TwoTowers 3.0: Enhancing Usability

Marco Bernardo
Università di Urbino "Carlo Bo"
Istituto di Scienze e Tecnologie dell'Informazione
Piazza della Repubblica 13, 61029 Urbino, Italy
bernardo@sti.uniurb.it

Abstract

TwoTowers is a software tool for the functional verification and performance evaluation of computer, communication and software systems represented through stochastic process algebra (SPA). In this paper we describe a novel version (3.0), in which the modeling language is no longer a pure SPA, but a SPA-based architectural description language called Æmilia. We show that TwoTowers 3.0 improves on the previous version in terms of usability, because Æmilia hides most of the technicalities of SPA, and also in terms of efficiency, bacause a new algorithm for state space generation has been implemented.

1 Introduction

In the last ten years the formalism of stochastic process algebra (SPA; see, e.g., [9, 11, 10, 4]) has emerged in the field of system performance evaluation as a novel modeling paradigm. A SPA is a formal description technique for complex systems, especially those with communicating, concurrently executing components, with the following key characteristics: (i) compositional modeling through a small number of constructs for building larger systems up from smaller ones; (ii) structural operational semantics that formally defines for each process term the state transition graph that it stands for; (iii) syntax-oriented and semantics-oriented behavioral reasoning via equivalences that capture the notion of same behavior, possibly abstracting from unnecessary details. A few tools based on SPA have also been developed [12, 8, 2], which differ for the expressiveness of their modeling languages and for the supported analysis techniques.

A frequently recurrent criticism to SPA, especially from practitioners, is that it is difficult to learn and use. Addressing this usability issue is crucial for a wider acceptance of SPA, and requires a deep understanding of the reasons why it is not seen as friendly as its supporters claim. We believe that one of the main obstacles to the ease of use of SPA is that, although SPA provides a compositional modeling facility via algebraic operators, this facility has not been exploited in the most appropriate manner. By this we mean that SPA has the potential to fully support a safe component-oriented way of modeling systems, in which the user can reason in terms of components and their interactions and can identify components that result in mismatches when assembled together, but this is buried under the technicalities of the algebraic operators and the synchronization discipline.

As observed in [5], an enhancement of the usability of SPA can be achieved by transforming it into an architectural description language [13], which elucidates the basic notions of components and connectors while making the SPA technicalities as transparent as possible. In this paper we describe our effort to implement a new version (3.0) of the software tool TwoTowers, originally based on pure SPA [2], in accordance with the guidelines of [5]. In Sect. 2 we illustrate the architecture of TwoTowers 3.0. In Sect. 3 we present its new modeling language Æmilia [5]. In Sect. 4 and 5 we report some considerations and experiences about the improvement in terms of usability and efficiency. Finally, in Sect. 6 we provide some remarks about future extensions.

2 Overview of TwoTowers 3.0

The architecture of TwoTowers 3.0, which is reported in Fig. 1, is slightly different from that of the previous version of the tool [2], in order to better reflect the supported analysis techniques (model checking, equivalence checking, and performance evaluation) rather than the analyzable properties (integrated, functional, and performance).

TwoTowers 3.0 is equipped with a simple graphical user interface through which the user can invoke the analysis routines. Each routine needs input files of certain types and writes its results onto files of other types. The graphical user interface takes care of the integrated management of

