Reasoning About Quantitative Architectural Attributes

(Invited Paper)

Lamia Labed Jilani
Institut Suprieur de Gestion, Tunis University, RIADI Lab, Tunisia
Email: Lamia.Labed@isg.rnu.tn

Imen Derbel
Institut Suprieur de Gestion, Tunis University, Tunisia
Email: imen.derbel@yahoo.fr

Khaled Bsaies
Fac. of Science, Univ of Tunis, El Manar University, URPAH Laboratory, Tunisia
Email: khaled.bsaies@fst.rnu.tn

Hamdi Nasreddine
Fac. of Science, Univ of Tunis, El Manar University, Tunisia
Email:hamdi.nasreddine@yahoo.fr

Ali Mili
New Jersey Inst. of Technology, Newark NJ 07102-1982 USA
Email: mili@cis.njit.edu

***Abstract*— It is widely accepted that software architectures represent non functional attributes of software systems. Yet we know of no Architectural Description Language that provides automated support for reasoning about such attributes. In this paper we discuss our ongoing research in representing and reasoning about non functional properties of software architectures.**

Keywords

Software architecture, non functional attributes, ACME, Wright, response time, throughput, reliability, security, availability, maintainability.

I. Introduction and Premises

It is widely accepted that while the source code of a software product represents its functional attributes, its architecture represents its non functional attributes. The architecture of a software product determines such attributes as its response time, throughput, reliability, security, maintainability, availability, etc. A sound discipline of software architecture consists in identifying the most important non functional attributes that we want our software product to achieve, and take architectural decisions to optimize these.

In light of these observations, and to the extent that they are valid, it is rather surprising that current architectural description languages do not give more consideration to the ability to represent non functional attributes and reason about them. For example, ACME [1], [2] does allow the software architect to represent non functional attributes through the construct of `properties`, but has a wide open syntax for such attributes, and does not perform any analysis on these. Also, because ACME represents topological information but does not represent any operational information, it does not capture all the information that is required to reason about non functional properties. Wright, on the other hand, does include operational information in the form of CSP expressions involving processes and events; but Wright offers no constructs for representing non functional attributes of components and connectors [1], [4].

In this paper we present a model for reasoning about quantitative non functional attributes of software architectures, and discuss our plan to develop automated support for this model. In section II we discuss the basic idea of our model, and in section III we present inductive rules that form the basis of our approach. In section IV we discuss the implementation of this model on an extension of Wright, along with simple illustrative examples. Finally in section V we summarize and assess, briefly discuss related work, and sketch directions of future research.

II. A Quantitative Model for Analyzing Architectures

A. Model Requirements

Given that software architectures determine the non functional attributes of software products, it is important, we

doi:10.4304/jsw.6.4.574-583

feel, that we be able to quantify, compute and reason about these attributes on an architectural representation. In the same way that the source code of a software system is mapped onto executable machine code that embodies its functional attributes, we want the architecture of the system to be mapped onto a set of equations that capture and characterize its non functional attributes.

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

- If we are given values of the non functional attributes of the components and connectors of an architecture, we want to derive the corresponding values for the whole architecture.
- 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?
- Given a system requirement formulated in terms of a quantitative non functional attribute, and given a system architecture, we want the model to help us propagate the requirements down the hierarchy to components and connectors, in such a way as to ensure that the system-level requirement is met.
- Given a non functional attribute to optimize (perhaps at the expense of the other attributes), and given architectural constraints, which architecture or architecture family helps to optimize the selected attribute?

The first goal is clearly a prerequisite for all the others, hence we focus on it in this paper, and will briefly discuss our plans for the other goals.

B. Architecture Model

For the sake of our project, we need an architectural model that offers the following features:

1) The availability of the concepts of component and port, as defined by ACME [3].
2) The availability of the concepts of connector and role, as defined by ACME [3].
3) The ability to assign relevant non functional properties to components and connectors.
4) 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.
5) The availability of automated support for reasoning about architectural attributes.

We have found that ACME meets all the requirements but the fourth, and Wright meets all the requirements but the third. We have chosen to adopt Wright for our purposes, while adding to it ACME's syntax for representing properties. We envision that an analyst uses the Wright toolset to represent her/ his architecture and check its syntax; then she/ he can add ACME-style specifications for non functional attributes to the Wright source, and submit the whole architecture specification to the compiler that we are producing, so that it peforms the analysis of its non functional attributes.

