TwoEagles: A Model Transformation Tool from Architectural Descriptions to Queueing Networks

Marco Bernardo[1] Vittorio Cortellessa[2] Mirko Flamminj[2]

[1] Dipartimento di Scienze di Base e Fondamenti – Università di Urbino – Italy
[2] Dipartimento di Informatica – Università dell'Aquila – Italy

Abstract. We present the implementation of a methodology for the modeling, analysis, and comparison of software architectures based on their performance characteristics. The implementation is part of a software tool that is called TwoEagles, which extends the architecture-centric tool TwoTowers – based on the stochastic process algebraic description language Æmilia – and integrates it into Eclipse. The extension consists of a Java-coded plugin that we have called AEmilia_to_QN. This plugin transforms Æmilia descriptions into queueing network models expressed in the XML schema PMIF, which can then be rendered via the QN_Editor tool or analyzed by multiple queueing network solvers that can be invoked through the Weasel web service.

1 Introduction

The importance of an integrated view of functional and nonfunctional characteristics in the early stages of software development is by now widely recognized. This is a consequence of the awareness of the risks arising either from considering those two classes of characteristics on two different classes of system models that are not necessarily consistent with each other, or from examining nonfunctional features at later stages of the development cycle. This has resulted in the extension of numerous semi-formal and formal notations with nonfunctional attributes yielding quantitative variants of logics, automata, Petri nets, process calculi, and specification languages as well as suitable UML profiles, many of which are surveyed, e.g., in [3].

The assessment of nonfunctional characteristics is not only instrumental to enhancing the quality of software systems. As an example, a number of alternative architectural designs may be developed for a given system, each of which is functionally correct. In that case, we need to establish some criteria for deciding which architectural design is more appropriate and hence is the one to implement. As it turns out, performance requirements and constraints are certainly among the most influential factors that drive architecture-level design choices.

In order to address the various issues mentioned above, in [2] a methodology has been proposed for predicting, improving, and comparing the performance of software architectures. This methodology, called PerfSel in [1], consists of a number of phases at the end of which typical performance indices are assessed in different scenarios for the various architectural designs both at the system level and at the component level. On the basis of those indices, it can be decided to discard some designs, improve others, or select the one to be implemented.

Although the PERFSEL methodology is independent to a large extent from the notation in which architectural designs are expressed, as in [2, 1] here we focus on ÆMILIA. This is an architectural description language based on stochastic process algebra that enables functional verification via model checking or equivalence checking, as well as performance evaluation through the numerical solution of continuous-time Markov chains or discrete-event simulation.

On the analysis side, PERFSEL instead employs queueing networks [10]. A main motivation of this choice is that, in contrast to continuous-time Markov chains – which are flat performance models – queueing networks are structured performance models providing support for establishing a correspondence between their constituent elements and the components of architectural descriptions. Moreover, some families of queueing networks, like product-form queueing networks [4], are equipped with efficient solution algorithms that do not require the construction of the underlying state space when calculating typical average performance indices at system level or component level, such as response time, throughput, utilization, and queue length. Therefore, the transformation of ÆMILIA models into queueing networks enables a wider set of performance analysis techniques on the same architectural model.

Starting from a number of alternative architectural designs – which we assume to be functionally correct – of a software system to be implemented, PERFSEL requires the designer to formalize each such design as an ÆMILIA description, which is subsequently transformed into a queueing network model. Whenever one of these models is not in product form, the model itself is replaced by an approximating product-form queueing network model. The possibly approximate product-form queueing network model associated with each architectural design is then evaluated in order to derive the typical average performance indices both at the system level and at the component level. The evaluation is done in several different scenarios of interest and the obtained performance figures are interpreted on the various ÆMILIA descriptions.

On the basis of those figures, for each alternative a decision has to be made as to whether the design is satisfactory, should be discarded, or may be improved. When the predict-improve cycle is terminated for all the survived architectural designs, a comparison among them takes place in the various scenarios according to the average performance indices. The selected architecture is finally checked against the specific performance requirements of the system under construction.

This final check is necessary for two reasons. Firstly, the selection is made by relying on general performance indices, which are not necessarily connected in any way to the specific performance requirements. Secondly, the product-form queueing network model associated with the selected architecture may have been subject to approximations. Although the perturbation of the average performance indices introduced by the approximations cannot be easily quantified, we recall from [11] that queueing network models are in general robust, in the sense that even their approximate analysis is in any case helpful to get useful insights into the performance of the systems they represent.

The key point of PERFSEL is the combined use of the two above mentioned formalisms: ÆMILIA for component-oriented modeling purposes and queueing networks for component-oriented performance analysis purposes. As observed in [2, 1], the two formalisms are quite different from each other. On the one hand, ÆMILIA is a completely formal, general-purpose architectural description language handling both functional and performance aspects, whose basic ingredients are actions and behavioral operators. On the other hand, queueing networks are instances of a queue-based graphical notation for performance aspects only, in which some details like the queueing disciplines are usually expressed in natural language. Another important feature to take into account is the different level of granularity of the models expressed in the two formalisms. In particular, it turns out that the components of an ÆMILIA description cannot be precisely mapped to the customer populations and the service centers of a queueing network model, but on finer parts called queueing network basic elements that represent arrival processes, buffers, service processes, fork processes, join processes, and routing processes. Therefore, not all ÆMILIA descriptions can be transformed into queueing network models, but only those satisfying certain constraints specified in [2, 1].