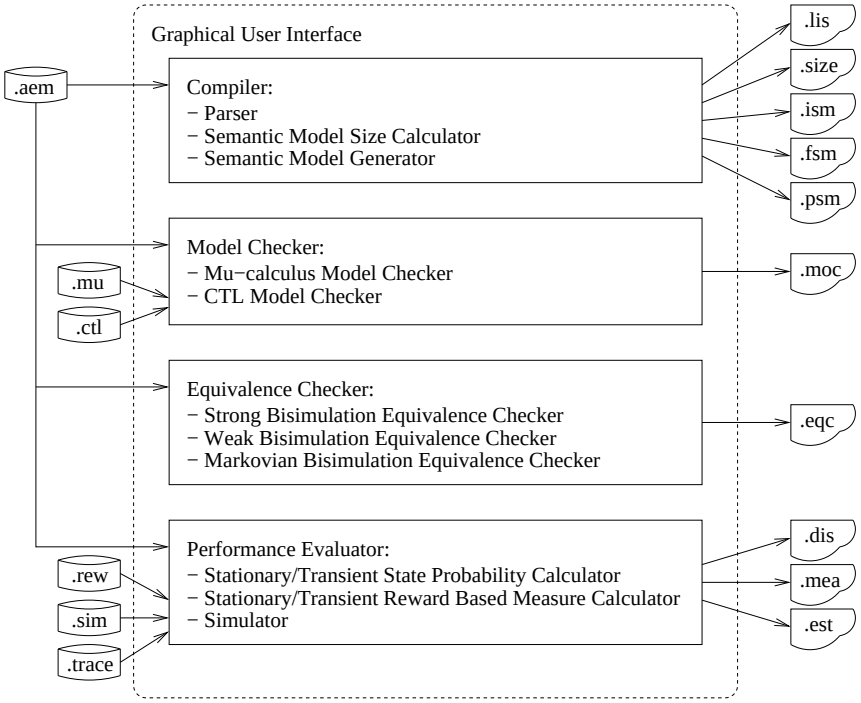

Figure 1. Architecture of TwoTowers 3.0

the various file types needed by the different routines.

The compiler is in charge of parsing system specifications stored in .aem files and signalling possible lexical, syntax and static semantic errors through a .lis file. If a specification is correct, the compiler can generate its integrated, functional or performance semantic model, which is written to a .ism, .fsm or .psm file, respectively. As a faster option that does not require printing the state space onto a file, the compiler can show only the size – in terms of number of states and transitions – of the semantic model, which is written to a .size file.

The model checker verifies whether a set of μ-calculus formulas (stored in a .mu file) or CTL formulas (stored in a .ctl file) are satisfied by a correct specification. The check is executed by invoking the CWB-NC 1.2 tool [7]. The outcome of the check is written to a .moc file.

The equivalence checker verifies whether two correct specifications are equivalent according to one of several different notions of equivalence. Except for the case of the Markovian bisimulation equivalence, the check is executed by invoking CWB-NC 1.2. The result of the check, together with some diagnostic information in case of non-equivalence, is written to a .eqc file.

Finally, the performance evaluator computes the performance characteristics of correct and performance closed specifications. First, it can calculate the stationary/transient state probability distribution of the performance semantic model of a specification, where the model is either a continuous-time or a discrete-time Markov chain. The distribution is written to a .dis file. Second, the performance evaluator can calculate as described in [4] a set of instant-of-time, stationary/transient performance measures specified through state and transition rewards stored in a .rew file. The values of the measures are written to a .mea file. In the stationary case the Gaussian elimination method and an adaptive variant of the symmetric SOR method are available, while in the transient case the method of uniformization is available. Third, the performance evaluator can estimate via discrete event simulation the mean, variance or distribution of a set of performance measures specified through an extension of state and transition rewards stored in a .sim file. The simulation is based on the method of independent replications and can be trace-driven, in which case the traces are stored in .trace files. The outcome of the simulation, which can be applied also to specifications whose underlying performance semantic model is not Markovian, is written to a .est file.

3 The New Modeling Language: Æmilia

The main innovation of TwoTowers 3.0 resides in its modeling language. Although SPA supports the compositional modeling of concurrent and distributed systems via algebraic operators, their technicalities often obfuscate the way in which the systems are modeled. For instance, if a system

is made out of several components, with SPA the system is simply described as the parallel composition of several subterms, each representing the behavior of a single component, with suitable synchronization sets defining the component interactions. It would be more natural to separately specify the behavior of each type of component, to indicate the actions through which each component type interacts with the others, to declare the instances of each component type that form the system, and to specify the way in which the interacting actions are attached to each other in order to make the component instances interact.

