ÆMPA: A Process Algebraic Description Language for the Performance Analysis of Software Architectures [*]

Marco Bernardo
Università di Torino
Dip. di Informatica
Corso Svizzera 185
10149 Torino, Italy
bernardo@di.unito.it

Paolo Ciancarini
Università di Bologna
Dip. di Scienze dell'Inf.
Mura Anteo Zamboni 7
40127 Bologna, Italy
cianca@cs.unibo.it

Lorenzo Donatiello
Università di Bologna
Dip. di Scienze dell'Inf.
Mura Anteo Zamboni 7
40127 Bologna, Italy
donat@cs.unibo.it

ABSTRACT

We address the problem of describing and analyzing not only functional but also performance properties of software architectures in a formal framework. We thus develop an architectural description language with a precise syntax and semantics and we illustrate the various kinds of analysis that can be conducted on its descriptions, such as architectural compatibility and conformity checking, functional verification, and performance evaluation. The proposed architectural description language is based on stochastically timed process algebras by virtue of their compositionality, which makes them suited to work with at the architectural level.

1. INTRODUCTION

The software architecture level of design allows to cope with the increasing size and complexity of software systems during the early stage of their development [21, 23]. To achieve this, the focus is turned from algorithmic and data structure related issues to the overall architecture of the system. The architecture is meant to be a collection of computational components together with a description of their connectors, i.e. the interactions between these components.

As software architecture emerges as a discipline within software engineering, it becomes increasingly important to support architectural development with languages and tools. It is widely recognized that suitable architectural description languages (ADLs for short) should be devised to formalize software architectures instead of using informal box-and-line diagrams, and related tools should be implemented to support the automatic analysis of architectural properties in order to allow the designer to make principled choices.

As far as we know, almost all the existing ADLs and tools deal only with functional aspects of software architectures. However, designers are often faced with the problem of choosing among different software architectures which are functionally equivalent. This choice is thus driven by nonfunctional factors, and mostly by performance requirements. In general, as recognized in [24], performance analysis should be integrated into the software development process, starting from the earliest stages and continuing throughout the whole life cycle.

In order to create a framework where the functional and performance properties of formally represented software systems can be automatically evaluated at the architectural level, we need a suitable theory which provides the necessary underpinnings to the architectural concepts of component and connector. This is witnessed e.g. by the fact that a well known ADL like WRIGHT [1] is based on CSP [13].

In the field of computer and communication system modeling and analysis, several formal description techniques have been proposed in the last two decades accounting for both functional and nonfunctional aspects of systems. Among such formal description techniques, we consider stochastically timed process algebras (see, e.g., [11, 3, 10]). Their key feature is compositionality. First of all, like classical process algebras, they allow for compositional model construction because they are algebraic languages endowed with a small set of powerful operators, such as parallel composition, sequential composition, and alternative composition, which allow descriptions to be systematically built from their components. Unlike classical process algebras, stochastically timed process algebras come equipped with the capability of expressing activity durations by means of exponentially distributed random variables, so that the underlying performance models turn out to be Markov chains which can thus be exploited to effectively derive performance measures. Second, stochastically timed process algebras allow for compositional model manipulation. This is achieved by means of equivalences which relate terms possessing the same functional and performance properties. Whenever such equivalences are congruences, i.e. they are substitutive w.r.t. the algebraic operators, they permit to replace algebraic components with equivalent (smaller) ones without altering the overall system properties. Third, stochastically timed process algebras allow for compositional model solution if the underlying Markov chain meets certain conditions.

[*] This research has been funded by Progetto MURST Cofinanziato SALADIN: "Architetture Software e Linguaggi per Coordinare Componenti Distribuite e Mobili" and by a grant from Microsoft Research Europe.

Since stochastically timed process algebras have been successfully used to conduct complex case studies on communication protocols and distributed algorithms (see a list in [3]) and their compositional nature make them suited to work with at the architectural level of design, in this paper we propose the adoption of stochastically timed process algebras to formally describe and analyze software architectures. From the performance standpoint, the objective is twofold: predicting the performance of software systems running on a given platform and comparing the performance of several alternative software architectures designed for the same system.

This paper is organized as follows. In Sect. 2 we recall EMPA [3], an expressive stochastically timed process algebra, and the related tool support TwoTowers [3]. In Sect. 3 we define an EMPA based ADL – called ÆMPA – inspired by WRIGHT. By means of a running example, we illustrate its textual and graphical notations, its semantics, architectural compatibility and conformity checking, functional verification, and performance evaluation. Finally, Sect. 4 concludes the paper with some remarks about related and future work.

2. A THEORY FOR PROCESS COMPOSITION AND PERFORMANCE EVALUATION

In this section we briefly recall the theory developed for the stochastically timed process algebra EMPA as well as the related tool support TwoTowers. The interested reader is referred to [3] for a complete presentation.

2.1 Syntax and Semantics

EMPA specifications are given as sets of constant defining equations of the form

$$A(type_1\, f\_par_1, \dots, type_n\, f\_par_n;\\ type_{n+1}\, l\_var_1, \dots, type_{n+m}\, l\_var_m) \stackrel{\Delta}{=} E$$

where A is a constant name with certain formal parameters and local variables, while E is a process term built out of actions (describing system activities) and algebraic operators. Formal parameters are bound to actual parameters whenever a parametrized constant invocation of the form $A(a\_par_1, \dots, a\_par_n)$ occurs. Local variables, instead, are used to get values from other system components via synchronization (see the parallel composition operator below). Both formal parameters and local variables are typed. The allowed types are int, real, bool, list, and array. It is worth noting that if E contains an invocation to A, the defining equation above is recursive. Recursive constant defining equations are typically employed to describe repetitive behaviors.

Each action $<\alpha, \tilde{\lambda}>$ is composed of the type α of the action (e.g. message transmission) and a random variable (denoted by its rate $\tilde{\lambda}$) expressing the duration of the action. As far as types are concerned, an action can be unstructured (e.g. a pure synchronization signal), carry values, or accept values. Among unstructured actions, we designate one of them as internal, i.e. not visible by an external observer, and we denote its type by τ. Invisible actions are useful to abstract from unnecessary or unknown internal details of a system. In the second case, we talk about an output action whose type is denoted $a!(\underline{e})$ where $\underline{e}$ is a vector of expressions, while in the third case we talk about an input action whose type is denoted $a?(\underline{x})$ where $\underline{x}$ is a vector of variables. Input/output actions are used to model information exchange among system components which synchronize when performing certain activities. We denote by $AType$ the set of action types.

As far as durations are concerned, the related random variables are restricted to be exponentially distributed, constant zero, or unspecified (e.g. in the case of input actions). Such random variables are concisely denoted through rates whose value is a positive real number in case of exponentially timed actions, $\infty_{l,w}$ (where number l is a priority level and number w is a weight) in case of immediate actions, and $*$ in case of passive actions. The restriction to exponential distributions causes the underlying performance models to be Markov chains, hence performance measures can be effectively derived out of algebraic specifications. We denote by $ARate$ the set of action rates.

