributes.

Given these conditions, we define as attribute grammar that assigns to each port and each role a set of attributes that are related to the quality attributes we are interested in. Hence each port has a response time attribute called RT, a throughput attribute called TP, a reliability attribute called FP (for failure probability), etc. Then we assign to the output port of the source component trivial values for these attributes, such as zero for the response time, zero for failure probability, infinity for throughput, etc. Then for each functional dependency relation we associate an equations between the attributes of the ports and roles that are involved in the relation. The equation depends of course on the nature of the functional dependency; for example, if two ports are linked by an allOf construct, the response time associated with the output ports of the relation is the max of the response times associated to the output ports to which we add the processing time of the components, and the throughput associated to the output ports is the min of the throughputs associated with the input ports, and the throughput capacity of the component. This process is illustrated in the following section.

### Response Time

For each port or role we associate the following attribute: port.RT is the minimal time that takes a computation proceeding from DataSource to reach this port. role.RT is the minimal time that takes a computation proceeding from DataSource to reach this role.

Whence we define the following semantic rules:

- We assign zero to the output port of *DataSource ,* we write:

  *DataSource.outputPort.RT = 0.*

- For each component *C*, having *multiple* input ports and *multiple* output ports we have:

For each functional dependency R, expressed as following:

FunDep : {

R ( Input(In_selection ( In_synchronisation (*inputPort1, inputPort2, …, inputPortn*)));

Output(Out_selection ( Out_synchronisation (*outputPort1, outputPort2, …, outputPortk*)) ) ;

Properties ( processing_time = 0.2 sec;)

)

};

For each output port *outputPort_i* expressed in the relation *R,* we write:

C.outputPort_i.RT = *function* (C.inputPort1.RT, C.inputPort2.RT, … , C.inputPortn.RT) + C.R.ProcessingTime.

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

- If In_selection is AllOf, *function* is the maximum, we write:

  C.outputPort_i.RT = *Max* (C.inputPort1.RT, C.inputPort2.RT, … , C.inputPortn.RT) + C.R.ProcessingTime.

- If In_selection is AnyOf, *function* is the minimum, we write:

  C.outputPort_i.RT = *Min* (C.inputPort1.RT, C.inputPort2.RT, … , C.inputPortn.RT) + C.R.ProcessingTime.

- For each connector N we have :

  N.receiverRole.RT = N.senderRole.RT + N.TransmissionTime

- For each attachment statement of the form

  C.ouputPort to N.receiverRole

  We write:

  C.outputPort.RT = N.receiverRole.RT

- For each attachment statement of the form

  C.inputPort to N.senderRole

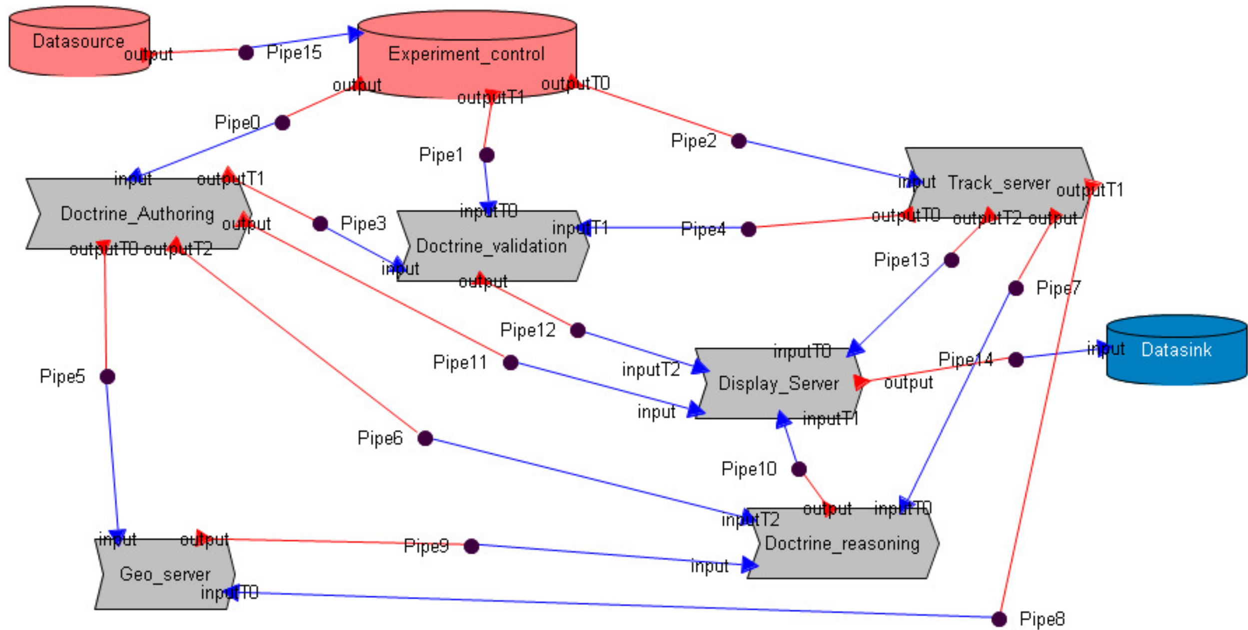
  We write:

  C.inputPort.RT = N.senderRole.RT

## 6. A Language for Software Architectures: A Compiler

As an illustration, we consider the architecture of the Aegis Weapons System [Allen, 1997]. We are interested in analyzing the system wide response time of this architecture. A graphic representation of this architecture is given below.

**Aegis System**




In this architecture, *Datasource* is a unique source component, and *Datasink* is a unique sink component. We show below the code we write for some of the components of this architecture, along with the Mathematica equations that our compiler generates from this code.

```
Component Datasource{ ... // ACME code describing this component
FunDep : {R1 ( ;  Output(output);
                Properties ( processing_time = 0 sec; )
               )
          };
```

The compiler generates the following Mathematica equation:

*Datasource. output.RT =0.*

For the following component,

```
Component Display_Server { ... // ACME code describing this component
FunDep : {R1 ( Input( AllOf(Synchronous(inputT0, inputT1, inputT2, input)));
                Output(output);
                Properties ( processing_time = 1 sec; )
               )
```

```
    };
```

The compiler generates the following Mathematica equation:

*Display_Server .output.RT = Max (Display_Server .inputT0.RT,  Display_Server .inputT1.RT,*

*Display_Server .inputT2.RT , Display_Server.input.RT)+ Display_Server .R1.RT*

For the following component,

```
Component Doctrine_Authoring {...  // ACME code describing this component
FunDep : { R1 ( Input( Synchronous(input));
                Output(Duplicate(Simultaneous (output, outputT1, outputT0, outputT2))) ;
                Properties ( processing_time = 0.7 sec;)
               )
        };
```

The compiler generates the following Mathematica equations:

*Doctrine_Authoring . output.RT = Doctrine_Authoring .R1.PT + Doctrine_Authoring . Input.RT*

*Doctrine_Authoring . outputT0.RT = Doctrine_Authoring . R1.PT + Doctrine_Authoring . Input.RT*

*Doctrine_Authoring . outputT1.RT = Doctrine_Authoring .R1.PT + Doctrine_Authoring . Input.RT*

*Doctrine_Authoring . outputT2.RT = Doctrine_Authoring . R1.PT + Doctrine_Authoring . Input.RT*

For the following component,

```
    Component Doctrine_validation  {  ...  // ACME code describing this component
FunDep : { R1 (  Input( AllOf(Synchronous(inputT0, inputT1, input)));
                 Output(output) ;
                 Properties ( processing_time = 0.7 sec;)
               )
            };
```

The compiler generates the following Mathematica equations:

*Doctrine_validation.output.RT = Max (Doctrine_validation.inputT0.RT,*
*Doctrine_validation.inputT1.RT, Doctrine_validation.input.RT)+ Doctrine_validation.R1.RT*

For the following component,

```
    Component Track_server {  ...  // ACME code describing this component
FunDep : {R1 (  Input(Synchronous(input)) ;
                Output(Duplicate(Simultaneous (outputT2, outputT1, outputT0, output))) ;
```

```
                    Properties ( processing_time = 1 sec; )
                )
        };
```

The compiler generates the following Mathematica equations:

*Track_server. output.RT = Track_server .R1.PT + Track_server. Input.RT*

*Track_server. OutputT0.RT = Track_server .R1.PT + Track_server. Input.RT*

*Track_server. OutputT1.RT = Track_server .R1.PT + Track_server. Input.RT*

*Track_server. OutputT2.RT = Track_server .R1.PT + Track_server. Input.RT*

For the following component,

```
   Component Geo_server  {  ...  // ACME code describing this component
FunDep : { R1 (Input( AllOf(Synchronous(inputT0, input)));
               Output(output) ;
              Properties ( processing_time = 3 sec;)
              )
         };
```

The compiler generates the following Mathematica equations

*Geo_server. output.RT = Geo_server .R1.PT + Max(Geo_server. Input.RT ; Geo_server. InputT0.RT )*

For the following component,

```
   Component Doctrine_reasoning  {  ...  // ACME code describing this component
FunDep : {R1 ( Input( AllOf(Synchronous(inputT0, inputT2, input)));
               Output(output) ;
               Properties ( processing_time = 1 sec;)
                )
         };
```

The compiler generates the following Mathematica equations

*Doctrine_reasoning. output.RT = Doctrine_reasoning .R1.PT +*

*Max(Doctrine_reasoning. Input.RT ; Doctrine_reasoning. InputT0.RT ; Doctrine_reasoning.InputT2.RT)*

Finally, for the following component,

```
Component Experiment_control  {  ...  // ACME code describing this component
FunDep : {R1 (  Input( Synchronous(input));
                Output(Exclusive(AsAvailable(outputT1, outputT0, output)) ) ;
                 Properties ( processing_time = 0.2 sec;)
               )
         };
```

The compiler generates the following Mathematica equations:

*Experiment_control. output.RT = Experiment_control.R1.PT + Experiment_control. Input.RT*

*Experiment_control. outputT0.RT = Experiment_control.R1.PT + Experiment_control. Input.RT*

*Experiment_control. outputT1.RT = Experiment_control.R1.PT + Experiment_control. Input.RT*

After we generate the equations pertaining to the connectors, and equate system attributes to the attributes of the output port of the sink component, we find equations that characterize the non functional attributes of the system as a function of the component and connector attributes.

## 7. An Automated Tool

We have developed an automated tool that analyzes architectures according to the pattern discussed in this paper. This tool invokes the compiler to generate the Mathematica equations that characterize the system's non-functional attributes. Then it uses the equations 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. A demo of this tool can be downloaded from the following address: http://web.njit.edu/~mili/arcdemo1.avi. This tool can be demo-ed at the conference in case of acceptance.

**Symbolic Resolution:*

System.RT= Datasink•input•RT=

Display_Server•R1•PT+Experiment_control•R1•PT+

**Max[**Doctrine_Authoring•R1•PT+Pipe0•TT+Pipe11•TT **,**

Track_server•R1•PT+ Pipe13•TT+Pipe2•TT**,**

Doctrine_validation•R1•PT+Pipe12•TT+Max[Pipe1•TT,Doctrine_Authoring•R1•PT+Pipe0•TT+Pipe3•TT,Pipe2•TT+Pipe4•TT+Track_server•R1•PT] **,**

**A** ]

Where

**A**=Doctrine_reasoning•R1•PT+Pipe10•TT+

Max[Doctrine_Authoring•R1•PT+Pipe0•TT+Pipe6•TT,

Pipe2•TT+Pipe7•TT+Track_server•R1•PT,
Geo_server•R1•PT+Pipe9•TT+Max[Doctrine_Authoring•R1•PT+Pipe0•TT+Pipe5•TT,Pipe2•TT+Pipe8•TT+Trackserver•R1•PT] ]

