Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language

Marco Bernardo[1], Lorenzo Donatiello[2], and Paolo Ciancarini[2]

[1] Università di Urbino, Centro per l'Appl. delle Sc. e Tecn. dell'Inf.
Piazza della Repubblica 13, 61029 Urbino, Italy
bernardo@sti.uniurb.it

[2] Università di Bologna, Dipartimento di Scienze dell'Informazione
Mura Anteo Zamboni 7, 40127 Bologna, Italy
donat, cianca@cs.unibo.it

Abstract. The objective of this tutorial is to describe the evolution of the field of stochastic process algebra in the past decade, through a presentation of the main achievements in the field. In particular, the tutorial stresses the current transformation of stochastic process algebra from a simple formalism to a fully fledged architectural description language for the functional verification and performance evaluation of complex computer, communication and software systems.

1 Introduction

Many computing systems consist of a possibly huge number of components that not only work independently but also communicate with each other. Examples of such systems are communication protocols, operating systems, embedded control systems for automobiles, airplanes, and medical equipment, banking systems, automated production systems, control systems of nuclear and chemical plants, railway signaling systems, air traffic control systems, distributed systems and algorithms, computer architectures, and integrated circuits.

The catastrophic consequences – loss of human lives, environmental damages, and financial losses – of failures in many of these critical systems have compelled computer scientists and engineers to develop techniques for ensuring that these systems are designed and implemented correctly despite of their complexity. The need of formal methods in developing complex systems is becoming well accepted. Formal methods seek to introduce mathematical rigor into each stage of the design process in order to build more reliable systems.

The need of formal methods is even more urgent when planning and implementing concurrent and distributed systems. In fact, they require a huge amount of detail to be taken into account (e.g., interconnection and synchronization structure, allocation and management of resources, real time constraints, performance requirements) and involve many people with different skills in the project (designers, implementors, debugging experts, performance and quality analysts). A uniform and formal description of the system under investigation reduces misunderstandings to a minimum when passing information from one task of the project to another.

Moreover, it is well known that the sooner errors are discovered, the less costly they are to fix. Consequently, it is imperative that a correct design is available before implementation begins. Formal methods are conceived to allow the correctness of a system design to be formally verified. Using formal methods, the design can be described in a mathematically precise fashion, correctness criteria can be specified in a similarly precise way, and the design can be rigorously proved to meet or not the stated criteria.

Although a number of description techniques and related software tools have been developed to support the formal modeling and verification of functional properties of systems, only in recent years temporal characteristics have received attention. This has required extending formal description techniques by introducing the concept of time, represented either in a deterministic way or in a stochastic way.

In the deterministic case, the focus typically is on verifying the satisfaction of real time constraints, i.e. the fact that the execution of specific actions is guaranteed by a fixed deadline after some event has happened. As an example, if a train is approaching a railroad crossing, then bars must be guaranteed to be lowered on due time.

In the stochastic case, instead, systems are considered whose behavior cannot be deterministically predicted as it fluctuates according to some probability distribution. Due to economic reasons, such stochastically behaving systems are referred to as shared resource systems, because there is a varying number of demands competing for the same resources. The consequences are mutual interference, delays due to contention, and varying service quality. Additionally, resource failures significantly influence the system behavior. In this case, the focus is on evaluating the performance of the systems. As an example, if we consider again a railway system, we may be interested in minimizing the average train delay or studying the characteristics of the flow of passengers. The purpose of performance evaluation is to investigate and optimize the time varying behavior within and among individual components of shared resource systems. This is achieved by modeling and assessing the temporal behavior of systems, identifying characteristic performance measures, and developing design rules that guarantee an adequate quality of service.

The desirability of taking account of the performance aspects of shared resource systems in the early stages of their design has been widely recognized [33, 68] and has fostered the development of formal methods for both functional verification and performance evaluation of rigorous system models. The focus of this tutorial is on stochastic process algebra (SPA), a formalism proposed in a seminal work by Ulrich Herzog [46, 47] in the early '90s, whose growing interest is witnessed by the annual organization of the international workshop on Process Algebra and Performance Modeling (PAPM) and a number of Ph.D. theses on this subject [48, 38, 70, 67, 63, 54, 62, 59, 40, 53, 10, 31, 24, 29, 22, 25]. With respect to formalisms traditionally used for performance evaluation purposes like Markov chains (MCs) and queueing networks (QNs) [56, 58], SPA provides a more complete framework in which also functional verification can be carried out. With respect to previous formal methods for performance evaluation like stochastic Petri nets (SPNs) [1], SPA provides novel capabilities related to compositionality and abstraction that help system modeling and analysis. The first part of this tutorial (Sect. 2) is devoted to the presentation of the main results achieved in the field of SPA since the early '90s.