Let us now consider the algebraic operators: guarded alternative composition, conditional, functional abstraction, functional relabeling, and parallel composition.

Term $\sum_{i=1}^{n} <\alpha_i, \tilde{\lambda}_i>.E_i$ expresses a choice among several alternatives: if action $<\alpha_j, \tilde{\lambda}_j>$ is chosen, then that action is executed and afterwards the system behaves as term E_j. If the involved actions are exponentially timed, then the choice is resolved according to the race policy: the action sampling the least duration succeeds. If immediate actions come into play, they take precedence over exponentially timed actions and the choice is resolved according to the preselection policy: the immediate actions with lower priority are discarded, then each of the remaining actions is given an execution probability proportional to its weight. If only passive actions are involved (meaning that the whole term is not completely specified from the performance standpoint), then the choice is purely nondeterministic. If $n = 0$, the guarded alternative composition above is denoted by $\underline{0}$.

Term *if* (β) *then* E_1 *else* E_2 behaves as either E_1 or E_2 depending on whether boolean expression β evaluates to true or not. Unlike the previous operator, this operator expresses a data driven choice as the behavior is determined by the value of the variables involved in expression β.

Term E/L, where $L \subseteq AType - \{\tau\}$, behaves as term E except that the type of each executed action is hidden, i.e. turned into τ, whenever it belongs to L. As an example, term $(<a, \lambda>.\underline{0} + <b, \mu>.\underline{0})/\{a\}$ can execute actions $<\tau, \lambda>$ and $<b, \mu>$. This operator provides a means to abstract from unnecessary information about the system behavior.

Term $E[\varphi]$, where $\varphi : AType \rightarrow AType$ is such that $\varphi^{-1}(\tau) = \{\tau\}$, behaves as term E except that the type α of each executed action is modified according to the action type relabeling function φ, i.e. becomes $\varphi(\alpha)$. As an example, term $(<a, \lambda>.\underline{0} + <b, \mu>.\underline{0})[a \mapsto c]$ can execute actions $<c, \lambda>$ and $<b, \mu>$. This operator provides a means to obtain more compact system descriptions.

Term $E_1 \parallel_S E_2$, where $S \subseteq AType - \{\tau\}$, asynchronously executes actions of E_1 or E_2 whose type does not belong to

S and synchronously executes actions of E_1 and E_2 of the same type belonging to S if at least one of the two actions is passive. In case of multiway synchronization, at most one exponentially timed or immediate action can be involved and it determines the overall rate; all the other involved actions must be passive, i.e. with unspecified duration. As an example, term $(<a, \lambda>.\underline{0} + <b, \mu>.\underline{0}) \parallel_{\{a,c\}} (<a, *>.\underline{0} + <c, \gamma>.\underline{0})$ can execute the synchronization action $<a, \lambda>$ as well as action $<b, \mu>$, but not $<c, \gamma>$.

In the following, we call dynamic the guarded alternative composition operator and the conditional operator, while we call static the functional abstraction operator, the functional relabeling operator, and the parallel composition operator. A term is said to be sequential if it comprises dynamic operators only.

The formal semantics for EMPA is based on a set of rules which inductively assign a state transition graph – called the integrated interleaving semantic model – to each algebraic term, where states are in correspondence with algebraic terms and transitions are labeled with actions. As an example, let us model a producer/consumer system with a buffer of capacity two. The overall system can be described as the interaction of a *Producer*, a *Buffer*, and a *Consumer*:

$$\begin{array}{rcl} \mathit{PCSystem}_2 & \triangleq & \mathit{Producer} \parallel_{\{produce\}} \\ & & \mathit{Buffer} \parallel_{\{consume\}} \\ & & \mathit{Consumer} \end{array}$$

The *Producer* repeatedly produces new items at rate λ:

$$\mathit{Producer} \triangleq <\mathit{produce}, \lambda>.\mathit{Producer}$$

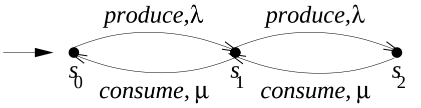
while the *Consumer* repeatedly consumes items (if available) at rate μ:

$$\mathit{Consumer} \triangleq <\mathit{consume}, \mu>.\mathit{Consumer}$$

The *Buffer*, instead, is at any time ready to accept new incoming items (if not full) and to deliver previously produced items (if not empty):

$$\begin{array}{rcl} \mathit{Buffer} & \triangleq & <\mathit{produce}, *>.\mathit{Buffer}_1 \\ \mathit{Buffer}_1 & \triangleq & <\mathit{produce}, *>.\mathit{Buffer}_2 + \\ & & <\mathit{consume}, *>.\mathit{Buffer} \\ \mathit{Buffer}_2 & \triangleq & <\mathit{consume}, *>.\mathit{Buffer}_1 \end{array}$$

Note that only passive actions occur in *Buffer*, to reflect the fact that the interaction is guided by the other two entities. The integrated interleaving semantic model of $\mathit{PCSystem}_2$ is reported below



where the initial state s_0 corresponds to $\mathit{PCSystem}_2$, state s_1 corresponds to

$$\mathit{Producer} \parallel_{\{produce\}} \mathit{Buffer}_1 \parallel_{\{consume\}} \mathit{Consumer}$$

and state s_2 corresponds to

$$\mathit{Producer} \parallel_{\{produce\}} \mathit{Buffer}_2 \parallel_{\{consume\}} \mathit{Consumer}$$

From graphs like this, a functional semantic model and a performance semantic model (in the form of a continuous time or discrete time Markov chain) can be derived by dropping action rates and action types, respectively. Such semantic models are used to analyze the functional and performance properties of the systems represented by EMPA terms.

In particular, for the integrated interleaving semantic model a notion of equivalence and a notion of preorder have been developed to formally compare EMPA terms. The former, called Markovian bisimulation equivalence, relates two terms if they describe two systems possessing the same functional and performance properties. The latter, called Markovian testing preorder [5], relates two terms describing two functionally equivalent systems if one of them is more efficient than the other one.

We conclude by recalling that, although only exponentially timed and zero durations can be directly expressed, by means of the value passing features of EMPA it is possible to express also generally distributed durations.

2.2 Tool Support

TwoTowers [3] is a software tool for modeling and analyzing functional and performance properties of computer and communication systems described in EMPA. As shown in Fig. 1, TwoTowers is composed of a graphical user interface, a tool driver, an integrated kernel, a functional kernel, and a performance kernel.

The tool driver carries out lexical, syntactic and static semantic checks on algebraic specifications. If there are errors, a message indicates the number of errors and a file is generated pinpointing the errors in the specification. If the specification is correct, instead, the control is passed to one of the three kernels for analysis.