** Numeric Application*

By substituting component properties and connector properties by their values, given in the table below, we find the following system response time :System.RT= 7.23

| | ProcessingTime | TransmissionTime |
|---|---|---|
| Pipe15 | (null) | (null) |
| Pipe14 | (null) | (null) |
| Pipe13 | (null) | 0.9 |
| Pipe12 | (null) | 0.5 |
| Pipe11 | (null) | 0.2 |
| Pipe10 | (null) | 0.2 |
| Pipe9 | (null) | 0.2 |
| Pipe8 | (null) | 0.42 |
| Pipe7 | (null) | 0.4 |
| Pipe6 | (null) | 0.2 |
| Pipe5 | (null) | 0.7 |
| Pipe4 | (null) | 0.5 |
| Pipe3 | (null) | 0.3 |
| Pipe2 | (null) | 0.12 |
| Pipe1 | (null) | 0.2 |
| Pipe0 | (null) | 0.23 |
| Display_Server.R1 | 1 | (null) |
| Doctrine_reasoning.R1 | 1 | (null) |
| Geo_server.R1 | 3 | (null) |
| Track_server.R1 | 1 | (null) |
| Doctrine_validation.R1 | 0.7 | (null) |
| Doctrine_Authoring.R1 | 0.7 | (null) |
| Experiment_control.R1 | 0.2 | (null) |
| System RT | Datasink.input.RT -> DisplayServer.R1.PT + Experimentcontrol.R1.PT + Max[DoctrineAuthoring.R1.PT + Pipe0.TT + Pipe11.TT, Pipe13.TT + Pipe2.TT + Trackserver.R1.PT, Doctrinevalidation.R1.PT + Pipe12.TT + Max[Pipe1.TT, DoctrineAuthoring.R1.PT + Pipe0.TT + Pipe3.TT, Pipe2.TT + Pipe4.TT + Trackserver.R1.PT], Doctrinereasoning.R1.PT + Pipe10.TT + Max[DoctrineAuthoring.R1.PT + Pipe0.TT + Pipe6.TT, Pipe2.TT + Pipe7.TT + Trackserver.R1.PT, Geoserver.R1.PT + Pipe9.TT + Max[DoctrineAuthoring.R1.PT + Pipe0.TT + Pipe5.TT, Pipe2.TT + Pipe8.TT + Trackserver.R1.PT]]] | 7.23 |

Screenshot of the tool showing Aegis components and connectors properties after the compilation of Acme description.

Acme compiler and property analyser

File Edit Compile Properties

```
Component Experimentcontrol : Filter = new Filter Extended With{
  Port outputT1 : outputT = new outputT Extended With {

  };
  Port outputT0 : outputT = new outputT Extended With {

  };
  Port output : outputT = new outputT Extended With {

  };
  Port input : inputT = new inputT Extended With {

  };

FunDep : {
      R1 ( Input( Synchronous(input));
         Output(Exclusive(AsAvailable(outputT1, outputT0, output)) ) ;
         Properties_values ( processing_time = 0.2 sec;throughput = 2 trans/sec; )
       )
      };
  };

  Component DoctrineAuthoring : Filter = new Filter Extended With{
    Port outputT0 : outputT = new outputT Extended With{

    };
    Port outputT1 : outputT = new outputT Extended With{

    };
    Port outputT2 : outputT = new outputT Extended With{

    };
    Port input : inputT = new inputT Extended With{

    };
    Port output : outputT = new outputT Extended With{

    };

 FunDep : {
```

Table

| | ProcessingTime | TransmissionTime | Throughput | FailureProb |
|---|---|---|---|---|
| Pipe15 | (null) | (null) | (null) | (null) |
| Pipe14 | (null) | (null) | (null) | (null) |
| Pipe13 | (null) | 0.9 | 0.6 | (null) |
| Pipe12 | (null) | 0.5 | 0.4 | (null) |
| Pipe11 | (null) | 0.2 | 0.2 | (null) |
| Pipe10 | (null) | 0.2 | 0.3 | (null) |
| Pipe9 | (null) | 0.2 | 0.1 | (null) |
| Pipe8 | (null) | 0.42 | 0.31 | (null) |
| Pipe7 | (null) | 0.4 | 0.3 | (null) |
| Pipe6 | (null) | 0.2 | 0.1 | (null) |
| Pipe5 | (null) | 0.7 | 0.6 | (null) |
| Pipe4 | (null) | 0.5 | 0.23 | (null) |
| Pipe3 | (null) | 0.3 | 0.13 | (null) |
| Pipe2 | (null) | 0.12 | 0.05 | (null) |
| Pipe1 | (null) | 0.2 | 0.1 | (null) |
| Pipe0 | (null) | 0.23 | 0.12 | (null) |
| DisplayServer.R1 | 1 | (null) | 3 | (null) |
| Doctrinereasoning.R1 | 1 | (null) | 2 | (null) |
| Geoserver.R1 | 3 | (null) | 1 | (null) |
| Trackserver.R1 | 1 | (null) | 4 | (null) |
| Doctrinevalidation.R1 | 0.7 | (null) | 2 | (null) |
| DoctrineAuthoring.R1 | 0.7 | (null) | 1 | (null) |

Screenshot of the tool showing symbolic and numeric resolution of equations generated by the compiler.