C. Non Functional Attributes

For the purpose of this discussion, we consider three sample non-functional properties, namely:

- Response time, measured in milliseconds. We assume that each component has a property of type real called *ProcessingTime* and each connection has a property of type real called *TransmissionTime*.
- Throughput, measured in transactions per second. We assume that each component has a property of type integer called *ProcessingThroughput* and each connector has a property of type integer called *TransmissionThroughput*.
- Reliability, measured as a probability. Though reliability is typically measured by the mean time to failure, we choose to represent it here as the mean probability of failure over a unitary period of operating time. Under some hypotheses, it is possible to convert failure probabilities into MTTF's, though we do not do that here. We assume that each component and each connector has a property of type real called *FailureProbability*.

Other standard non functional attributes that we can define (though we do not do so in this paper) include: Security properties (probability of intrusion, probability of intrusion detection, probability of security violation), Buffer capacity (maximum number of transactions received but not processed), Availability (percentage of time a component or connector is operational), Safety (probability of violating no safety requirement over a unitary period of operating time), etc.

Extensions of the model may include user-defined properties, provided the user also provides the necessary axiomatisation for them; subsequent discussions will highlight what form such axiomatisations may take.

D. A Logical Framework

We consider a Wright architecture with the following characteristics:

- All the ports of all the 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 the roles of connectors are labeled as origin or as destination; connectors carry data or control information from their origin roles to their destination roles.
- The architecture has a single component that has no input port; we call this component the *source*.
- The architecture has a single component that has no output port; we call this component the *sink*.

We refer to an architecture that satisfies these conditions as a *canonical architecture*. We are not sure to what extent these conditions constitute a loss of generality. Architectures that have no source component or no sink components cannot be transformed to satisfy the last two conditions, but they may occur very rarely; however, architectures that have more than one component without input port or more than one component without output port can be transformed to satisfy the last two conditions by adding a dummy source and a dummy sink and connecting them appropriately. The connectors used for this purpose will have trivial values for their properties, such as: zero transmission time, infinite throughput, and zero failure probability.

Given a canonical architecture, we define its system-wide attributes as follows:

- All the dummy connectors that may have been introduced to make the architecture canonical have trivial property values: for any connector K, we have:

$$K.TransmissionTime = 0,$$

$$K.Throughput = \infty,$$

$$K.FailureProbability = 0.$$

- For each non functional property we are interested in (for the time being, response time, throughput, and failure probability), associate an attribute to each port and each role of the architecture. Each port and each role is given the following attributes: RT (for response time), TP (for throughout), and FP (for failure probability). See Figure 1.
- The values of the non functional properties for the overall system are then the values of the relevant attributes for the output port of the source component; hence the response time of the system is $source.RT$; the throughput of the system is $source.TP$; and the failure probability of the system is $source.FP$. The values of these attributes are computed inductively from the properties attached to the components and connectors (ProcessingTime, TransmissionTime, ProcessingThroughput, TransmissionThroughput, FailureProbability). The inductive process is defined as follows.
- *Basis of Induction*. The basis of induction defines the values of the relevant attributes for the input port of the sink component. We write:

$$sink.inputPort.RT = 0.$$

Figure 1. Properties and Attributes

$$sink.inputPort.TP = \infty.$$

$$sink.inputPort.FP = 0.$$

- *Inductive Step: Within Components and Connectors*. For each component, we write an equation that links the attributes of the input ports, the attributes of the output ports, and the properties of the component. Likewise, for each connector, we write an equation that links the attributes of the origin role, the attributes of the destination role, and the properties of the connector. These equations will be discussed in some detail in section III.
- *Inductive Step: Between Components and Connectors*. Whenever a port of a component is attached to the role of a connector, their attributes are equated. For example, if the output port of component C is attached to the origin role of connector K, we write:

$$C.outputPort.RT = K.originRole.RT.$$

$$C.outputPort.TP = K.originRole.TP.$$

$$C.outputPort.FP = K.originRole.FP.$$

The inductive formulas within a component and within a connector depend on the attribute in question (RT, TP, FP), on the topology of the component and connector, and on relations between ports and roles within the same component or connector. We present them in the next section.