The integrated kernel contains the routine to generate the integrated interleaving semantic model of correct, finite state EMPA specifications according to the formal semantics. The purpose of the integrated kernel is to perform those analyses that require both functional and performance information. More precisely, the integrated kernel contains a routine to check two correct, finite state EMPA specifications for Markovian bisimulation equivalence as well as a routine for the simulative analysis of the performance of a correct EMPA specification.

The functional kernel contains the routine to generate the functional semantic model of correct, finite state EMPA specifications. The functional kernel is then interfaced with a version of CWB-NC [7] that has been retargeted for EMPA. The functional kernel therefore comprises the analysis capabilities of CWB-NC such as model checking, equivalence checking, and preorder checking.

Finally, the performance kernel contains the routine to generate the performance semantic model of correct, finite state EMPA specifications. The performance kernel is then interfaced with a version of MarCA [26] that has been retargeted to interact with TwoTowers. The performance kernel therefore comprises all the analysis methods of MarCA for the computation of stationary/transient state probability distributions and reward based performance measures.

3. AN ADL FOR PERFORMANCE EVALUATION OF SOFTWARE SYSTEMS

In this section we define an EMPA based ADL – we call ÆMPA – for the functional and performance analysis of

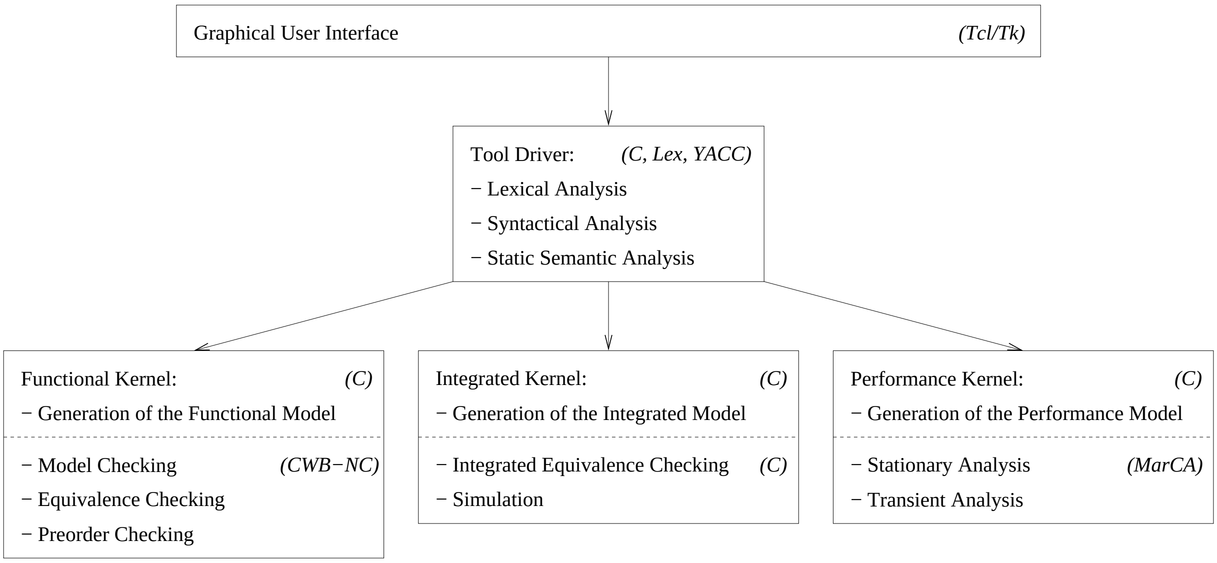


Figure 1: Architecture of TwoTowers

software architectures. Throughout this section, we use a running example to illustrate the syntax of ÆMPA (textual and graphical notation), its semantics (given by translation into EMPA), and the kinds of analysis that can be conducted (architectural compatibility and conformity checking, functional verification, and performance evaluation).

A description in ÆMPA is an architectural type. Each architectural type is defined as a function of its component and connector types, its topology, its interactions, and its generic parameters. A component/connector type is in turn defined as a function of its behavior, specified as a family of EMPA sequential terms, and its interactions, specified as a set of EMPA action types. The architectural topology consists of a set of component/connector instances related by a set of attachments among their interactions. Architectural interactions, which are interactions of component instances, support hierarchical architectural modeling. Finally, generic parameters are basically values for parametric rates and weights.

The concept of architectural type, proposed in [4], is an abstraction of the concept of architectural style, which refers to a recognized software organizational structure characterized by a vocabulary of components and connectors and a set of constraints on their interactions. Given an architectural style, the set of component and connector instances and their internal behavior can vary from architectural instance to architectural instance, but the structure of the overall interconnection of component and connector instances and their internal behavior w.r.t. interactions is fixed. Because of the presence of two degrees of freedom (variability of the set of component and connector instances and variability of their internal behavior), investigating the properties which are common to all the instances of an architectural style is not an easy task. To make such a task manageable, architectural types are advocated in [4] because they constrain the set of component and connector instances to be fixed. The instances of a given architectural type are then generated by letting the behavior of component and connector types vary. In other words, the component/connector types specified in an architectural type are viewed as being formal, so one can call for an architectural type and pass to it actual component/connector types. An example of architectural invocation will be shown in Table 3 in connection with hierarchical modeling.

3.1 Textual Notation

An architectural description in ÆMPA is as described in Table 1. In order to illustrate the various parts of an ÆMPA description, we consider a client-server system composed of one client and one server. The client generates a request at a time and waits for its outcome. The server follows a pipe-filter organizational style and consists of three processing elements with identical structure (a server with a one position buffer) but different service rates. After leaving the first processing element, an item immediately reaches one of the other two processing elements with a certain routing probability. Below, we start with the description of the pipe-filter architecture of the server.

The first part of an ÆMPA description defines the name of the architectural type and its formal parameters, i.e. component types, connector types, architectural interactions, and generic parameters:

$$\begin{array}{ll}\textbf{archi\_type} & \mathit{PipeFilter}(\mathit{FilterT}; \\ & \quad \mathit{PipeT}; \\ & \quad \mathit{PipeFilterI}; \\ & \quad \text{rate } \mathit{service\_rate}_0, \\ & \quad \text{rate } \mathit{service\_rate}_1, \\ & \quad \text{rate } \mathit{service\_rate}_2, \\ & \quad \text{weight } \mathit{routing\_prob})\end{array}$$

The specification above indicates that the architectural type named *PipeFilter* relies on a single component type $\mathit{FilterT}$, a single connector type PipeT, a set of architectural interactions $\mathit{PipeFilterI}$, and four generic parameters which represent the service rates of the three processing elements and the routing probability.

archi_type	⟨name and formal parameters⟩
archi_cc_types	⟨architectural component/connector types⟩
archi_topology	⟨architectural topology: component/connector instances and attachments⟩
archi_interactions	⟨architectural interactions⟩
end	