This paper presents an implementation of PERFSEL and is organized as follows. In Sect. 2, we define the model transformation carried out by the plugin ÆMILIA_to_QN within TWOEAGLES. In Sect. 3, we introduce the plugin ÆMILIA_to_QN itself. In Sect. 4, we describe the architecture of TWOEAGLES and we show how it interoperates with other tools via ÆMILIA_to_QN. In Sect. 5, we illustrate by means of an automated teller machine example the adequacy of the model transformation and the higher degree of scalability achieved by TWOEAGLES in the performance evaluation of software architectures. Finally, in Sect. 6 we report some concluding remarks.

2 The Transformation from ÆMILIA to Queueing Networks

In this section, we present the transformation from ÆMILIA descriptions to queueing network models that we have implemented. More precisely, after recalling the transformation source (Sect. 2.1) and the transformation target (Sect. 2.2), we give an idea of how the transformation works in accordance with the queueing network basic elements identified in [2, 1] (Sect. 2.3). Finally, we detail the transformation through a hierarchy that we have specifically developed for our implementation of PERFSEL, which is composed of an action classification, a behavioral pattern classification, pattern combination rules, and connectivity rules for the queueing network basic elements (Sect. 2.4).

2.1 The Transformation Source: ÆMILIA

ÆMILIA [1] is an architectural description language based on stochastic process algebra. An ÆMILIA description represents an architectural type, which is a family of software systems sharing certain constraints on the observable behavior of their components as well as on their topology. As shown in Table 1, the textual description of an architectural type in ÆMILIA starts with its name and its formal parameters (initialized with default values), then comprises an architectural behavior section and an architectural topology section.

The first section defines the overall behavior of the system family by means of types of software components and connectors, which are collectively called architectural element types. The definition of an AET, which starts with its name and its formal parameters, consists of the specification of its behavior and its interactions.

`ARCHI_TYPE`	◁*name and initialized formal parameters*▷
`ARCHI_BEHAVIOR`	
⋮	⋮
`ARCHI_ELEM_TYPE`	◁*AET name and formal parameters*▷
`BEHAVIOR`	◁*sequence of stochastic process algebraic equations built from stop, action prefix, choice, and recursion*▷
`INPUT_INTERACTIONS`	◁*input synchronous/semi-synchronous/asynchronous uni/and/or-interactions*▷
`OUTPUT_INTERACTIONS`	◁*output synchronous/semi-synchronous/asynchronous uni/and/or-interactions*▷
⋮	⋮
`ARCHI_TOPOLOGY`	
`ARCHI_ELEM_INSTANCES`	◁*AEI names and actual parameters*▷
`ARCHI_INTERACTIONS`	◁*architecture-level AEI interactions*▷
`ARCHI_ATTACHMENTS`	◁*attachments between AEI local interactions*▷
`END`	

Table 1. Structure of an ÆMILIA textual description

The behavior of an AET has to be provided in the form of a sequence of behavioral equations written in a verbose variant of stochastic process algebra allowing only for the inactive process (rendered as `stop`), the action prefix operator supporting possible boolean guards and value passing, the alternative composition operator (rendered as `choice`), and recursion. Every action represents an activity and is described as a pair composed of the activity name and the activity duration. On the basis of their duration, actions are divided into exponentially timed (duration `exp(r)`), immediate (duration `inf(l, w)`), and passive (duration `_(l, w)`).

The interactions of an AET are actions occurring in the stochastic process algebraic specification of the behavior of the AET that act as interfaces for the AET itself, while all the other actions are assumed to represent internal activities. Each interaction has to be equipped with three qualifiers, with the first qualifier establishing whether the interaction is an input or output interaction.

The second qualifier represents the synchronicity of the communications in which the interaction can be involved. We distinguish among synchronous interactions which are blocking (default qualifier `SYNC`), semi-synchronous interactions which cause no blocking as they raise an exception if prevented (qualifier `SSYNC`), and asynchronous interactions which are completely decoupled from the other parties involved in the communication (qualifier `ASYNC`). Every semi-synchronous interaction is implicitly equipped with a boolean variable usable in the architectural description, which is automatically set to true if the interaction can be executed, false if an exception is raised.

The third qualifier describes the multiplicity of the communications in which the interaction can be involved. We distinguish among uni-interactions which are mainly involved in one-to-one communications (qualifier `UNI`), and-interactions guiding inclusive one-to-many communications like multicasts (qualifier `AND`), and or-interactions guiding selective one-to-many communications like in a server-clients setting (qualifier `OR`). It can also be established that an output or-interaction depends on an input or-interaction, in order to guarantee that a selective one-to-many output is sent to the same element from which a selective many-to-one input was received (keyword `DEP`).

The second section of an ÆMILIA description defines the topology of the system family. This is accomplished in three steps. Firstly, we have the declaration of the instances of the AETs – called AEIs – which represent the actual system components and connectors, together with their actual parameters. Secondly, we have the declaration of the architectural (as opposed to local) interactions, which are some of the interactions of the AEIs that act as interfaces for the whole systems of the family. Thirdly, we have the declaration of the architectural attachments among the local interactions of the AEIs, which make the AEIs communicate with each other. An attachment is admissible only if it goes from an output interaction of an AEI to an input interaction of another AEI. Moreover, a uni-interaction can be attached only to one interaction, whereas an and/or-interaction can be attached only to uni-interactions. Within a set of attached interactions, at most one of them can be exponentially timed or immediate.

The semantics for ÆMILIA is given by translation into stochastic process algebra. Basically, the semantics of every AEI is the sequence of stochastic process algebraic equations defining the behavior of the corresponding AET. Then, the semantics of an entire architectural description is the parallel composition of the semantics of the constituent AEIs, with synchronization sets determined by the attachments. From the state-transition graph underlying the resulting stochastic process term, a continuous-time Markov chain can be derived for performance evaluation purposes, provided that there are no transitions labeled with passive actions (performance closure) and all the transitions labeled with immediate actions are suitably removed.