III. INDUCTIVE RULES

The purpose of the inductive rules is to formulate equations that allow us to propagate values of the relevant attributes (RT, TP, FP) through the architecture. For each component, these equations link the values of the attributes at the input ports and output ports with the values of internal properties (ProcessingTime, ProcessingThroughput, FailureProbability). Likewise, for each connector, these equations link the values of the attributes at the origin roles and destination roles with the values of internal properties (TransmissionTime, TransmissionThroughput, FailureProbability). For the sake of readability, we first present these rules in the simple context where each internal component has a single input port and a single output port, and each connector has a single origin role and a single destination role.

A. Single Entry, Single Exit

We let C designate a component, whose ports are called *inputPort* and *outputPort*, and let K designate a connector, whose roles are called *originRole*, and *destinationRole*. We review in turn, the attributes of response time, throughput, and failure probability.

Response time. For each component C, we write:

$$C.inputPort.RT =$$

$$C.outputPort.RT + C.ProcessingTime.$$

For each connector K, we write:

$$K.originRole.RT =$$

$$K.destinationRole.RT + K.TransmissionTime.$$

Throughput. For each component C, we write:

$$C.inputPort.TP =$$

$$min(C.outputPort.TP,\ C.ProcessingThroughout).$$

For each connector K, we write:

$$K.originRole.TP =$$

$$min(K.destinationRole.TP,$$

$$K.TransmissionThroughout).$$

Failure Probability. For each component C, we write:

$$C.inputPort.FP =$$

$$1 - (1 - C.outputPort.FP)$$

$$\times(1 - C.FailureProbability).$$

For each connector K, we write:

$$K.originRole.FP =$$

$$1 - (1 - K.destinationRole.FP)$$

$$\times(1 - K.FailureProbability).$$

B. Multiple Exits, Entries

If all our components have no more than one input port and no more than one output port, and if all our connectors have no more than one origin role and no more than one destination role, then the only architecture we can represent is a linear topology such as the pipe-and-filter architecture. But we want to analyze architectures that have arbitrary topologies, where components have an arbitrary number of ports of any type (inout or output) and connectors have arbitrary an arbitrary number of roles of any type (origin/ destination).

The study of general inductive rules for components with multiple input ports and output ports and for connectors with multiple origin roles and destination roles is beyond the scope of this paper. What we will do, in this paper, for the sake of illustration, is present sample inductive rules for two output ports and two destination roles, in two discrete cases:

- The case where the output ports and the destination roles provide complementary information; for example, an order processing component sends payment information to the financial processing component and shipping information to the warehouse management component.
- The case where the output ports and the destination roles provide duplicate information: for example, a real time component sends time critical sensor data to two redundant components, so that the first component that analyzes it and produces actuator data can control the actuator (and temporarily override the slowest component).

For the sake of brevity, we will present the rules for components, and leave it to the reader to imagine how the rules for connectors can be derived by analogy.

Response Time. We consider a component C with a single input port called *inputPort* and two output ports that provide complementary information, called *half1Port* and *half2Port*. We submit the following inductive rule pertaining to response time:

$$C.inputPort.RT =$$

$$C.ProcessingTime + max(C.half1Port.RT,$$
$$C.half2Port.RT).$$

We consider a component C with a single input port called *inputPort* and two output ports that provide redundant/ duplicate information, called *copy1Port* and *copy2Port*. We submit the following inductive rule pertaining to response time:

$$C.inputPort.RT =$$

$$C.ProcessingTime + min(C.copy1Port.RT,$$
$$C.copy2Port.RT).$$

Note that to determine which case we are in requires a through analysis of the roles that the two output ports play in the computation. For example, if the output ports produce

safety critical data that is sent to two redundant components in a fault tolerant scheme, where their outputs are compared before they are used, then the first formula is used, rather than the second. The precise analysis of Wright's behavior description code under computation for components, and glue for connectors, is necessary to determine what situation we are in, and possibly to identify other situations. This analysis is beyond the scope of this paper, and is currently under investigation.