Table 1: ÆMPA textual notation

The second part of the description defines the component and connector types which have been specified to be formal parameters of the architectural type in the first part. Each component/connector type is defined through a family of EMPA sequential terms expressing its behavior and a set of EMPA action types expressing its interactions:

$$
\begin{array}{l}
\textbf{archi\_cc\_types} \\
\quad \textbf{component\_type}\ \mathit{FilterT}(\mathit{Filter}; \\
\qquad\qquad\qquad\qquad\quad \mathit{FilterI}; \\
\qquad\qquad\qquad\qquad\quad \text{rate}\ \mu) \\
\qquad \textbf{behavior} \\
\qquad\quad \mathit{Filter} \stackrel{\Delta}{=} <\mathit{accept\_item}, *>.\mathit{Filter}' \\
\qquad\quad \mathit{Filter}' \stackrel{\Delta}{=} <\mathit{accept\_item}, *>.\mathit{Filter}'' + \\
\qquad\qquad\qquad <\mathit{serve\_item}, \mu>.\mathit{Filter} \\
\qquad\quad \mathit{Filter}'' \stackrel{\Delta}{=} <\mathit{serve\_item}, \mu>.\mathit{Filter}' \\
\qquad \textbf{interactions}\ \mathit{FilterI} = \mathit{accept\_item}, \\
\qquad\qquad\qquad\qquad\quad \mathit{serve\_item} \\
\quad \textbf{connector\_type}\ \mathit{PipeT}(\mathit{Pipe}; \\
\qquad\qquad\qquad\qquad\quad \mathit{PipeI}; \\
\qquad\qquad\qquad\qquad\quad \text{weight}\ p) \\
\qquad \textbf{behavior} \\
\qquad\quad \mathit{Pipe} \stackrel{\Delta}{=} <\mathit{accept\_item}, *>. \\
\qquad\qquad\qquad (<\mathit{forward\_item}_1, \infty_{1,p}>.\mathit{Pipe} + \\
\qquad\qquad\qquad\ <\mathit{forward\_item}_2, \infty_{1,1-p}>.\mathit{Pipe}) \\
\qquad \textbf{interactions}\ \mathit{PipeI} = \mathit{accept\_item}, \\
\qquad\qquad\qquad\qquad\quad \mathit{forward\_item}_1, \\
\qquad\qquad\qquad\qquad\quad \mathit{forward\_item}_2
\end{array}
$$

The specification above indicates that the component type *FilterT* has the behavior described by term *Filter*, which represents a processing element with capacity two and service rate μ, and the interactions described by action type set *FilterI*. Similarly, the connector type *PipeT* has the behavior described by term *Pipe* and the interactions described by action type set *PipeI*.

The third part of the description defines the topology of the architectural type in terms of instances of the previously introduced component and connector types and related attachments among their interactions:

$$
\begin{array}{ll}
\textbf{archi\_topology} & F_0 : \mathit{FilterT}(\mathit{service\_rate}_0) \\
 & F_1 : \mathit{FilterT}(\mathit{service\_rate}_1) \\
 & F_2 : \mathit{FilterT}(\mathit{service\_rate}_2) \\
 & P : \mathit{PipeT}(\mathit{routing\_prob}) \\
 & F_0.\mathit{serve\_item}\ \textbf{to}\ P.\mathit{accept\_item} \\
 & F_1.\mathit{accept\_item}\ \textbf{to}\ P.\mathit{forward\_item}_1 \\
 & F_2.\mathit{accept\_item}\ \textbf{to}\ P.\mathit{forward\_item}_2
\end{array}
$$

Note that, thanks to the possibility of expressing generic parameters in ÆMPA, it has been possible to define a single component type for the three processing elements although they have different service rates.

Finally, the fourth part defines the architectural interactions. They are interactions of the previously defined component instances:

$$
\begin{array}{ll}
\textbf{archi\_interactions} & \mathit{PipeFilterI} = F_0.\mathit{accept\_item}, \\
 & \qquad\qquad\quad F_1.\mathit{serve\_item}, \\
 & \qquad\qquad\quad F_2.\mathit{serve\_item}
\end{array}
$$

The complete architectural description is given in Table 2.

The last part above supports hierarchical modeling, because architectural interactions are to be used when plugging an architectural type in the context of the description of a larger system. This is exploited to formalize the architecture of the server of the client-server system, as shown in Table 3. As it can be noted, the behavior of *ServerT* is defined through the invocation of *PipeFilter*, where actual component/connector types are not passed (hence the formal ones of *PipeFilter* are used as such) and the architectural interactions of *PipeFilter* are renamed as *accept_request*, *generate_outcome*, and *generate_outcome*, respectively.

3.2 Graphical Notation

For the sake of ease of use, the textual notation of the previous section is accompanied by a graphical notation inspired by the flow graphs of [19], as they provide a visual help to the development of architectural descriptions.

A flow graph is a network of nodes each equipped with a set of ports; two ports of two different nodes are linked together if the two nodes can interact through those ports. Given an ÆMPA description, each component/connector type can be represented as a node (depicted as a box/rounded box) with its behavior textually reported inside the node and its interactions/architectural interactions labeling the ports (depicted as black circles/white squares). We then create an instance of a node for each instance of the component/connector type it refers to. Finally, nodes are linked together according to the specified attachments. In case of hierarchical architectural modeling, a node can contain a flow graph whose architectural interactions are linked (through dashed lines) to the ports of the node.

As an example, we show in Fig. 2 the flow graph representation of the architectural types *ClientServer* and *PipeFilter* textually described in Tables 3 and 2, respectively. From this graph it is immediately clear that the system has a hierarchical architecture, as the node for S contains a flow graph describing its structure.

3.3 Architectural Semantics

The formal semantics for ÆMPA is given by translation into EMPA by essentially exploiting the parallel composition operator. Let us consider the architectural type *PipeFilter* of Table 2. We define the semantics of a component/connector type to be the term expressing its behavior:

archi_type	$\mathit{PipeFilter}(\mathit{FilterT};$
	$\mathit{PipeT};$
	$\mathit{PipeFilterI};$
	rate $\mathit{service\_rate}_0$, rate $\mathit{service\_rate}_1$, rate $\mathit{service\_rate}_2$, weight $\mathit{routing\_prob})$
archi_cc_types	
component_type	$\mathit{FilterT}(\mathit{Filter};$
	$\mathit{FilterI};$
	rate $\mu)$
behavior	$\mathit{Filter} \triangleq <\mathit{accept\_item}, *>.\mathit{Filter}'$
	$\mathit{Filter}' \triangleq <\mathit{accept\_item}, *>.\mathit{Filter}'' +$
	$<\mathit{serve\_item}, \mu>.\mathit{Filter}$
	$\mathit{Filter}'' \triangleq <\mathit{serve\_item}, \mu>.\mathit{Filter}'$
interactions	$\mathit{FilterI} = \mathit{accept\_item}, \mathit{serve\_item}$
connector_type	$\mathit{PipeT}(\mathit{Pipe};$
	$\mathit{PipeI};$
	weight $p)$
behavior	$\mathit{Pipe} \triangleq <\mathit{accept\_item}, *>.(<\mathit{forward\_item}_1, \infty_{1,p}>.\mathit{Pipe} +$
	$<\mathit{forward\_item}_2, \infty_{1,1-p}>.\mathit{Pipe})$
interactions	$\mathit{PipeI} = \mathit{accept\_item}, \mathit{forward\_item}_1, \mathit{forward\_item}_2$
archi_topology	
	$F_0 : \mathit{FilterT}(\mathit{service\_rate}_0)$
	$F_1 : \mathit{FilterT}(\mathit{service\_rate}_1)$
	$F_2 : \mathit{FilterT}(\mathit{service\_rate}_2)$
	$P : \mathit{PipeT}(\mathit{routing\_prob})$
	$F_0.\mathit{serve\_item}$ **to** $P.\mathit{accept\_item}$
	$F_1.\mathit{accept\_item}$ **to** $P.\mathit{forward\_item}_1$
	$F_2.\mathit{accept\_item}$ **to** $P.\mathit{forward\_item}_2$
archi_interactions	
	$\mathit{PipeFilterI} = F_0.\mathit{accept\_item}, F_1.\mathit{serve\_item}, F_2.\mathit{serve\_item}$
end	

Table 2: ÆMPA description of architectural type *PipeFilter*

$$\begin{aligned}[\![\mathit{FilterT}]\!] &= \mathit{Filter}\\ [\![\mathit{PipeT}]\!] &= \mathit{Pipe}\end{aligned}$$

Likewise, we define the semantics of a component/connector instance to be the semantics of the related type:

$$\begin{aligned}[\![F_0]\!] &= \mathit{Filter}\\ [\![F_1]\!] &= \mathit{Filter}\\ [\![F_2]\!] &= \mathit{Filter}\\ [\![P]\!] &= \mathit{Pipe}\end{aligned}$$

The semantics of the whole architectural type is then obtained by composing in parallel the semantics of the component/connector instances according to the specified attachments:

$$\begin{aligned}[\![\mathit{PipeFilter}]\!] = \; & [\![F_0]\!][\mathit{serve\_item} \mapsto a] \parallel_{\{a\}}\\ & [\![P]\!][\mathit{accept\_item} \mapsto a]\\ & [\mathit{forward\_item}_1 \mapsto a_1,\\ & \mathit{forward\_item}_2 \mapsto a_2] \parallel_{\{a_1,a_2\}}\\ & ([\![F_1]\!][\mathit{accept\_item} \mapsto a_1] \parallel_{\emptyset}\\ & [\![F_2]\!][\mathit{accept\_item} \mapsto a_2])\end{aligned}$$

The use of the functional relabeling operator is necessary to make instances interact. As an example, F_0 and P must interact via *serve_item* and *accept_item* which are different from each other. Since the parallel composition operator allows actions to synchronize only if they have the same type, in $[\![\mathit{PipeFilter}]\!]$ above *serve_item* and *accept_item* are relabeled to the same action type a (occurring neither in $[\![F_0]\!]$ nor in $[\![P]\!]$), then a synchronization on a is forced by means of operator $\parallel_{\{a\}}$.

Likewise, the semantics of architectural type *ClientServer* of Table 3 is given by:

$$\begin{aligned}[\![\mathit{ClientServer}]\!] = \; & [\![C]\!][\mathit{generate\_request} \mapsto b,\\ & \mathit{accept\_outcome} \mapsto c] \parallel_{\{b,c\}}\\ & ([\![N_r]\!][\mathit{receive} \mapsto b,\\ & \mathit{forward} \mapsto c] \parallel_{\emptyset}\\ & [\![N_o]\!][\mathit{forward} \mapsto d,\\ & \mathit{receive} \mapsto e]) \parallel_{\{d,e\}}\\ & [\![S]\!][\mathit{accept\_request} \mapsto d,\\ & \mathit{generate\_outcome} \mapsto e]\end{aligned}$$

where:

$$\begin{aligned}[\![S]\!] = \; & [\![\mathit{PipeFilter}]\!][F_0.\mathit{accept\_item} \mapsto \mathit{accept\_request},\\ & F_1.\mathit{serve\_item} \mapsto \mathit{generate\_outcome},\\ & F_2.\mathit{serve\_item} \mapsto \mathit{generate\_outcome}]\end{aligned}$$

i.e. a suitable functional relabeling is applied to $[\![\mathit{PipeFilter}]\!]$ to take its actual architectural interactions into account.

3.4 Architectural Compatibility and Conformity

From the process algebra perspective, creating an ADL can be viewed as an attempt to force the designer to model systems in a more controled way, which in particular elucidates the basic architectural concepts of component and connector and hopefully enhances the usability of process algebras.

archi_type	$\mathit{ClientServer}(\mathit{ClientT}, \mathit{ServerT};$
	$\mathit{NetworkT};$
	$;$
	rate $\mathit{gen\_rate}$,
	rate $\mathit{service\_rate}_0$, rate $\mathit{service\_rate}_1$, rate $\mathit{service\_rate}_2$, weight $\mathit{routing\_prob}$,
	rate $\mathit{prop\_rate})$
archi_cc_types	
component_type	$\mathit{ClientT}(\mathit{Client};$
	$\mathit{ClientI};$
	rate $\lambda)$
behavior	$\mathit{Client} \triangleq <\mathit{generate\_request}, \lambda>.<\mathit{accept\_outcome}, *>.\mathit{Client}$
interactions	$\mathit{ClientI} = \mathit{generate\_request}, \mathit{accept\_outcome}$
component_type	$\mathit{ServerT}(\mathit{Server};$
	$\mathit{ServerI};$
	rate μ_0, rate μ_1, rate μ_2, weight $p)$
behavior	$\mathit{Server} \triangleq \mathit{PipeFilter}(;$
	$;$
	$\mathit{accept\_request}, \mathit{generate\_outcome}, \mathit{generate\_outcome};$
	$\mu_0, \mu_1, \mu_2, p)$
interactions	$\mathit{ServerI} = \mathit{accept\_request}, \mathit{generate\_outcome}$
connector_type	$\mathit{NetworkT}(\mathit{Network};$
	$\mathit{NetworkI};$
	rate $\delta)$
behavior	$\mathit{Network} \triangleq <\mathit{receive}, *>.<\mathit{forward}, \delta>.\mathit{Network}$
interactions	$\mathit{NetworkI} = \mathit{receive}, \mathit{forward}$
archi_topology	
	$C : \mathit{ClientT}(\mathit{gen\_rate})$
	$S : \mathit{ServerT}(\mathit{service\_rate}_0, \mathit{service\_rate}_1, \mathit{service\_rate}_2, \mathit{routing\_prob})$
	$N_r : \mathit{NetworkT}(\mathit{prop\_rate})$
	$N_o : \mathit{NetworkT}(\mathit{prop\_rate})$
	$C.\mathit{generate\_request}$ **to** $N_r.\mathit{receive}$
	$C.\mathit{accept\_outcome}$ **to** $N_o.\mathit{forward}$
	$S.\mathit{accept\_request}$ **to** $N_r.\mathit{forward}$
	$S.\mathit{generate\_outcome}$ **to** $N_o.\mathit{receive}$
archi_interactions	
end	

Table 3: ÆMPA description of architectural type *ClientServer*

However, this syntactic sugar alone is not enough to create a useful ADL. It must be accompanied by suitable techniques to verify the well formedness of architectural descriptions, such as the architectural compatibility and conformity checking we are going to explain.