2.2 The Transformation Target: Queueing Networks

A queueing network (see, e.g., [10, 11]) is a collection of interacting service centers that represent resources shared by classes of customers, where customer competition for resources corresponds to queueing into the service centers. In contrast to continuous-time Markov chains, queueing networks are structured performance models because they elucidate system components and their connectivity.

This brings a number of advantages in the architectural design phase. Firstly, typical average performance indices like throughput, utilization, mean queue length, and mean response time can be computed both at the level of an entire queueing network and at the level of its constituent service centers. Such global and local indicators can then be interpreted back at the level of an entire architectural description and at the level of its constituent components, respectively, in order to obtain diagnostic information. Secondly, there exist families of queueing networks that are equipped with fast solution algorithms that do not require the construction of the underlying state space. Among those families, we mention product-form queueing networks [4], which can be analyzed compositionally by solving each service center in isolation and then combining their solutions via multiplications. This provides support for a performance analysis that scales with respect to the number of components in architectural descriptions. Thirdly, the solution of a queueing network can be expressed symbolically in the case of certain topologies. This feature is useful in the early stages of the software development cycle, since the actual values of system performance parameters may be unknown at that time.

2.3 The Transformation at a Glance

As mentioned in Sect. 1, the transformation source and target are quite different from each other. In particular, the respective models have different levels of granularity. As a consequence, the AEIs of an ÆMILIA description cannot be precisely mapped to the customer populations and the service centers of a queueing network model. For this reason, in [2, 1] a number of finer parts called queueing network basic elements (QNBE for short) have been identified together with suitable syntactical restrictions that establish when an AEI can be transformed into one of those elements and when the AEIs from which those elements have been derived are connected in a way that yields a well-formed queueing network. The various QNBEs are shown in Fig. 1, where f (resp. r) denotes the number of alternative destinations (resp. sources), h denotes the number of customer classes, and interarrival and service times are expressed through phase-type distributions, i.e., suitable combinations of exponential distributions.

An arrival process is a generator of arrivals of customers of a certain class. While a single arrival process is enough in the case of an unbounded population, an instance of the arrival process is necessary for each customer in the case of a finite population, with the return of the customer being explicitly modeled. A buffer is a repository of customers of different classes that are waiting to be served according to some queueing discipline that we assume to be first-come-first-served. In the case of a bounded buffer, incoming customers of class i can be accommodated only if the buffer capacity c_i for that class is not exceeded. A service process is a server for customers of various classes, whose service times can be different for each class. When a service center is composed of multiple servers, it is necessary to represent each of them through an instance of the service process. A fork process splits requests coming from customers of a certain class into subrequests directed to different service centers, which are then recombined together by a join process. Finally, a routing process simply forwards customers of a certain class towards different destinations.

2.4 A Hierarchical Approach to the Transformation

We now describe the hierarchical approach that we have developed for implementing the transformation. As sketched in Sect. 2.3, we need to build a mapping between ÆMILIA elements and the QNBEs depicted in Fig. 1. Due to the notational gap between these two modeling languages, in the mapping implementation we have followed a bottom-up approach that starts from small-grained ÆMILIA elements and ends up to assemblies of QNBEs. In particular, this section presents: the ÆMILIA action classification, the ÆMILIA behavioral pattern classification, the ÆMILIA pattern combination rules to make QNBEs, and the connectivity rules for QNBEs.

For the sake of readability, in the remainder of this section actions are *italicized*, behavioral patterns are `typewritten`, and QNBEs are **bolded**.