Throughput. We consider a component C with a single input port called *inputPort* and two output ports that provide complementary information, called *half1Port* and *half2Port*. We submit the following inductive rule pertaining to throughput:

$$C.inputPort.TP =$$

$$min(min(C.ProcessingThroughput, C.half1Port.TP), min(C.ProcessingThroughput, C.half2Port.TP)).$$

We consider a component C with a single input port called *inputPort* and two output ports that provide redundant/ duplicate information, called *copy1Port* and *copy2Port*. We submit the following inductive rule pertaining to throughput:

$$C.inputPort.TP =$$

$$max(min(C.ProcessingThroughput, C.half1Port.TP), min(C.ProcessingThroughput, C.half2Port.TP)).$$

Failure Probability. We consider a component C with a single input port called *inputPort* and two output ports that provide complementary information, called *half1Port* and *half2Port*. We submit the following inductive rule pertaining to failure probability:

$$C.inputPort.FP =$$

$$1 - (1 - C.FailureProbability) \times$$

$$(1 - C.half1Port.FP) \times$$

$$(1 - C.half2Port.FP).$$

Justification: In order for a computation that is initiated at $C.inputPort$ to succeed, component C has to succeed, and the computations initiated at the two output ports of C have to succeed. Assuming statistical independence, the probability of these simultaneous events is the product of probabilities.

We consider a component C with a single input port called *inputPort* and two output ports that provide redundant/ duplicate information, called *copy1Port* and *copy2Port*. We submit the following inductive rule pertaining to failure probability:

$$C.inputPort.FP =$$

$$1 - (1 - C.FailureProbability) \times$$

$$(1 - C.copy1Port.FP \times C.copy2Port.FP).$$

Justification: In order for a computation that is initiated at $C.inputPort$ to succeed, component C has to succeed, and one of the computations initiated at output ports copy1Port and copy2Port has to succeed. Whence we write:

$$C.inputPort.FP =$$

$$1 - (1 - C.FailureProbability) \times$$

$$PsPorts,$$

where $PsPorts$ is the probability that (at least) one of the computations initiated at the ports succeeds. We have

$$PsPorts = 1 - PfPorts,$$

where $PfPorts$ is the probability that both computations initiated at the output ports fail. Under the hypothesis of statistical independence of port failures, we find

$$PfPorts = C.copy1Port.FP \times C.copy2Port.FP.$$

C. Prospects

The formulas discussed above illustrate the kind of inductive argument we want to build as we compute system attributes, but they are too piecemeal to be generally applicable. The model we are envisionning for arbitrary architectures can be characterized by the following premises:

- *Equation Generation is Local, and Equation Analysis is Global*. When components have a single input port and a single output port, and when connectors have a single origin role and a single destination role, we basically have one equation per component and one equation per connector. But when components (resp. connectors) have multiple ports (resp. roles) of each type (input/ output, resp. origin/ destination), it becomes impossible to catalog all the possible configurations of inter-port (resp. inter-role) relations and generate an equation for each. What we envision are localized equations that use partial information, and do not require a global knowledge of other related ports (resp. roles).
- *Local Equations are Inequalities, and Global Analysis is Optimization*. In section III-A we had generated straightforward equations that reflect our analysis of the architecture; all that remained was to combine the equations and solve them in the unknowns that interest us (the attributes of the output port of the source component). Under the more general model, the individual equations are inequalities, reflecting partial/ localized information. Global analysis then takes the form of an optimization step, whereby we seek the smallest value for RT (the response time), the largest value for TP (the throughput) and the smallest value for FP (the failure probability).
  For example, if we know that component C passes part of its output information to output port *Port1* (and assume that the system computation completes only if

all parts of the data have been processed), then we can write:

$$C.ProcessingTime+C.Port1.RT \leq C.InputPort.RT.$$

To write this equation, we only need to look at port $Port1$. On the other hand, if we know that component C passes part of its output information to output port *Port2* (and assume that the system computation completes only if all parts of the data have been processed), then we can write:

$$C.ProcessingTime+C.Port2.RT \leq C.InputPort.RT.$$

To write this equation, we only need to look at port $Port2$. If $Port1$ and $Port2$ are the only two output ports that component C has (a global property) then the smallest value of $C.InputPort.RT$ can be derived by optimization as:
$C.InputPort.RT = max(C.ProcessingTime + C.Port1.RT, C.ProcessingTime + C.Port2.RT)$
which we can write as:
$C.InputPort.RT = C.ProcessingTime+ max(C.Port1.RT, C.Port2.RT)$.

The main research challenge that we face in implementing this approach is to derive the inequations using architectural information; Wright and ACME appear to be too detailed for our purposes in some aspects, and too sketchy in others. This matter is currently under investigation.