The purpose of the former check is to ensure that an architectural type is well connected, in the sense that every pair composed of a component instance and a connector instance attached to each other interact in a proper way. This is formalized by requiring that the functional behavior of the two instances, when projected on the interactions involved in the related attachments, is the same. The latter check, instead, aims at guaranteeing that the actual parameters are consistent with the formal ones in case of architectural type invocation. This is formalized by requiring that the actual parameters do not alter the functional semantics of the architectural type w.r.t. component and connector interactions, i.e. the functional behavior of the architectural type when projected on such interactions. Technically, both checks are carried out by means of the weak bisimulation equivalence [19], a purely functional equivalence whose major feature is its ability to reason about the functional behavior of process terms when projected on certain actions, i.e. when abstracting from τ actions. The functional abstraction operator is exploited to derive projected behaviors.

Let us denote by $\approx$ the weak bisimulation equivalence. Given a set of attachments involving a given component instance and a given connector instance, we say that the component instance and the connector instance are compatible if their behaviors, when projected on the interactions mentioned in the attachments, are the same w.r.t. $\approx$. We say that an architectural type is well connected, i.e. meets architectural compatibility, if all of its component and connector instances related by some attachment are compatible.

The architectural type *PipeFilter* of Table 2 meets architectural compatibility because:

$$\llbracket F_0 \rrbracket / (\mathit{AType} - \{\mathit{serve\_item}, \tau\})[\mathit{serve\_item} \mapsto a] \approx \\ \llbracket P \rrbracket / (\mathit{AType} - \{\mathit{accept\_item}, \tau\})[\mathit{accept\_item} \mapsto a]$$

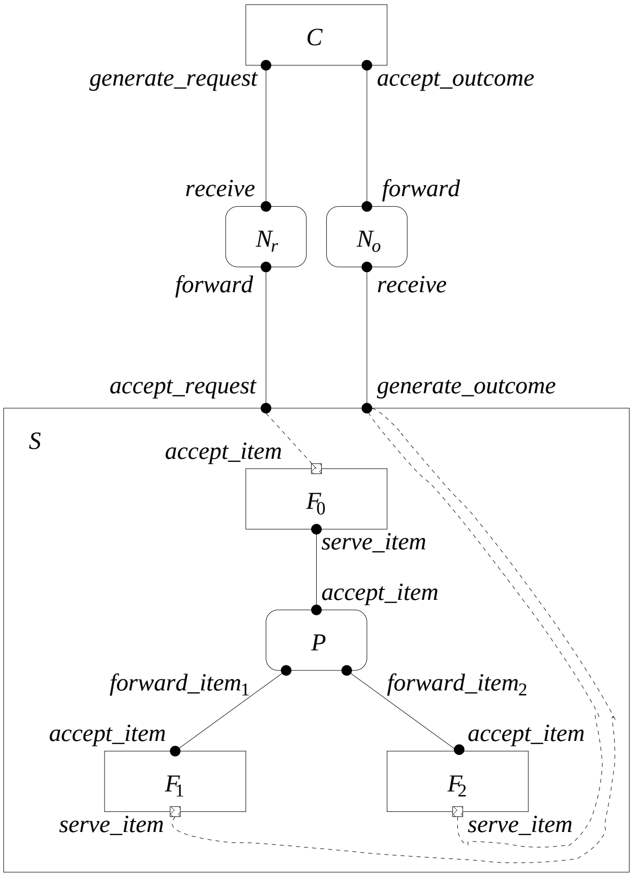


Figure 2: Flow graph representations of the architectures *ClientServer* and *PipeFilter*

$$
\begin{array}{l}
[\![F_1]\!]/(AType - \{accept\_item, \tau\})[accept\_item \mapsto a_1] \approx \\
\quad [\![P]\!]/(AType - \{forward\_item_1, \tau\})[forward\_item_1 \mapsto a_1] \\
[\![F_2]\!]/(AType - \{accept\_item, \tau\})[accept\_item \mapsto a_2] \approx \\
\quad [\![P]\!]/(AType - \{forward\_item_2, \tau\})[forward\_item_2 \mapsto a_2]
\end{array}
$$

Likewise, architectural type *ClientServer* of Table 3 meets architectural compatibility. Note that, analogously to the definition of the semantics of an architectural type, the use of the functional relabeling operator is necessary to make $\approx$ applicable.

As far as architectural conformity is concerned, given an invocation of an architectural type we have to make sure that its functional semantics, intended as the projection of its functional behavior on its component/connector instance interactions, is not altered. If this is the case, the architectural type and the architecture resulting from its invocation possess the same interactional properties, i.e. the same functional properties making assertions only about actions which are interactions. The related check consists of verifying that the behavior of each actual component/connector type and the behavior of the corresponding formal component/connector type, when projected on the related interactions, are the same w.r.t. $\approx$. In the case of the invocation of *PipeFilter* in Table 3, architectural compatibility is trivially satisfied. The interested reader is referred to [4] for a more complex example of invocation.

3.5 Functional Verification

As an example of functional verification that can be conducted on a formal description in ÆMPA, we show that *ClientServer* is deadlock free. This has been accomplished by applying model checking to $[\![ClientServer]\!]$. More precisely, the presence of deadlock can be formalized through the modal logic formula

$$\min x = [-]\mathrm{ff} \vee \langle - \rangle x$$

which is satisfied by those states of a state transition graph which have no outgoing transitions (left hand operand of the disjunction) or can reach such states (right hand operand of the disjunction). Since by means of TwoTowers we have established that the formula above is not satisfied, we can conclude that *ClientServer* is deadlock free.