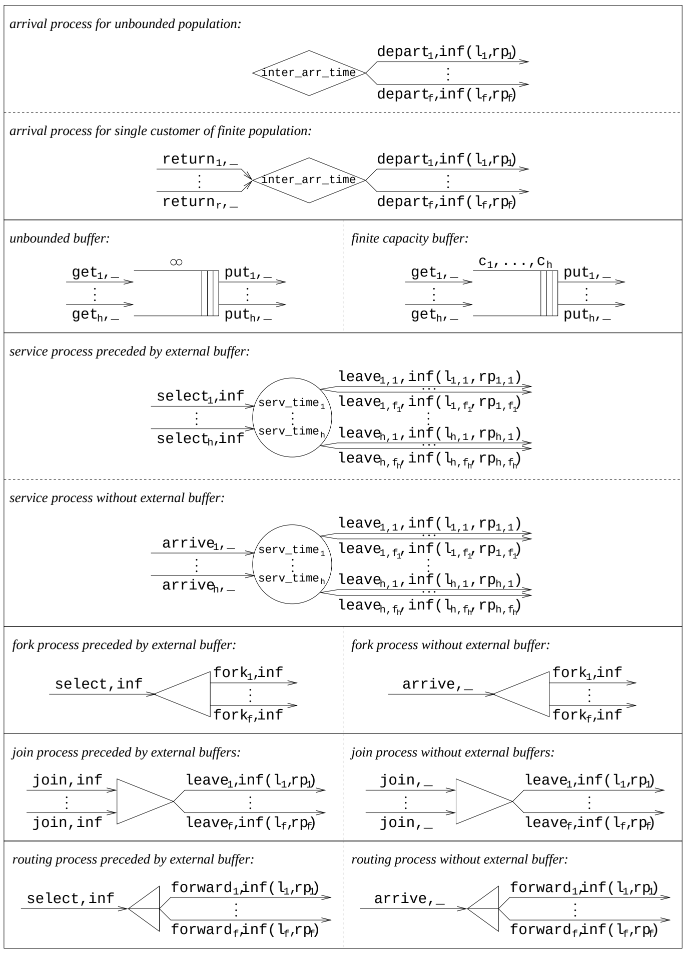

arrival process for unbounded population:
inter_arr_time
depart_1,inf(l_1,rp_1)
depart_f,inf(l_f,rp_f)
arrival process for single customer of finite population:
return_1,_
return_r,_
inter_arr_time
depart_1,inf(l_1,rp_1)
depart_f,inf(l_f,rp_f)
unbounded buffer:
∞
get_1,_
get_h,_
put_1,_
put_h,_
finite capacity buffer:
c_1, ..., c_h
get_1,_
get_h,_
put_1,_
put_h,_
service process preceded by external buffer:
select_1,inf
select_h,inf
serv_time_1
serv_time_h
leave_1,1,inf(l_1,1,rp_1,1)
leave_1,f_1,inf(l_1,f_1,rp_1,f_1)
leave_h,1,inf(l_h,1,rp_h,1)
leave_h,f_h,inf(l_h,f_h,rp_h,f_h)
service process without external buffer:
arrive_1,_
arrive_h,_
serv_time_1
serv_time_h
leave_1,1,inf(l_1,1,rp_1,1)
leave_1,f_1,inf(l_1,f_1,rp_1,f_1)
leave_h,1,inf(l_h,1,rp_h,1)
leave_h,f_h,inf(l_h,f_h,rp_h,f_h)
fork process preceded by external buffer:
select,inf
fork_1,inf
fork_f,inf
fork process without external buffer:
arrive,_
fork_1,inf
fork_f,inf
join process preceded by external buffers:
join,inf
join,inf
leave_1,inf(l_1,rp_1)
leave_f,inf(l_f,rp_f)
join process without external buffers:
join,_
join,_
leave_1,inf(l_1,rp_1)
leave_f,inf(l_f,rp_f)
routing process preceded by external buffer:
select,inf
forward_1,inf(l_1,rp_1)
forward_f,inf(l_f,rp_f)
routing process without external buffer:
arrive,_
forward_1,inf(l_1,rp_1)
forward_f,inf(l_f,rp_f)

Fig. 1. Queueing network basic elements

Action Classification In Table 2, the classification of ÆMILIA actions is illustrated. Actions are placed on the table rows. The first column is used to partition the rows into three groups of actions, which are: input interactions, output interactions, and internal actions. The other columns represent respectively: the name given to the action, the action duration (i.e., exponential, immediate, passive), the connection multiplicity of the action in the case that it is an interaction (i.e., uni, and, or), and the QNBEs that need such an action within their behavioral description (see Fig. 1).

For example, the first row of Table 2 specifies that a passive input uni-interaction might represent a *return* action in a **Single Client Arrival Process** of a queueing network. Similarly, an immediate internal action named *pre-exit* can belong to many different QNBEs such as **Arrival** and **Service Processes**.

	Action Name	Action Duration	Action Multiplicity	QNBEs
Input Interactions	*return*	passive	uni	**Single Client Arrival Process**
	get	passive	uni, or	**(Finite or Infinte Capacity) Buffer**
	select	immediate	uni	**Buffered Service Process, Buffered Fork Process, Buffered Routing Process**
	arrive	passive	uni, or	**Unbuffered Service Process, Unbuffered Fork Process, Unbuffered Routing Process**
	join	immediate	and	**(Buffered or Unbuffered) Join Process**
Output Interactions	*exit*	passive, immediate	uni, or	**(Single Client or Infinite) Arrival Process, (Buffered or Unbuffered) Service Process, (Buffered or Unbuffered) Join Process, (Buffered or Unbuffered) Routing Process**
	put	passive	uni, or	**(Finite or Infinte Capacity) Buffer**
	fork	immediate	and	**(Buffered and Unbuffered) Fork Process**
Internal Actions	*pre-exit*	immediate	N/A	**(Single Client or Infinite) Arrival Process, (Buffered or Unbuffered) Service Process, (Buffered or Unbuffered) Join Process, (Buffered or Unbuffered) Routing Process**
	exp-phase	exponential	N/A	**(Single Client or Infinite) Arrival Process, (Buffered or Unbuffered) Service Process**
	pre-phase	immediate	N/A	**(Single Client or Infinite) Arrival Process, (Buffered or Unbuffered) Service Process**

Table 2. ÆMILIA action classification

This classification helps to restrict the focus on the 11 actions that have been listed in Table 2. Behavioral patterns of interest for the transformation are built only using these actions. Therefore, such actions represent the alphabet to build words, which are the behavioral patterns introduced in next paragraph to model QNBE behaviors.

Behavioral Pattern Classification Following our bottom-up approach, we build up on the actions classified in Table 2 to obtain patterns that typically describe (partial) behaviors of QNBEs. The result is illustrated in Table 3, where we have identified 8 behavioral patterns. The table has an organization similar to the one of Table 2. Behavioral patterns are on the table rows and they are grouped into input, output, and internal behaviors. A name is assigned to each pattern, then the pattern is described in the third column of the table, and the QNBEs where such pattern occurs are listed in the fourth column. Parameters are associated with names of patterns that depend on their values (e.g., `uncond-get(i,n)`).

For example, the sixth row of Table 3 (i.e., the first output behavior) specifies that an `exit` behavior describes a job exiting a QNBE and routing somewhere else. Such a behavior can manifest itself in two ways: either as a set of unconditioned alternative behaviors, where each behavior has an *exit* action, or as a single *exit* action.