IV. An Automated Tool for Analyzing Non Functional Attributes

A. A Synthesized Attribute Grammar

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

- We adopt Wright [1], [4] as the architectural description language on which we attach our analysis model. We add to this language ACME's properties construct, in which the property value is a quantitative value of predefined unit (millisecond for response time, transactions per second for throughput, probability for failure probability, etc).
- We define an *attribute grammar* on top of Wright's 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. Because the downward propagation is not deterministic, we restrict ourselves to the first interpretation for now.
- We define semantic rules in the form of equations that involve these attribute, and attach them to various reductions of Wright's BNF. The equations in question are nothing but the inductive equations we have discussed in the previous section, along with associated bookkeeping operations (symbol table operations and the like).
- We use compiler generation technology to generate a compiler for the augmented Wright language. The purpose of this compiler is to generate equations 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), all we have to do is solve the equations derived by the compiler taking for unknowns the values attached to the output port of the source component ($source.outputPort.RT$, $source.outputPort.TP$, $source.outputPort.FP$, etc).

This compiler is currently under construction, using compiler generation technology; the equations it generates are written in *Mathematica* (©Wolfram Research). Some of the difficulties that arise in this task include:

- The difficulty of identifying the input ports and output ports of a component.
- The difficulty of identifying the origin roles and the destination roles of a connector.
- Most especially, the difficulty of identifying the relations between the input ports of a component, the output ports of a component, the origin roles of a connector, and the destination roles of a connector, even in the simple case where we have no more than two ports and no more than two roles of the same type.

An easy solution to these difficulties would be simply to increase the language to force the architect to make these determinations. But for principled reasons (not to create a new language for each new problem), as well as pragmatic reasons (not to worry about maintaining a viable support environment for the language), we are reluctant to give in to this option, yet.

The equations generated by the compiler can be used in one of two ways:

- *Numerically*, by assigning actual values to component properties and connector properties, and having Mathematica produce actual numerical values for the overall architecture. This is illustrated in section IV-B.
- *Symbolically*, by keeping component properties and connector properties unspecified, and having Mathematica produce an expression of the overall system attributes as a function of the component and connector properties. This form is useful for, e.g., sensitivity analysis. If we want to increase the throughput of the overall system and are interested to know which component or connector needs to have its throughput increased to maximize overall impact (in other words, which component or connector is a throughput bottleneck), then we could compute the derivatives

$$\frac{d(source.outputPort.TP)}{d(C.Throughput)}$$

Figure 2. A Simple Pipe-and-Filter Architecture

for all components C, and

$$\frac{d(source.outputPort.TP)}{d(K.Throughput)}$$

for all connectors K and see which derivative takes the largest value in the present configuration. Several computer algebra systems, such as Mathematica (©Wolfram Research) can help with such steps, by computing derivatives, performing optimizations, etc.

Interested readers may look up a short demo of our prototype tool, available online at `http://web.njit.edu/mili/arcdemo.exe`. This demo works only on linear architectures, and shows how system attributes are computed from component and connector attributes by generating then solving equations.

B. A Sample Linear Architecture

As an illustrative example, we consider a very simple architecture, made up of two components, a producer and a consumer, and one connector, a channel. To this very simple configuration we add, solely for the sake of illustration, the dummy components *source* and *sink*, and their associated connectors; this is represented in Figure 2. To make matters simple, we adopt standard names for all the ports, and roles of the architecture: component ports are called *inputP* and *outputP*; connector roles are called *originR* and *destR* (destination role). We write, in turn, the basis of induction equations then the inductive step equations, involving the attributes of response time, throughput, and failure probability.

Basis of Induction. We write the following equations:

$sink.inputP.RT = 0.$
$sink.inputP.TP = \infty.$
$sink.inputP.FP = 0.$

Induction Step: Attachments Between Ports and Roles.

From *downstream* to *sink*:

$downstream.destR.RT = sink.inputP.RT.$
$downstream.destR.TP = sink.inputP.TP.$
$downstream.destR.FP = sink.inputP.FP.$

From *consumer* to *downstream*:

$consumer.outputP.RT = downstream.originR.RT.$
$consumer.outputP.TP = downstream.originR.TP.$
$consumer.outputP.FP = downstream.originR.FP.$

From *channel* to *consumer*:

$channel.destR.RT = consumer.inputP.RT.$
$channel.destR.TP = consumer.inputP.TP.$
$channel.destR.FP = consumer.inputP.FP.$

From *producer* to *channel*:

$producer.outputP.RT = channel.originR.RT.$
$producer.outputP.TP = channel.originR.TP.$
$producer.outputP.FP = channel.originR.FP.$

From *upstream* to *producer*:

$upstream.destR.RT = producer.inputP.RT.$
$upstream.destR.TP = producer.inputP.TP.$
$upstream.destR.FP = producer.inputP.FP.$

From *source* to *upstream*:

$source.outputP.RT = upstream.originR.RT.$
$source.outputP.TP = upstream.originR.TP.$
$source.outputP.FP = upstream.originR.FP.$

Inductive Equations Within Connectors.