3.6 Performance Evaluation

As an example of performance evaluation, we have computed with TwoTowers the system throughput (i.e. the mean number of requests served per unit of time) and the utilization of each of the three processing elements for the architecture $ClientServer(;;;5,1,3,3.5,4.5,0.5)$. Such performance measures are defined as weighted sums of the steady state probabilities of the Markov chain associated with architecture $[\![ClientServer(;;;5,1,3,3.5,4.5,0.5)]\!]$, where the weights are rewards attached to actions [3]. For the system throughput, actions $F_1.serve\_item$ and $F_2.serve\_item$ are given rewards 3.5 and 4.5, respectively. In this way, all the states enabling action $F_1.serve\_item$ ($F_2.serve\_item$)

are given a weight equal to the rate of that action, while all the other states are given weight zero. In the case of the utilization of the upstream processing element, instead, action $F_0.serve\_item$ is given reward 1. By so doing, the utilization is correctly computed as the sum of the steady state probabilities of all and only the states enabling action $F_0.serve\_item$, i.e. the states in which the processing element is used. The utilization of the other two processing elements is specified similarly. The results obtained by solving the Markov chain above and computing the suitable weighted sums with TwoTowers show that the system throughput is 0.35877 and the utilizations of the three processing elements are 0.11959, 0.0512528, and 0.0398633, respectively.

We observe that the configuration of the client-server system we are considering is extremely simple. In particular, it would be interesting to see how the performance figures above vary when increasing the number of clients. To model this scenario, one may think of modifying the architectural type *ClientServer* by introducing several instances of *ClientT*. However, this would result in an exponential growth of the state space as the number of clients (composed in parallel) increases, which would make the obtainment of performance results quite time consuming.

Fortunately, we can overcome this drawback, hence we can scale w.r.t. the number of clients, by exploiting the Markovian bisimulation equivalence defined for EMPA. In fact, it can be shown that the parallel composition of n terms *Client* is Markovian bisimulation equivalent to the following family of terms:

$$\begin{array}{rcl} Client_n & \stackrel{\Delta}{=} & <generate\_request, n \cdot \lambda>.Client_{n-1} \\ Client_i & \stackrel{\Delta}{=} & <generate\_request, i \cdot \lambda>.Client_{i-1} + \\ & & <accept\_outcome, *>.Client_{i+1}, \ \ 0 < i < n \\ Client_0 & \stackrel{\Delta}{=} & <accept\_outcome, *>.Client_1 \end{array}$$

which give rise to a reduced state space. To understand the reason of the reduction, consider the parallel composition of n terms *Client*. Its initial state has n outgoing transitions labeled with $<generate\_request, \lambda>$ towards n different states. The initial state of $Client_n$, instead, has just one outgoing transition labeled with $<generate\_request, n \cdot \lambda>$ towards state $Client_{n-1}$. The two initial states are equivalent according to the Markovian bisimulation equivalence because both can execute only actions of the same type (*generate_request*) at the same aggregated rate $(n \cdot \lambda)$ thereby reaching equivalent states. The reason why such target states are equivalent is that $\|_\emptyset$ is associative and commutative w.r.t. the Markovian bisimulation equivalence.

Since the Markovian bisimulation equivalence is a congruence, i.e. it is substitutive, w.r.t. the operators of a sublanguage of EMPA including the semantics of the two alternative architectural descriptions under consideration, the semantics (hence the properties) of the whole architectural description does not change if we use $Client_n$ instead of the parallel composition of n terms *Client*. The results of the analysis for n varying from 10 to 50 are shown in Table 4, where the second (third) column indicates the number of states (transitions) of the integrated interleaving semantic model underlying the architectural description, the fourth column indicates the system throughput, and the three trailing columns indicate the utilization of the three processing elements. As it can be noted, the state space growth is under control, thus allowing for scalability.

4. CONCLUSION

In this paper we have considered the problem of describing and analyzing functional and performance aspects of software architectures. As a solution to the problem, we have proposed the adoption of ÆMPA, a WRIGHT like ADL building on the stochastically timed process algebra EMPA, and we have illustrated its features. We observe that the choice of EMPA is inessential w.r.t. the considered problem; any stochastically timed process algebra could have been adopted.

Related Work. There are three kinds of related research: process algebras, process algebra based ADLs, and performance evaluation of software architectures.

From the process algebra perspective, ÆMPA introduces a more clear and controled way of modeling systems w.r.t. EMPA. The reason is that the user is forced to reason in terms of the basic architectural concepts of component and connector in a hierarchical way, and to model them the user can exploit only the dynamic operators of the algebra, which allows the user to ignore the details about the more complex static operators. Additionally, the presence of generic parameters enhances the possibility of algebraic term reuse in the descriptions, as illustrated by component instances F_0, F_1, and F_2 in Table 2. In essence, we can say that ÆMPA is a more usable version of EMPA which supports term reuse, hierarchical modeling, and static checks like architectural compatibility and conformity, and may result in longer descriptions for trivial systems.

As far as process algebra based ADLs are concerned, ÆMPA is clearly inspired by WRIGHT. However, there are some considerable differences among the two ADLs, which make ÆMPA an extension and a simplification of WRIGHT. First, ÆMPA permits to model types of architectures instead of single architectures in a hierarchical way, and introduces a related check (architectural conformity) to correctly generate the instances of an architectural type. Second, interactions are expressed in ÆMPA just by means of action types instead of algebraic terms, which reduces the redundance in the description of component and connector types and eases the the definition of the language semantics and the architectural compatibility and conformity. Third, ÆMPA is equipped with a graphical notation which provides a useful visual help during the modeling process. Fourth, ÆMPA addresses also performance related issues.

Concerning the performance evaluation of software architectures, also in [2, 14] a formal approach was taken like in ÆMPA. Such an approach consists of describing software architectures with the CHAM [6] (as proposed in [15]) and deriving a queueing network model [16] out of it which is used to extract performance measures. This approach differs from ours in that their modeling formalism is not integrated, as performance characteristics are described separately only upon generation of the queueing network model, and does not fully elucidate architectural concepts nor allows for ar-

clients	states	transitions	throughput	utilization$_0$	utilization$_1$	utilization$_2$
10	1067	2430	0.635987	0.224099	0.088165	0.072757
20	2667	6150	0.699606	0.242998	0.096040	0.080769
30	4267	9870	0.744251	0.265766	0.101318	0.086586
40	5867	13590	0.794978	0.292759	0.107302	0.093205
50	7467	17310	0.858178	0.319252	0.114846	0.101381

Table 4: Performance analysis results

chitectural well formedness checks. Their approach has the advantage that queueing network models achieve a favorable balance between accuracy and efficiency when solving them to calculate performance measures. However, the computationally expensive operation is the derivation of the performance model from an architectural description. It turns out that deriving a queueing network model out of a CHAM description is as costly as deriving a (more general) Markov chain out of an ÆMPA description of the same software system. Moreover, as shown in [12], in the case of algebraic specifications it is sometimes possible to avoid the generation of the whole Markov chain. We also observe that with our approach we can deal with generally distributed activity durations by exploiting value passing related features and simulative analysis.