Acme compiler and property analyser

File Edit Compile Properties

```
Component Experimentcontrol : Filter = new Filter Extended With{
  Port outputT1 : outputT = new outputT Extended With {

  };
  Port outputT0 : outputT = new outputT Extended With {

  };
  Port output : outputT = new outputT Extended With {

  };
  Port input : inputT = new inputT Extended With {

  };

FunDep : {
      R1 ( Input( Synchronous(input));
         Output(Exclusive(AsAvailable(outputT1, outputT0, output)) ) ;
         Properties_values ( processing_time = 0.2 sec;throughput = 2 trans/sec; )
       )
      };
  };

  Component DoctrineAuthoring : Filter = new Filter Extended With{
    Port outputT0 : outputT = new outputT Extended With{

    };
    Port outputT1 : outputT = new outputT Extended With{

    };
    Port outputT2 : outputT = new outputT Extended With{

    };
    Port input : inputT = new inputT Extended With{

    };
    Port output : outputT = new outputT Extended With{

    };

 FunDep : {
```

Table

| | ProcessingTime | Transmissio... | Thr... | Fai... |
|---|---|---|---|---|
| Pipe15 | (null) | (null) | (null) | (nu... |
| Pipe14 | (null) | (null) | (null) | (nu... |
| Pipe13 | (null) | 0.9 | 0.6 | (nu... |
| Pipe12 | (null) | 0.5 | 0.4 | (nu... |
| Pipe11 | (null) | 0.2 | 0.2 | (nu... |
| Pipe10 | (null) | 0.2 | 0.3 | (nu... |
| Pipe9 | (null) | 0.2 | 0.1 | (nu... |
| Pipe8 | (null) | 0.42 | 0.31 | (nu... |
| Pipe7 | (null) | 0.4 | 0.3 | (nu... |
| Pipe6 | (null) | 0.2 | 0.1 | (nu... |
| Pipe5 | (null) | 0.7 | 0.6 | (nu... |
| Pipe4 | (null) | 0.5 | 0.23 | (nu... |
| Pipe3 | (null) | 0.3 | 0.13 | (nu... |
| Pipe2 | (null) | 0.12 | 0.05 | (nu... |
| Pipe1 | (null) | 0.2 | 0.1 | (nu... |
| Pipe0 | (null) | 0.23 | 0.12 | (nu... |
| DisplayServer.R1 | 1 | (null) | 3 | (nu... |
| Doctrinereasoning.R1 | 1 | (null) | 2 | (nu... |
| Geoserver.R1 | 3 | (null) | 1 | (nu... |
| Trackserver.R1 | 1 | (null) | 4 | (nu... |
| Doctrinevalidation.R1 | 0.7 | (null) | 2 | (nu... |
| DoctrineAuthoring.R1 | 0.7 | (null) | 1 | (nu... |
| Experimentcontrol.R1 | 0.2 | (null) | 2 | (nu... |
| System.RT | Datasink.input.RT -> DisplayServer.R1.PT + Experimentcontrol.R1.PT + Max[Doctri... | 7.23 | | |

## 8. Conclusion: Summary, Assessment, and Prospects

In this paper, we have discussed the need to develop automated tools to analyze software architectures written in a formal ADL. Also, we have proposed an ADL that is an extension of ACME, and discussed the development and operation of a compiler that compiles architectures written in this language to generate equations that characterize non functional attributes of software architectures. The tool that we have developed, which includes the compiler and the user interface, may be demo-ed at the conference in case of interest. Our work can be characterized by the following attributes, which set it apart from other work on architectural analysis [Aldini and Bernardo, 2005; Aldini et al, 2010; Balsamo et al, 2003; Clements et al, 2010; Spitznagel and Garlan, 1998]:

- It is based on ACME's architectural ontology,
- It is based on the architectural-level concept of functional dependency,
- It supports symbolic analysis of architectural attributes, by means of symbolic equations generated by Mathematica (in addition to numeric analysis, which computes actual system attributes as a function of component and connector attributes).
- It is supported by an automated tool.