We have three connectors, *upstream*, *channel* and *downstream*. We apply the equations of section III, and find, for *downstream*:

$downstream.originR.RT$
$= downstream.destR.RT$
$+downstream.TransmissionTime.$
$downstream.originR.TP$
$= min(downstream.destR.TP,$
$downstream.Throughput).$
$downstream.originR.FP$
$= 1 - (1 - downstream.destR.FP) \times$
$(1 - downstream.FailureProbability).$

For *upstream*:

$upstream.originR.RT$
$= upstream.destR.RT + upstream.TransmissionTime.$
$upstream.originR.TP$
$= min(upstream.destR.TP, upstream.Throughput).$
$upstream.originR.FP$
$= 1 - (1 - upstream.destR.FP) \times A$
$A = (1 - upstream.FailureProbability).$

For *channel*:

$channel.originR.RT$
$= channel.destR.RT + channel.TransmissionTime.$
$channel.originR.TP$
$= min(channel.destR.TP, channel.Throughput).$
$channel.originR.FP$
$= 1 - (1 - channel.destR.FP) \times A$
$A = (1 - channel.FailureProbability).$

Connectors *upstream* and *downstream* are fictitious connectors, that have zero transmission time, infinite throughput, and zero failure probability. We use this information to simplify the inductive equations above, and find, not unexpectedly: For *downstream*:

$downstream.originR.RT = downstream.destR.RT.$
$downstream.originR.TP = downstream.destR.TP.$
$downstream.originR.FP = downstream.destR.FP.$

For *upstream*:

$upstream.originR.RT = upstream.destR.RT.$
$upstream.originR.TP = upstream.destR.TP.$
$upstream.originR.FP = upstream.destR.FP.$

Inductive Equations Within Components. We have four components, namely *source*, *producer*, *consumer*, and *sink*. Component *source* has no inpput port, hence we need not apply inductive rules to it; also, component *sink* has no output port, hence we need not apply inductive rules to it. We are left with two components, whose equations we write below. For component *producer*, we write:

$producer.inputP.RT$
$= producer.outputP.RT$
$+producer.ProcessingTime.$
$producer.inputP.TP$
$= min(producer.outputP.TP,$
$producer.Throughput).$
$producer.inputP.FP$
$= 1 - (1 - producer.outputP.FP)$
$\times(1 - producer.FailureProbability).$

For component *consumer*, we write:

$consumer.inputP.RT$
$= consumer.outputP.RT$
$+consumer.ProcessingTime.$
$consumer.inputP.TP$
$= min(consumer.outputP.TP,$
$consumer.Throughput).$
$consumer.inputP.FP$
$= 1 - (1 - consumer.outputP.FP)$
$\times(1 - consumer.FailureProbability).$

Numeric Application. As an application, we take the following values for component properties:

Component	Processing Time (ms)	Throughput (tr/sec)	Failure Probability
producer	2.0	500	0.0002
consumer	1.5	479	0.00015

For the *channel* connector, we take the following property values:

Connector	Transmission Time (ms)	Throughput (tr/sec)	Failure Probability
channel	0.5	2000	0.00005

System Properties. By substituting component properties and connector properties by their values, given in the tables above, we find the following system attributes:

System Attribute	Response Time (ms)	Throughput (tr/sec)	Failure Probability
expression	source. outputP.RT	source. outputP.TP	source. outputP.FP
value	4	470	0.00039

C. A Sample Parallel Architecture

We consider the sample architecture represented in Figure 3. We assume that the output ports of A provide complementary information, and we call them (respectively) $A.half1Port$ and $A.half2Port$. Using the inductive rules we have presented for single input port/ single output port components and connectors, we derive the RT, TP and FP attributes of these ports. Applying the rule for multiple output ports to component A, we find the RT, TP and FP for the input port of A. By virtue of the inductive step as it applies between components and connectors, the attributes of $A.inputPort$ are identical to those of $SA.DestinationRole$. Using the inductive steps within a connector and the trivial values of the properties of connector SA, which is a dummy connector, we find that the origin role of SA has the same attributes as its destination role. Applying again the inductive step between the output port of the source component and the origin role of the SA connector, we find that the attributes of $source.outputPort$ are identical to those of $SA.originRole$. By definition (see Figure 1), the attributes of $source.outputPort$ are those of the overall architecture. For the sake of illustration, we only show the explicit formulas for response time and throughput. We write:

$System.ResponseTime = max(RT1, RT2).$
$RT1 = A.ProcessingTime + AB1.TransmissionTime$
$+B1.ProcessingTime + BC1.TransmissionTime$
$+C1.ProcessingTime.$
$RT2 = A.ProcessingTime + AB2.TransmissionTime$
$+B2.ProcessingTime + BC2.TransmissionTime$
$+C2.ProcessingTime.$
$System.Throughput = min(TP1, TP2).$
$TP1 = min(A.ProcessingThroughput,$
$AB1.TransmissionThroughput,$
$B1.ProcessingThroughput,$
$BC1.TransmissionThroughput,$
$C1.ProcessingThroughput).$
$TP2 = min(A.ProcessingThroughput,$
$AB2.TransmissionThroughput,$
$B2.ProcessingThroughput,$
$BC2.TransmissionThroughput,$
$C2.ProcessingThroughput).$

Had the output ports of component A been providing redundant information, such that processing one copy or another were sufficient, we would have had the following formulas:

$System.ResponseTime = min(RT1, RT2).$
$RT1 = A.ProcessingTime + AB1.TransmissionTime$
$+B1.ProcessingTime + BC1.TransmissionTime$

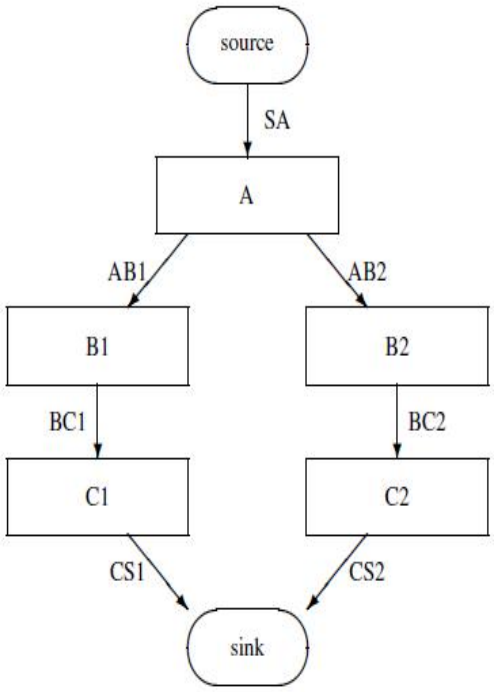

Figure 3. A Simple Parallel Architecture

$+C1.ProcessingTime.$
$RT2 = A.ProcessingTime + AB2.TransmissionTime$
$+B2.ProcessingTime + BC2.TransmissionTime$
$+C2.ProcessingTime).$
$System.Throughput = max(TP1, TP2).$
$TP1 = min(A.ProcessingThroughput,$
$AB1.TransmissionThroughput,$
$B1.ProcessingThroughput,$
$BC1.TransmissionThroughput,$
$C1.ProcessingThroughput).$
$TP2 = min(A.ProcessingThroughput,$
$AB2.TransmissionThroughput,$
$B2.ProcessingThroughput,$
$BC2.TransmissionThroughput,$
$C2.ProcessingThroughput).$

This is intuitively appealing: in the case of complementary ports, the response time is the max of those achieved by each port and the throughput is the min; in the case of duplicate ports, the response time is the min of those achieved by each port and the throughput is the max.

V. Conclusion

A. Summary and Assessment

In this paper, we have presented a tentative model that can be used to analyze the properties of a software architecture as a function of the properties of its components and connectors. The main idea of this paper is that system wide properties can be derived from local component and connector properties, as well as from attachments between components and connectors, using an inductive argument. We have illustrated our idea on simple examples, that have trivial topologies: a single source component, a single sink component, each internal component has a single input port and one or two output ports, each connector has a single origin role and a single destination role. This example is not meant to showcase the proposed approach as much as it is meant to illustrate it. The interesting applications of this approach begin with compnents and connectors having an arbitrary number of ports and roles, which then allows us to build architectures with arbitrary topologies.