These behavioral patterns represent the words that can be used to build sentences representing QNBEs, as it will be illustrated in the next paragraph.

Combination Rules for Behavioral Patterns The last step to obtain the mapping illustrated in Fig. 1 between ÆMILIA constructs and QNBEs is accomplished by introducing rules to combine the previously identified behavioral patterns into QNBEs [1]. When parsing an ÆMILIA description, the ÆMILIA_to_QN component of Fig. 1 looks for such combinations of behavioral patterns in order to identify QNBEs within an ÆMILIA description and to generate them.

	Pattern Name	Pattern Description	QNBEs
Input Behaviors	`return`	The return of a job can be described in two different ways: (i) two or more alternative processes, each made of an unconditioned *return* action; (ii) a process represented by a *return* action	**Single Client Arrival Process**
	`uncond-get(i,n)`	An unconditioned get process for the i-th class of clients is made of an unconditioned *get* action, and a behavioral call with actual parameters $pa_1, pa_2, ..., pa_n$ that satisfy the following constraints: $pa_j = p_j + 1$ for $j = i$ and $pa_j = p_j$ for $j \neq i$, where p_j is the current number of clients of j-th class	**Infinite Capacity Buffer**
	`cond-get(i,n,`N_i`)`	A conditioned get process for the i-th class of clients is made of: the condition $p_i < N_i$ (where N_i is the buffer capacity for the i-th class of clients), a *get* action, and a behavioral call with actual parameters $pa_1, pa_2, ..., pa_n$ that satisfy the following constraints: $pa_j = p_j + 1$ for $j = i$ and $pa_j = p_j$ for $j \neq i$, where p_j is the current number of clients of j-th class	**Finite Capacity Buffer**
	`select`	A selection behavior is made of one or more alternative processes, where each process starts with an unconditioned *select* action	**Buffered Service Process, Buffered Fork Process, Buffered Routing Process**
	`arrive`	An arrival behavior is made of one or more alternative processes, where each process starts with an unconditioned *arrive* action	**Unbuffered Service Process, Unbuffered Fork Process, Unbuffered Routing Process**
Output Behaviors	`exit`	A job exit (and routing) can be described in two different ways: (i) two or more alternative processes, each made of an unconditioned *pre-exit* action followed by an *exit* action; (ii) a process represented by an *exit* action	**(Buffered or Unbuffered) Service Process, (Buffered or Unbuffered) Join Process, (Buffered or Unbuffered) Routing Process, (Single Client or Infinite) Arrival Process**
	`put(i,n)`	A put process for the i-th class of clients is made of: the condition $p_i > 0$, a *put* action, and a behavioral call with actual parameters $pa_1, pa_2, ..., pa_n$ that satisfy the following constraints: $pa_j = p_j - 1$ for $j = i$ and $pa_j = p_j$ for $j \neq i$, where p_j is the current number of clients of j-th class	**(Finite or Infinite Capacity) Buffer**
Internal Behaviors	`phase`	The behavior of a phase-type distribution is an arbitrary combination of *exp-phase* and *pre-phase* actions that determine a set of alternatives, where each alternative terminates with a non-phase behavior	**(Single Client or Infinite) Arrival Process, (Buffered or Unbuffered) Service Process**

Table 3. ÆMILIA behavioral pattern classification

These rules are defined in Table 4. The order of QNBEs in the table is the same as the one in Fig. 1. Each row in Table 4 represents a QNBE, where the second column provides the combination rules for behavioral patterns that define the QNBE behavior, whereas the third column represents additional assumptions that have to be verified

[1] All behaviors occurring in ÆMILIA descriptions are assumed to be tail recursive.

before generating the QNBE itself. Note that where behavioral patterns have been numbered (e.g., **Infinite Arrival Process**), it means that a simple sequencing rule has to be applied to those patterns. In some cases, such as **Buffered Fork Process**, rules have to be applied to simple actions beside behavioral patterns.

For example, a **Single Client Arrival Process** is defined as a sequence of three behavioral patterns (i.e., `phase`, `exit`, and `return`), with the additional conditions that the process must only have *exit* output interactions and *return* input interactions. As another example, a **Join Process** is defined as a sequence of a *join* action and an `exit` behavioral pattern, with the additional conditions that the only input interaction is the *join* action and the process must only have *exit* output interactions.

QNBE	Combination rules of behavioral patterns	Additional assumptions
Infinite Arrival Process	1. `phase` 2. `exit`	a. output interactions must only be *exit* b. no input interactions
Single Client Arrival Process	1. `phase` 2. `exit` 3. `return`	a. output interactions must only be *exit* b. input interactions must only be *return*
Infinite Capacity Buffer	· Parameters $p_1, p_2, \ldots, p_n$ have to be declared · Parameters have to be initialized to non-negative numbers · $2n$ alternative processes have to be defined: n `uncond-get(i,n)` patterns and n `put(i,n)` patterns	a. output interactions must only be *put* (with different names) b. input interactions must only be *get* (with different names)
Finite Capacity Buffer	· Parameters $p_1, p_2, \ldots, p_n$ have to be declared · Parameters have to be initialized to non-negative numbers and have to be all declared as intervals of integers · Each parameter p_i has to fall within the $[0, N_i]$ interval · $2n$ alternative processes have to be defined: n `cond-get(i,n)` patterns and n `put(i,n)` patterns	a. output interactions must only be *put* (with different names) b. input interactions must only be *get* (with different names)
Buffered Service Process	1. `select` 2. `phase` 3. `exit`	a. output interactions must only be *exit* b. input interactions must only be *select*
Unbuffered Service Process	1. `arrive` 2. `phase` 3. `exit`	a. output interactions must only be *exit* b. input interactions must only be *arrive*
Buffered Fork Process	1. *select* 2. *fork*	a. the only output interaction is the *fork* action b. the only input interaction is the *select* action
Unbuffered Fork Process	1. *arrive* 2. *fork*	a. the only output interaction is the *fork* action b. the only input interaction is the *arrive* action
Buffered or Unbuffered Join Process	1. *join* 2. `exit`	a. output interactions must only be *exit* b. the only input interaction is the *join* action
Buffered Routing Process	1. *select* 2. `exit`	a. output interactions must only be *exit* b. the only input interaction is the *select* action
Unbuffered Routing Process	1. *arrive* 2. `exit`	a. output interactions must only be *exit* b. the only input interaction is the *arrive* action