This component-oriented view is now embodied in the new modeling language of TwoTowers 3.0, the architectural description language Æmilia [5], which is based on the modeling language of TwoTowers 2.0, i.e. EMPA_{gr} [4]. A description in Æmilia represents an architectural type, which is a family of software systems sharing certain constraints on the component observable behavior as well as on the architectural topology. The description of an architectural type starts with its name and its formal parameters, which can represent variables as well as exponential rates, priorities, and weights for EMPA_{gr} actions. Each architectural type is defined through its architectural element types (AETs) and its architectural topology. An AET, whose description starts with its name and its formal parameters, is defined through its behavior, specified as a list of sequential EMPA_{gr} defining equations, and its input and output interactions, specified as a set of EMPA_{gr} action types occurring in the behavior that act as interfaces for the AET. The architectural topology is specified through the declaration of a set of architectural element instances (AEIs) representing the system components, a set of architectural interactions given by some interactions of the AEIs that act as interfaces for the whole architectural type, and a set of directed architectural attachments among the interactions of the AEIs that make the AEIs communicate with each other. Every attachment must go from an output interaction to an input interaction of two different AEIs. Given that every interaction is declared to be a uni-interaction, an and-interaction, or an or-interaction, the only legal attachments are those between two uni-interactions, an and-interaction and a uni-interaction, and an or-interaction and a uni-interaction. An and-interaction and an or-interaction can be attached to several uni-interactions. In the case of execution of an and-interaction (resp. an or-interaction), it synchronizes with all (resp. only one of) the uni-interactions attached to it. The whole behavior of an Æmilia description is given by a family of EMPA_{gr} defining equations transparently obtained by composing in parallel the behaviors of the declared AEIs according to the specified attachments. From the overall behavior, integrated, functional and performance semantic models can automatically be derived, which can undergo to the model checking, equivalence checking and performance evaluation techniques implemented in TwoTowers 3.0.

We now illustrate Æmilia by means of a simple example concerning a client-server system. The system is composed of n of clients, each requesting service with rate λ_i for $1 \leq i \leq n$. The server is internally organized as a pipe-filter system comprising an upstream filter, a splitting pipe, two middle filters (referred to as left filter and right filter), a joining pipe, and a downstream filter. Each filter $j \in \{\mathtt{u}, \mathtt{l}, \mathtt{r}, \mathtt{d}\}$ is a service center of capacity two with service rate σ_j. Each item processed by the upstream filter is instantaneously forwarded by the splitting pipe to one of the two middle filters. If both have free positions in their buffers, the choice is resolved according to a routing probability $\mathtt{p}_{\mathtt{routing}}$. All the items processed by the middle filters are then instantaneously propagated by the joining pipe to the downstream filter. Here is the header of the Æmilia description for the considered client-server system with $n = 2$:

```
ARCHI_TYPE Client_Server_Type(rate λ1, λ2
                              rate σu, σl, σr, σd,
                              weight p_routing)
```

Then the Æmilia description specifies the behavior and the interactions of the client component type:

```
ARCHI_ELEM_TYPES
 ELEM_TYPE Client_Type(rate λ)
  BEHAVIOR
   Client(void; void) =
    <send_request, λ>.<receive_outcome, *>.Client()
  INPUT_INTERACTIONS UNI receive_outcome
  OUTPUT_INTERACTIONS UNI send_request
```

Every client submits requests at a rate λ, waits for receiving back an outcome, then repeats this behavior. The definition of `Client` has neither formal parameters nor local variables, hence the two occurrences of `void` in its header. The action with type `send_request` is exponentially timed and is declared to be an output uni-interaction, i.e. it can synchronize only with one input interaction of another AEI. The action with type `receive_outcome`, instead, is passive — its rate is unspecified — and is declared to be an input uni-interaction.

The behavior and the interactions of the filter component type are specified as follows:

```
ELEM_TYPE Filter_Type(rate σ)
 BEHAVIOR
  Filter0(void; void) =
   <accept_item, *>.Filter1();
  Filter1(void; void) =
   choice
   {
    <accept_item, *>.Filter2(),
    <serve_item, σ>.Filter0()
   };
  Filter2(void; void) =
   <serve_item, σ>.Filter1()
 INPUT_INTERACTIONS UNI accept_item
 OUTPUT_INTERACTIONS UNI serve_item
```

Initially ($\texttt{Filter}_0$), the filter waits for an item to arrive. When an item is already in the filter buffer ($\texttt{Filter}_1$), either another item arrives at the filter, or a previously arrived item finishes to be served at rate σ and is sent out. When two items are already in the filter buffer ($\texttt{Filter}_2$), no more items can be accepted until one of the two previously arrived items finishes to be served. Upstream_Filter_Type and Downstream_Filter_Type differ from Filter_Type only in the interactions. The upstream filter must accept requests from one among several clients:

INPUT_INTERACTIONS OR accept_item
OUTPUT_INTERACTIONS UNI serve_item

while the downstream filter must send outcomes to one among several clients:

INPUT_INTERACTIONS UNI accept_item
OUTPUT_INTERACTIONS OR serve_item

The behavior and the interactions of the splitting pipe component type are specified as follows:

```
ELEM_TYPE Split_Pipe_Type(weight p_routing)
 BEHAVIOR
  Split_Pipe(void; void) =
   <accept_item, *>.
    choice
    {
     <send_item_l, inf(1, p_routing)>.Split_Pipe(),
     <send_item_r, inf(1, 1 - p_routing)>.Split_Pipe()
    }
 INPUT_INTERACTIONS UNI accept_item
 OUTPUT_INTERACTIONS UNI send_item_l; send_item_r
```

The splitting pipe waits for an item, instantaneously forwards it to one of two different destinations, then repeats this behavior. The instantaneous forwarding is formalized by associating an infinite rate to the actions with type $\texttt{send\_item}_\texttt{l}$ and $\texttt{send\_item}_\texttt{r}$. If both of them can synchronize with their respective attached interactions, the choice among them is governed by the weights based on $\texttt{p}_\texttt{routing}$ associated with their infinite rates (the priority level 1 associated with the infinite rates is unimportant here). The definition of the joining pipe component type is similar to the previous one:

```
ELEM_TYPE Join_Pipe_Type()
 BEHAVIOR
  Join_Pipe(void; void) =
   <accept_item, *>.<send_item, inf>.Join_Pipe()
 INPUT_INTERACTIONS OR accept_item
 OUTPUT_INTERACTIONS UNI send_item
```

Finally, here is the architectural topology section of the Æmilia description:

```
ARCHI_TOPOLOGY
 ARCHI_ELEM_INSTANCES
  C_1 : Client_Type(λ_1);
  C_2 : Client_Type(λ_2);
  UF : Upstream_Filter_Type(σ_u);
  SP : Split_Pipe_Type(p_routing);
  LF : Filter_Type(σ_l);
  RF : Filter_Type(σ_r);
  JP : Join_Pipe_Type();
  DF : Downstream_Filter_Type(σ_d)
 ARCHI_INTERACTIONS
 ARCHI_ATTACHMENTS
  FROM C_1.send_request TO UF.accept_item;
  FROM C_2.send_request TO UF.accept_item;
  FROM UF.serve_item TO SP.accept_item;
  FROM SP.send_item_l TO LF.accept_item;
  FROM SP.send_item_r TO RF.accept_item;
  FROM LF.serve_item TO JP.accept_item;
  FROM RF.serve_item TO JP.accept_item;
  FROM JP.send_item TO DF.accept_item;
  FROM DF.serve_item TO C_1.receive_outcome;
  FROM DF.serve_item TO C_2.receive_outcome
END
```

Suppose that we are interested in measuring the server throughput and the utilization of the four filters. Then we prepare a .rew file, where the throughput is expressed by associating an instantaneous reward equal to 1.0 with every transition of a state in which action serve_item of the downstream filter is enabled:

```
MEASURE server_throughput IS
 enabled(DF.serve_item) -> trans_rew(1.0)
```

whereas the utilization of e.g. the upstream filter is expressed by associating a cumulative reward with every state in which action serve_item of that filter is enabled:

```
MEASURE UF_utilization IS
 enabled(UF.serve_item) -> state_rew(1.0)
```

The results obtained with TwoTowers 3.0 are shown in Fig. 2 and 3, where the number of clients varies between 1 and 15 and $\lambda_i = 0.1 \cdot i$, $\sigma_\texttt{u} = 5.0$, $\sigma_\texttt{l} = 6.5$, $\sigma_\texttt{r} = 7.5$, $\sigma_\texttt{d} = 8.0$, and $\texttt{p}_\texttt{routing} = 0.6$.