We have discussed some simple rules for components with more than one output port, just to show that these general cases are not beyond the reach of our model, though they clearly make it more complicated, but also more interesting and more useful. We have also discussed how the inductive rules we propose for computing system wide properties can be used to build an attribute grammar for an architectural description language, and have briefly discussed our plans to do so on an extension of the Wright architectural description language.

In [5] Van Eenoo et al raise the question of representing non functional attributes in software architectures and propose an extention of Wright that enables them to represent *required* non functional properties as well as *ensured* non functional properties. Our work goes one step further by providing means to reason about ensured properties. Our model can also be used to reason about required properties, but we have not discussed this aspect of it in this paper. In [6], [7] Bernardo et al. present an architectural description language under the name of PADL, which is based on process algebra, and use its algebraic expressions to model and analyze operational aspects of system performance. Our approach differs from the work of Bernardo et al, in the sense that we are interested in a broader set of properties, we rely on less detailed descriptions of the architecture, and analyze the system at a higher level of abstraction.

B. Prospects

This work is currentloy in progress, and is pursued in many directions. What we view as the most pressing issues in our research plan are the following:

- How to determine, for a particular component or connector written in Wright, the classes of ports of a component (input/ output) and the classes of roles of a connector (origin/ destination).
- For a given set of ports or roles, how to determine, by analyzing topological and operational information, what relations exist between the ports and roles of the same class. This determines the choice of inductive rule, as we have seen in section III-B.
- For a given component or connector with arbitrary degree (number of neighbors), how to derive general inductive rules for the various non functional attributes, along the lines discussed in section III-C.

REFERENCES

[1] A. R. and G. D., "A formal basis for architectural connection," *ACM Transactions on Software Engineering and methodology*, vol. 6, no. 3, 1997.
[2] B.Schmerl and D.Garlan, "Acmestudio: Supporting style centered architecture development," in *Proceedings, 26th International Conference on Software Engineering*, May 2004.
[3] D.Garlan, R.T.Monroe, and D.Wile, "Acme: An architecture description interchange language," in *Proceedings, CASCON'97*, 1997.
[4] R. AAllen, "A formal approach to software architecture," Carnegie Mellon University, Pittsburgh, Technical Report CMU-CS-97-144, May 1997.
[5] C.V.Eenoo, O.Hylooz, and K. Khan, "Addressing non functional properties in software architecture using adl," in *Proceedings, Sixth Australian Workshop on Software and System Architecture*, 2005.
[6] B. M., C. P., and D. L., "Architecting families of software systems with process algebras," *ACM TOSEM: Transactions on Software Engineering and Methodology*, 2002.
[7] A. A. and B. M., "On the usability of process algebra: An architectural view," *Theoretical Computer Science*, May 2005.

Lamia Labed received a Computer Science Engineering degree from the University of Tunis in 1990, and the Ph.D. degree from the same Faculty, in 1998. She is currently teaching at the Higher Institute of Management, University of Tunis and member of the Software Engineering RIADI Laboratory. Her current research interests since 1993 focus on Software Engineering, particularly on software reuse, formal methods and software architectures. She is a co-author of a book on Discrete Mathematics and Logic.

Imen Derbel is currently a Ph.D candidate at the Institute of Management of Tunis (ISG). She received her Engineering degree in computer science in 2006 and her MS degree in 2008 from Faculty of Science of Tunis (FST). She is actually a computer science teacher in a secondary school in Tunis. Her research interests include software engineering and software architecture analysis.

Hamdi Nasreddine has received his Engineering degree in computer science in 2008 and his MS degree in 2010 from Faculty of Science of Tunis (FST).

Khaled Bsaies holds a Phd from the universite Henri Poincare de Nancy , France (1993) and an Habilitation Degree in Computer Science from University of Tunis-El Manar (Tunisia) (1999). Senior Researcher with the INRIA French National Institute of Research in Computer Science and Automata in France: 1993-1995. Full Professor in Computer Science, University of Manar (ex Univ. Tunis II), Department of Computer Science, since 2004. His research interests are in Logic Programming, Theorem Proving and Software Engineering.

Ali Mili holds a Phd from the university of Illinois, Urbana, IL, USA (1981) and a Doctorat es Sciences d'Etat from the Universite Joseph Fourier de Grenoble, France (1985). His research ineterests are in software engineering, ranging from managerial to technical aspects. A. Mili has published six books, twelve book chapters, and more than 200 papers in journals and conference proceedings.