Table 4. Combining behavioral patterns into QNBEs

Connectivity Rules for Queueing Network Basic Elements Finally, we introduce several connectivity rules that allow QNBEs to be assembled in semantically valid queueing network models:

– An arrival process can be followed only by a service or fork process, possibly preceded by a buffer.
– A buffer can be followed only by a service, fork, join, or routing process.
– A service process can be followed by any QNBE.
– A fork process can be followed only by a service process or another fork process, possibly preceded by a buffer.
– A join process can be followed by any QNBE.
– A routing process can be followed by any QNBE.

3 The Eclipse Plugin ÆMILIA_to_QN

In order to enable the application of PERFSEL, we have developed ÆMILIA_to_QN, a Java-coded Eclipse plugin for transforming ÆMILIA descriptions into queueing networks. As shown in Sect. 2, the model transformations realized by ÆMILIA_to_QN rely on two queue-driven classifications – one for actions and one for behavioral patterns built from actions and process algebraic operators that can occur in ÆMILIA descriptions, respectively – which hierarchically formalize most of the syntactical restrictions of [2, 1]. These two classifications are then complemented by a number of rules establishing which combinations of behavioral patterns result in QNBEs (combination rules) and, in turn, how QNBEs should be connected to each other in order to yield well-formed queueing networks (connectivity rules).

Given an ÆMILIA description, ÆMILIA_to_QN parses the behavioral part of the ÆMILIA representation of each AEI to search for occurrences of the previously identified action classes and queue-like behavioral patterns. Whenever this search is successful on all AEIs and the combination rules are respected, then ÆMILIA_to_QN transforms each AEI into the corresponding QNBE. Afterwards, ÆMILIA_to_QN checks the topological part of the ÆMILIA description for compliance with the previously established connectivity rules of QNBEs. If this check succeeds too, ÆMILIA_to_QN transforms the entire ÆMILIA description into a queueing network model.

The queueing network models produced by ÆMILIA_to_QN are stored in PMIF format [13]. The reason is that this is an XML schema that acts as an interchange format and hence makes it possible to pass those models as input to queueing network tools.

4 The Architecture of TWOEAGLES

ÆMILIA_to_QN can be launched from within TWOEAGLES. This is a new version of TwoTowers [5] – a software tool for the functional verification, performance evaluation, and security analysis of software architectures described with ÆMILIA – which is entirely integrated in the Eclipse framework.

The software architecture of TWOEAGLES is depicted in Fig. 2, where the provided interfaces of components are represented by lollipops whereas required interfaces

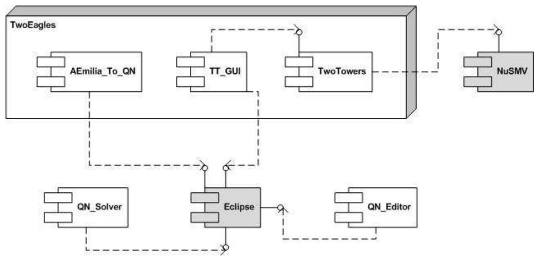


Fig. 2. Composition and environment of TWOEAGLES

by dashed arrows. In the box labeled with TWOEAGLES, only the components that strictly belong to the tool have been included, which are: ÆMILIA_to_QN, TT_GUI, and TwoTowers. The remaining components of Fig. 2 are either external tools (i.e., NuSMV and Eclipse) or in-house built components that support the TWOEAGLES task but do not belong to the tool (i.e., QN_Editor and QN_Solver).