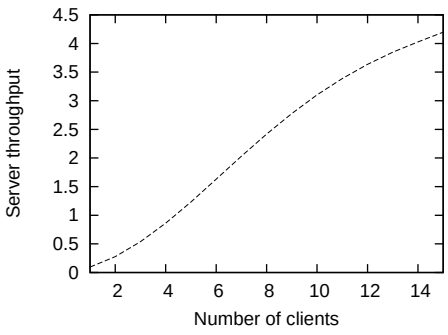

Figure 2. Server throughput

4 Degree of Usability

Let us now compare the usability of Æmilia with the usability of a pure SPA. To start with, Æmilia explicitly supports a component-oriented way of modeling, as it requires to declare the components that are in the system and the way

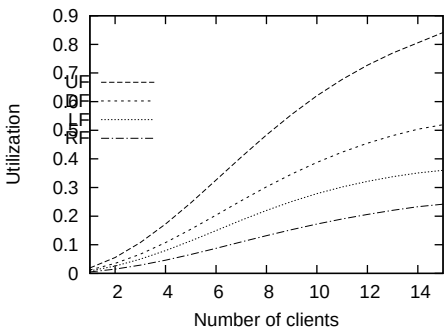

Figure 3. Filter utilization

in which they are attached to each other in order to interact. With SPA, one should define a parallel composition of the terms representing the behavior of the components, with the appropriate synchronization sets. This would result in a much more obscure description of the system topology.

The definition of every AET in Æmilia has a clear separation between the behavior and the interactions. In particular, the declaration of the interactions, together with their qualifiers (input/output, uni/and/or), makes explicit the intended use of every action type occurring in the behavior, and in particular the distinction between internal and external activities. This is not the case with SPA, where this information must be inferred from the syncronization sets associated with the parallel composition operators.

The behavior and the interactions of the components of the same type must be defined only once with Æmilia. In addition, the definition of the behavior can be parameterized with respect to exponential rates, weights, priorities, and values that can be exchanged when interacting with other components. The possibility of defining a parameterized AET only once, together with the possibility of declaring as many AEIs of that AET as needed, considerably enhances the degree of specification reuse, hence reduces the time to prepare the formal specification, with respect to SPA.

The component-oriented way of modeling supported by Æmilia facilitates the subsequent modifications of the formal descriptions, while this is not the case with SPA. For instance, in the client-server system description, if we want to add one more client, all we have to do is to declare a further instance of the client type with the appropriate rate, together with the attachments between the new instance of the client type and the upstream filter and the downstream filter.

Most of the SPA operators are hidden in Æmilia. To be more precise, only the dynamic ones (action prefix and alternative composition) can still be used within the specification of the behavior of the AETs, while the more complicated, static ones (parallel composition, hiding, and relabeling) are transparently introduced when translating an Æmilia description to its corresponding family of SPA defining equations. In addition, only the action prefix operator "." has maintained its original notation, while the infix notation "+" for the alternative composition operator has been converted in Æmilia into the more intuitive, prefixed notation `choice`.

In addition, Æmilia allows error-prone situations to be detected in its descriptions, while this is not the case with SPA. As an example, checks are implemented to avoid an inconsistent use of an action type within the behavior of an AET, i.e. the fact that an action type occurs several times with different kinds of rate (e.g., some times with exponential rates, some other times with infinite rates). As another example, there are checks to ensure that the attachments comply with the qualifiers of the involved interactions (e.g.: no attachment between two output interactions, no multiple attachments involving the same uni-interaction), and that no internal action is involved in an attachment. As a final example, a further check ensures that the declared topology results in a connected graph, so that there are no isolated groups of components.

On the experimental side Æmilia, besides being used at the University of Urbino, is also being used by some students at the University of L'Aquila. Such students, who are familiar with software engineering concepts and methodologies but not with formal methods like SPA, were previously exposed to pure SPA when using TwoTowers 2.0. It turned out that they had a lot of difficulties with the use of the parallel composition operator, mainly because of the fact that the action types that must be part of the synchronization sets depend on the order in which the terms modeling the system components are expressed. Now the students are using TwoTowers 3.0 and feel more confident about the correctness of the synchronizations they want to impose, as these are simply expressed through attachments between the right interactions and automatically checked against their qualifiers (input/output, uni/and/or). Furthermore, such students are finding very beneficial the possibility of parameterizing the definitions of the AETs with respect to exponential rates, weights, and priorities, as this reduces the amount of specification that must be written.