This work is clearly in its infancy; it is no more than a proof of concept to the effect that it is possible to reason automatically about non functional attributes of software architectures, given sufficient architectural information and component/ connector attributes. Among the extensions we envision for this work, we cite:

- Extend our work to cases where the same component may have more than one functional dependency relation.
- Extend our work to other non functional attributes; in the longer term, extent it to user defined attributes, that then need to be axiomatized by the user to support automated reasoning.
- Make the inductive rules more flexible/ more generally applicable, by replacing the current inductive equations with inequalities, and replacing the current equation resolution by function optimization.
- In the longer term, we envision to broaden our model from a "components and connectors" view of architectures, to accommodate other architectural styles.
- Concurrently, we are also considering a radically different approach to architectural analysis, which consists in computing non functional attributes by means of general graph algorithms, such as shortest path, or maximum flow, or minimum spanning tree, etc.

# 9. References

**[Ahmed]:** Mohammad Usman Ahmed. *E-Bay E-Commerce Platform: A Case Study in Scalability*. Technical Report, School of Computer Science, McGill University, Montreal, Quebec, Canada. Available online at: http://www.cs.mcgill.ca/~mahmed26/eBay_Architecture_Study.pdf

**[Aldini and Bernardo, 2005]:** Alessandro Aldini and Marco Bernardo. *On the usability of process algebra: An architectural view*. Theoretical Computer Science, vol 335, no 2-3, pages 281-329, 2005.

**[Aldini et al, 2010]:** Alessandro Aldini, Marco Bernardo and Flavio Corradini. *A process Algebraic Approach to Software Architecture Design*. Springer Verlag, 2010.

**[Allen, 1997]:** A Formal Approach to Software Architecture. Ph.D. Thesis, Carnegie Mellon University, Technical Report Number: CMU-CS-97-144, May, 1997.

**[Balsamo et al, 2003]:** Simonetta Balsamo, Marco Bernardo and Marta Simeoni, *Performance Evaluation at the Software Architecture Level.* Proceedings, SFM 2003: Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003, Bertinoro, Italy, September 22-27, 2003, pages 207-258.

**[Bass et al, 2003]:** Len Bass, Paul Clements, Rick Kazman: *Software Architecture in Practice, Second Edition*. Addison Wesley, Reading 5/9/2003 ISBN 0-321-15495-9.

**[Buschmann et al, 2007]:** Frank Buschmann; Kevlin Henney; Douglas C. Schmidt (2007). *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. John Wiley & Sons.

**[Clements et al, 2010]:** Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, Judith Stafford: *Documenting Software Architectures: Views and Beyond, Second Edition*. Addison-Wesley, 2010, ISBN 0321552687.

**[Frakes and Kang, 2007]:** Frakes, William B.; Kang, Kyo (July 2007). "Software Reuse Research: Status and Future". *IEEE Transactions on Software Engineering.* **31** (7): 529–536.

**[Garlan and Shaw, 1996]:** David Garlan and Mary Shaw (1994). "An Introduction to Software Architecture: Perspectives on an Emerging Discipline". Prentice Hall, 1996.

**[Hoare, 2004]:** C.A.R. Hoare. Communicating Sequential Processes. June 2004. Manuscript available online at: http://www.usingcsp.com/cspbook.pdf

**[Kang et al, 2004]:** Kang, Kyo C.; Lee, Jaejoon; Kim, Kijoo; Kim, Gerard Jounghyun; Shin, Euiseob (October 2004). "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures". *Annals of Software Engineering* (Springer Netherlands): 143–168.

**[Mili et al, 2002]:** Mili, H., A. Mili, Sh. Yacoub and E. Addy. Reuse Based Software Engineering: Techniques, Organizations, and Controls. John Wiley and Sons, 2002.

**[Spitznagel and Garlan, 1998]:** Bridget Spitznagel and David Garlan. *Architecture-Based Performance Analysis*. Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering. Pages 146-151, 1998.