The use of queueing network models is advocated also in [25, 22], where the authors concentrate only on some (graphically introduced) architectural styles for which they provide suitable transformations to allow for performance evaluation. Simulation is used instead in [9] to compare some architectural styles w.r.t. a selection of performance indicators. Finally, other proposals are based on software performance engineering techniques. In [27] the 4 + 1 view model of [17] is exploited; use case scenarios are employed to provide graphical architectural descriptions from which execution graphs can be extracted and analyzed to derive mean, best and worst case response times. A similar approach is taken in [18], where different architectural views are considered.

Future Work. To make ÆMPA usable in practice, we plan to provide tool support by exploiting TwoTowers. The idea is to allow the user to specify software architectures with the graphical or the textual notation, automate the translation of ÆMPA descriptions into the corresponding EMPA terms, implement routines for architectural compatibility and conformity checking, and invoke (as we have done for our running example) the routines of TwoTowers to conduct functional verification and performance evaluation. Having this tool available will help us to conduct medium and large size case studies to assess the adequacy of our approach. In particular, we shall consider scalability related issues (our running example has shown that ÆMPA has the potential for scalability) and the possibility of extracting diagnostic information at the architectural level in case of poor performance.

We are also interested in investigating the integration of ÆMPA in the software life cycle to broaden the usability of our approach. More precisely, on the one hand we would like to study whether standard graphical notations used in software engineering such as UML [20] can be mapped to ÆMPA in order to be able to conduct some kind of analysis. On the other hand, we plan to address the feasibility of code generation out of ÆMPA descriptions. The purpose is to ensure that correctness and performance characteristics investigated on an ÆMPA description are achieved by the related implementation. A study on this subject can be found in [8], where a translation is given from specifications in the stochastically timed process algebra PEPA [11] to Ada programs.

5. REFERENCES

[1] R. Allen, D. Garlan, *"A Formal Basis for Architectural Connection"*, in ACM Trans. on Software Engineering and Methodology 6:312-249, 1997

[2] S. Balsamo, P. Inverardi, C. Mangano, *"An Approach to Performance Evaluation of Software Architectures"*, in Proc. of the *1st Int. Workshop on Software and Performance (WOSP '98)*, ACM Press, pp. 178-190, Santa Fe (NM), 1998

[3] M. Bernardo, *"Theory and Application of Extended Markovian Process Algebra"*, Ph.D. Thesis, University of Bologna (Italy), 1999 (`http://www.di.unito.it/~bernardo/`)

[4] M. Bernardo, P. Ciancarini, L. Donatiello, *"On the Formalization of Architectural Types with Process Algebras"*, to appear in Proc. of the *8th ACM Int. Symp. on the Foundations of Software Engineering (FSE-8)*, ACM Press, San Diego (CA), 2000

[5] M. Bernardo, W.R. Cleaveland, *"A Theory of Testing for Markovian Processes"*, to appear in Proc. of the *11th Int. Conf. on Concurrency Theory (CONCUR 2000)*, LNCS, State College (PA), 2000

[6] G. Berry, G. Boudol, *"The Chemical Abstract Machine"*, in Theoretical Computer Science 96:217-248, 1992

[7] W.R. Cleaveland, S. Sims, *"The NCSU Concurrency Workbench"*, in Proc. of the *8th Int. Conf. on Computer Aided Verification (CAV '96)*, LNCS 1102:394-397, New Brunswick (NJ), 1996

[8] S. Gilmore, J. Hillston, R. Holton, *"From SPA Models to Programs"*, in Proc. of the *4th Int. Workshop on Process Algebras and Performance Modelling (PAPM '96)*, CLUT, pp. 179-197, Torino (Italy), 1996

[9] H. Grahn, J. Bosch, *"Some Initial Performance Characteristics of Three Architectural Styles"*, in Proc. of the *1st Int. Workshop on Software and Performance (WOSP '98)*, ACM Press, pp. 197-198, Santa Fe (NM), 1998

[10] H. Hermanns, *"Interactive Markov Chains"*, Ph.D. Thesis, University of Erlangen-Nürnberg (Germany), 1998

[11] J. Hillston, *"A Compositional Approach to Performance Modelling"*, Cambridge University Press, 1996

[12] J. Hillston, *"Exploiting Structure in Solution: Decomposing Composed Models"*, in Proc. of the *6th Int. Workshop on Process Algebras and Performance Modelling (PAPM '98)*, pp. 1-15, Nice (France), 1998

[13] C.A.R. Hoare, *"Communicating Sequential Processes"*, Prentice Hall, 1985

[14] P. Inverardi, C. Mangano, S. Balsamo, *"Performance Evaluation of a Software Architecture: A Case Study"*, in Proc. of the *8th Int. Workshop on Software Specification and Design (IWSSD-8)*, Paderborn (Germany), 1998

[15] P. Inverardi, A.L. Wolf, *"Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model"*, in IEEE Trans. on Software Engineering 21:373-386, 1995

[16] L. Kleinrock, *"Queueing Systems"*, John Wiley & Sons, 1975

[17] P.B. Kruchten, *"The* $4+1$ *View Model of Architecture"*, in IEEE Software 12(6):42-50, 1995

[18] C.-H. Lung, A. Jalnapurkar, A. El-Rayess, *"Performance-Oriented Software Architecture Engineering: An Experience Report"*, in Proc. of the *1st Int. Workshop on Software and Performance (WOSP '98)*, ACM Press, pp. 191-196, Santa Fe (NM), 1998

[19] R. Milner, *"Communication and Concurrency"*, Prentice Hall, 1989

[20] Object Management Group, *"Unified Modeling Language Specification"*, Tech. Rep. available at uml.shl.com, 1998

[21] D.E. Perry, A.L. Wolf, *"Foundations for the Study of Software Architecture"*, in ACM SIGSOFT Software Engineering Notes 17:40-52, 1992

[22] D.C. Petriu, X. Wang, *"Deriving Software Performance Models from Architectural Patterns by Graph Transformations"*, in Proc. of the *6th Int. Workshop on Theory and Application of Graph Transformation (TAGT '98)*, pp. 340-347, Paderborn (Germany), 1998

[23] M. Shaw, D. Garlan, *"Software Architecture: Perspectives on an Emerging Discpline"*, Prentice Hall, 1996

[24] C.U. Smith, *"Performance Engineering of Software Systems"*, Addison-Wesley, 1990

[25] B. Spitznagel, D. Garlan, *"Architecture-Based Performance Analysis"*, in Proc. of the *10th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE '98)*, 1998

[26] W.J. Stewart, *"Introduction to the Numerical Solution of Markov Chains"*, Princeton University Press, 1994

[27] L.G. Williams, C.U. Smith, *"Performance Evaluation of Software Architectures"*, in Proc. of the *1st Int. Workshop on Software and Performance (WOSP '98)*, ACM Press, pp. 164-177, Santa Fe (NM), 1998