5 Impact on the Efficiency

The adoption of Æmilia in TwoTowers 3.0 has occasioned an improvement of the efficiency with respect to TwoTowers 2.0, because of the implementation of a new algorithm for the state space generation. Some experiments — on a PC with a 1.5 GHz Pentium IV processor, 1 Gbyte of RAM, and the Linux operating system — with the specification of several systems that we analyzed in the past have shown that we scale from hundreds of thousands of states plus transitions to millions of states plus transitions, thus gaining one

order of magnitude. As an example, we report in Table 1 a comparison between the state spaces that we have been able to generate with TwoTowers 2.0 and TwoTowers 3.0 when increasing the number n of clients in the considered client-server system.

n	states	transitions	2.0	3.0
1	7	8	√	√
2	31	56	√	√
3	108	253	√	√
4	322	903	√	√
5	859	2763	√	√
6	2104	7562	√	√
7	4812	18987	√	√
8	10405	44473	√	√
9	21447	98297	√	√
10	42413	206776	√	√
11	80862	416691	√	√
12	149211	808585	–	√
13	267320	1517165	–	√
14	466200	2761858	–	√
15	793213	4891514	–	√

Table 1. TwoTowers 2.0 vs. TwoTowers 3.0

In TwoTowers 3.0 the state space is generated in two steps. In the first step, the local state space of each AET is generated, with each local state stored in a hash table and represented by the string of the corresponding sequential EMPA_{gr} term in concise notation, i.e. with every identifier replaced by a serial number. In the second step, the global state space is generated by combining in parallel the local states spaces according to the declaration of the AEIs and of the attachments between them, with each global state stored in another hash table and represented by the vector of pointers to its constituent local states. Although in general in TwoTowers 3.0 the memory has been utilized in a more careful way, the reason for the efficiency improvement is essentially twofold. First, all the AEIs of each AET share the local state space of their AET, instead of having their own separate local state spaces. Second, every global state is represented through a vector of pointers, instead of the string of the corresponding parallel EMPA_{gr} term in concise notation.

6 Future Extensions

We would like to continue our work towards a fully enhanced usability. This requires implementing the graphical notation associated with Æmilia, the architectural checks for deadlock freedom and performance closure described in [5], and an easier way to express temporal logic formulas and reward structures. Furthermore, we would like to address the scalability issue. This has already been done for the simulation, as it generates only the needed global states and employs the symbolic technique of [3] to deal with infinite data domains. On the model checking side, we plan to interface TwoTowers 3.0 with tools supporting symbolic techniques like NuSMV 2.1 [6]. On the Markovian analysis side, we plan to implement the Æmilia-to-queueing networks translation of [1].

References

[1] S. Balsamo, M. Bernardo, and M. Simeoni, *"Performance Evaluation at the Software Architecture Level"*, in *Formal Methods for Software Architectures*, LNCS 2804:209-260, 2003.

[2] M. Bernardo, *"TwoTowers 2.0 User Manual"*, `www.sti.uniurb.it/bernardo/twotowers/`, 2002.

[3] M. Bernardo, *"Symbolic Semantic Rules for Producing Compact STGLA from Value Passing Process Descriptions"*, to appear in ACM Trans. on Computational Logic, 2003.

[4] M. Bernardo and M. Bravetti, *"Performance Measure Sensitive Congruences for Markovian Process Algebras"*, in Theoretical Computer Science 290:117-160, 2003.

[5] M. Bernardo, L. Donatiello, and P. Ciancarini, *"Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language"*, in *Performance Evaluation of Complex Systems: Techniques and Tools*, LNCS 2459:236-260, 2002.

[6] R. Cavada, A. Cimatti, E. Olivetti, M. Pistore, and M. Roveri, *"NuSMV 2.1 User Manual"*, `nusmv.irst.itc.it/`, 2002.

[7] W.R. Cleaveland, T. Li, and S. Sims, *"The Concurrency Workbench of the New Century - Version 1.2 - User's Manual"*, `www.cs.sunysb.edu/~cwb/`, 2000.

[8] S. Gilmore, *"The PEPA Workbench User Manual"*, `www.dcs.ed.ac.uk/pepa/tools.html`, 2001.

[9] N. Götz, U. Herzog, and M. Rettelbach, *"Multiprocessor and Distributed System Design: The Integration of Functional Specification and Performance Analysis Using Stochastic Process Algebras"*, in Proc. of PERFORMANCE 1993, LNCS 729:121-146, 1993.

[10] H. Hermanns, *"Interactive Markov Chains"*, LNCS 2428, 2002.

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

[12] U. Klehmet and V. Mertsiotakis, *"TIPPtool – User's Guide"*, `www7.informatik.uni-erlangen.de/tipp/tool.html`, 1998.

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