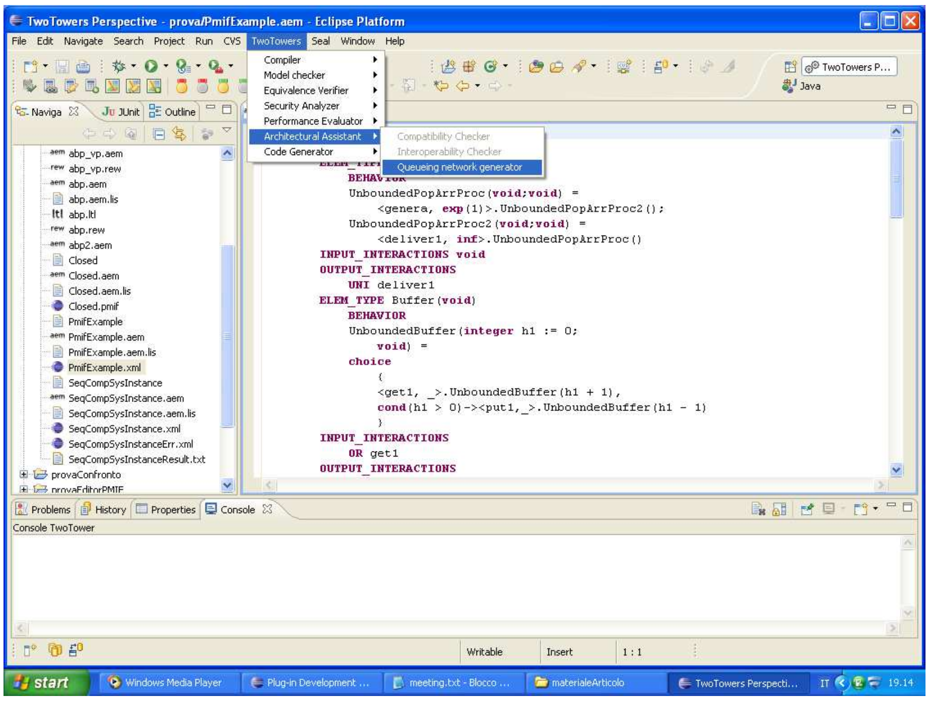
In order to embed TwoTowers in Eclipse, TWOEAGLES relies on a new graphical user interface for TwoTowers – the TT_GUI component in Fig. 2 – which has been developed as an Eclipse plugin. TT_GUI tailors the Eclipse environment to offer all the original TwoTowers functionalities to users, along with the model transformation introduced in this paper. This wrapping of TwoTowers has allowed its implementation to be kept basically unchanged – including the use of the external model checker NuSMV [6] – whereas its interfaces can be invoked, as they are, through the TT_GUI component.

In addition to TT_GUI, there are other three Eclipse plugins in Fig. 2. The first one, ÆMILIA_to_QN, has already been described in the previous section. Since it relies on PMIF, it acts as a bridge between TwoTowers on one side and the next two plugins on the other side. The second one, QN_Editor, allows PMIF-based queueing networks to be imported and edited in Eclipse and supports their graphical visualization. The third one, QN_Solver, is the client side of a web service called Weasel [15], which can be exploited to invoke several existing queueing network solvers.

A typical scenario for the architecture in Fig. 2 is the following. TWOEAGLES starts and TT_GUI, within Eclipse, is ready to accept user commands. For example, the user opens the ÆMILIA editor, which is part of TwoTowers, and enters an ÆMILIA description. The user may then run any (functional, security, or performance) analysis technique provided by the original TwoTowers release, including the NuSMV model checker. In addition, due to the extension presented in this paper, the user may decide to invoke a model transformation that generates a queueing network from the ÆMILIA description (i.e., the ÆMILIA_to_QN component in Fig. 2). The output queueing network, which can also be modified with the QN_Editor, is represented in PMIF and can be rendered (i) in a textual XML format through the standard Eclipse XML editor or (ii) in a graphical format through the QN_Editor component shown in Fig. 2. The latter is able to import and export queueing networks in PMIF format and to graphically represent them within Eclipse. Finally, the user can invoke the queueing network solver (i.e., QN_Solver in Fig. 2), which is a web service able to invoke different existing solvers. The solution results are represented in the standard Eclipse text editor.

5 TwoEagles at Work: An Automated Teller Machine

In this section, we illustrate TwoEagles at work on an automated teller machine (ATM), a system made of a certain number of distributed client terminals from where it is possible to require services to a central server. Common types of service requests are: withdrawal, deposit, and balance. Terminals only enable clients to perform I/O operations, whereas large part of computation is performed on the central server. The experiments that we illustrate on this example are aimed at: (i) showing the tool usability, (ii) validating the transformation from Æmilia descriptions to queueing network models on the basis of numerical results, and (iii) showing the larger scalability of queueing network solvers with respect to TwoTowers traditional performance evaluator.



```
        BEHAVIOR
        UnboundedPopArrProc(void;void) =
            <genera, exp(1)>.UnboundedPopArrProc2();
        UnboundedPopArrProc2(void;void) =
            <deliver1, inf>.UnboundedPopArrProc()
    INPUT_INTERACTIONS void
    OUTPUT_INTERACTIONS
        UNI deliver1
    ELEM_TYPE Buffer(void)
        BEHAVIOR
        UnboundedBuffer(integer h1 := 0;
            void) =
        choice
            {
            <get1, _>.UnboundedBuffer(h1 + 1),
            cond(h1 > 0)-><put1,_>.UnboundedBuffer(h1 - 1)
            }
    INPUT_INTERACTIONS
        OR get1
    OUTPUT_INTERACTIONS
```

Fig. 3. Eclipse user interface for TwoEagles

The application of Æmilia_to_QN (see Fig. 3) to the Æmilia description of the ATM system (which is not shown here due to lack of space) results in a queueing network formed by four QNBEs (corresponding to as many AEIs): *ThinkDevice*, which is the workload generator for the whole network, together with *CPU*, *DISKS*, and *VIDEO*, whose service times are 0.5 ms, 0.5 ms, and 1 ms, respectively (these values are only approximations of real scenarios, as we are more interested in the validation of the transformation and the analysis of pros and cons of our approach, rather than in the numbers themselves). Jobs originated from *ThinkDevice* are delivered to *CPU*. On the basis of interaction rates, *CPU* decides whether sending jobs to *DISKS* and/or *VIDEO*. Thereafter, jobs are sent back to *ThinkDevice*. Being a closed queueing network, the parameters that drive our performance analysis are: the number N of client terminals and the thinking time Z of each client.

In Fig. 4, we have reported the throughput (on the left) and the utilization (on the right) that we have obtained for the main ATM devices, which are *CPU* and *DISKS*, while varying the number of clients N in the system, with a fixed thinking time of $Z = 1$ s. Four curves are shown because for each device we have represented both the values obtained with the TwoTowers performance solver and the ones obtained with the external queueing network solver after the ÆMILIA description has been transformed into a queueing network. As can be seen, the two solvers obtain exactly the same numerical results for both considered devices, and this supports the correctness of the transformation of the ÆMILIA description into the queueing network model. However, the TwoTowers solver, whose results are labeled as TwoTowers in the figure, is unable to solve models with more than 7 clients. This is due to the state space explosion phenomenon encountered when the solver handles the continuous-time Markov chain model. In contrast, the queueing network solver, whose results are labeled as PMVA in the figure, is able to solve larger models in few seconds. This is due to the product form [10] of the resulting queueing network model, which allows polynomial-time solution algorithms such as Mean Value Analysis (MVA) to be applied.

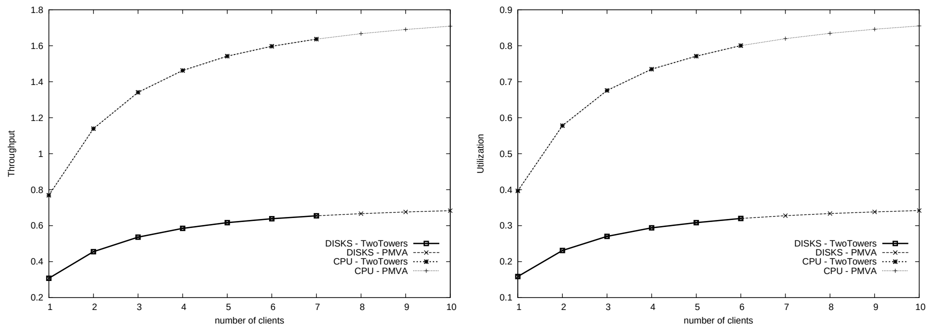


Fig. 4. ATM throughput (left) and utilization (right)

6 Conclusions and Future Work

In this paper, we have presented a tool integrated into Eclipse that allows ÆMILIA-based architectural descriptions to be transformed into queueing network models and hence supports the PERFSEL methodology of [2, 1]. As shown by the ATM example, the tool TWOEAGLES improves on TwoTowers because the possibility of exploiting queueing network solvers makes the performance evaluation process faster and applicable to larger software architectures with respect to continuous-time Markov chain solvers.

Many approaches have been introduced in the last decade to transform architectural models into performance models [3], but very few of them have been implemented in working tools and rely on structured models like queueing networks. Moreover, most implementations are based on UML, whereas in TwoEagles we consider a fully fledged, formally defined architectural description language as source notation.

With regard to future work, we intend to strengthen the transformation implemented in TwoEagles by moving from a general-purpose programming language like Java to model transformation languages like ATL [9] and QVT [12]. Moreover, we would like to investigate whether results relating stochastic process algebras and queueing networks [8, 7, 14] can be exploited in our architectural framework.

Acknowledgment: Work funded by MIUR-PRIN project *PaCo – Performability-Aware Computing: Logics, Models, and Languages*.

References

1. A. Aldini, M. Bernardo, and F. Corradini, *"A Process Algebraic Approach to Software Architecture Design"*, Springer, 2010.
2. S. Balsamo, M. Bernardo, and M. Simeoni, *"Performance Evaluation at the Software Architecture Level"*, in *"Formal Methods for Software Architectures"*, Springer, LNCS 2804:207–258, 2003.
3. S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, *"Model-Based Performance Prediction in Software Development: A Survey"*, in IEEE Trans. on Software Engineering 30:295–310, 2004.
4. F. Baskett, K.M. Chandy, R.R. Muntz, and G. Palacios, *"Open, Closed, and Mixed Networks of Queues with Different Classes of Customers"*, in JACM 22:248–260, 1975.
5. M. Bernardo, *"TwoTowers 5.1 User Manual"*, `http://www.sti.uniurb.it/bernardo/twotowers/`, 2006.
6. R. Cavada, A. Cimatti, E. Olivetti, M. Pistore, and M. Roveri, *"NuSMV 2.1 User Manual"*, 2002.
7. P.G. Harrison, *"Compositional Reversed Markov Processes, with Applications to G-Networks"*, in Performance Evaluation 57:379–408, 2004.
8. J. Hillston and N. Thomas, *"Product Form Solution for a Class of PEPA Models"*, in Performance Evaluation 35:171–192, 1999.
9. F. Jouault, F. Allilaire, J. Bezivin, and I. Kurtev, *"ATL: A Model Transformation Tool"*, in Science of Computer Programming 72:31–39, 2008.
10. L. Kleinrock, *"Queueing Systems"*, John Wiley & Sons, 1975.
11. E.D. Lazowska, J. Zahorjan, G. Scott Graham, and K.C. Sevcik, *"Quantitative System Performance: Computer System Analysis Using Queueing Network Models"*, Prentice Hall, 1984.
12. OMG, "Query/View/Transformation", formal/08-04-03.
13. C.U. Smith and C.M. Llado, *"Performance Model Interchange Format (PMIF 2.0): XML Definition"*, in Proc. of QEST 2004, IEEE-CS Press, pp. 38–47, 2004.
14. N. Thomas and Y. Zhao, *"Mean Value Analysis for a Class of PEPA Models"*, in Proc. of EPEW 2009, Springer, LNCS 5652:59–72, 2009.
15. S. Zallocco, *"Web sErvice for Analyzing queueing networkS with multiplE soLvers"*, `http://sealabtools.di.univaq.it/Weasel/`, 2006.