Although SPA supports compositional modeling via algebraic operators, this feature has not been exploited yet to enforce a more controlled way of describing systems that makes SPA technicalities transparent. By this we mean that in a SPA specification the basic concepts of system component and connection are not clearly elucidated, nor checks are available to detect mismatches when assembling components together. Since nowadays systems are made out of numerous components, in the early design stages it is crucial to be equipped with a formal specification language that permits to reason in terms of components and component interactions and to identify components that result in mismatches when put together. The importance of this activity is witnessed by the growing interest in the field of software architecture and the development of architectural description languages (ADLs) [61, 66]. The formal description of the architecture of a complex system serves two purposes. First and foremost is making available a precise document describing the structure of the system to all the people involved in the design, implementation, and maintainance of the system itself. The second one is concerned with the possibility of analyzing the properties of the system at the architectural level, thus allowing for the early detection of design errors. The second part of this tutorial (Sect. 3) is devoted to show how SPA can easily be transformed into a compositional, graphical and hierchical ADL endowed with some architectural checks, which can be profitably employed for both functional verification and performance evaluation at the architectural level of design.

The tutorial finally concludes with some remarks about future directions in the field of SPA based ADLs.

2 SPA: Basic Notions and Main Achievements

SPA is a compositional specification language of algebraic nature that integrates process algebra theory [60, 50, 5] and stochastic processes. In this section we provide a quick overview of the basic notions about the syntax, the semantics, and the equivalences for SPA, as well as the main results and applications that have been developed in the past decade.

2.1 Syntax: Actions, Operators, and Synchronization Disciplines

SPA is characterized by three main ingredients: the actions modeling the system activities, the algebraic operators whereby composing the subsystem specifications, and the synchronization disciplines.

An action is usually composed of a type a and an exponential rate λ: $<a, \lambda>$ [48, 45, 27]. The type indicates the kind of activity that is performed by the system at a certain point, while the rate indicates the reciprocal of the average duration of the activity assuming that the duration is an exponentially distributed random variable. A special action type, traditionally denoted by τ, designates a system activity whose functionality cannot be observed and serves for functional abstraction purposes. In order to increase the expressiveness, in [10] prioritized, weighted immediate actions of the form $<a, \infty_{l,w}>$ are proposed, which are useful to model activities whose timing is irrelevant from the performance viewpoint as well as activities whose duration follows a phase type distribution. In alternative to the durational actions considered so far, in [40] a different view is taken according to which an action is either an instantaneous activity a or an exponentially distributed time passage λ.

Several algebraic operators are usually present. The zeroary operator $\underline{0}$ represents the term that cannot execute any action. The action prefix operator $<a, \lambda>.E$ denotes the term that can execute an action with type a and rate λ and then behaves as term E; in the approach of [40], there are the two action prefix operators $a.E$ and $\lambda.E$. The functional abstraction operator E/L, where L is a set of action types not including τ, denotes the term that behaves as term E except that the type a of each executed action is turned into τ whenever $a \in L$. The functional relabeling operator $E[\varphi]$, where φ is a function over action types preserving observability, denotes a term that behaves as term E except that the type a of each executed action becomes $\varphi(a)$. The alternative composition operator $E_1 + E_2$ denotes a term that behaves as either term E_1 or term E_2 depending on whether an action of E_1 or an action of E_2 is executed. The action choice is regulated by the race policy (the fastest one succeeds), so that each action of E_1 and E_2 has an execution probability proportional to its rate. In the approach of [10], immediate actions take precedence over exponentially timed ones and the choice among them is governed by the preselection policy: the lower priority immediate actions are discarded, then each of the remaining immediate actions is given an execution probability proportional to its weight. In the approach of [40], the choice between two instantaneous activities is nondeterministic. The parallel composition operator $E_1 \parallel_S E_2$, where S is a set of action types not including τ, denotes a term that asynchronously executes actions of E_1 or E_2 whose type does not belong to S, and synchronously executes – according to a synchronization discipline – equally typed actions of E_1 and E_2 whose type belongs to S. Finally, a constant A denotes a term that behaves according to the associated defining equation $A \stackrel{\Delta}{=} E$, which allows for recursive behaviors.

There are many different synchronization disciplines. In [45] the rate of the action resulting from the synchronization of two actions is the product of the rates of the two synchronizing actions, where the physical interpretation is that one rate is the formal rate and the other rate acts like a scaling factor. In [48] the bounded capacity assumption is introduced, according to which the rate of an action cannot be increased/decreased due to the synchronization with another action of the same type. In this approach, patient synchronizations are considered, i.e. the rate of the action resulting from the synchronization of two equally typed actions of E_1 and E_2 is given by the minimum of the two total rates with which E_1 and E_2 can execute actions of the considered type, multiplied by the local execution probabilities of the two synchronizing actions. Following the terminology of [36], in [26] a generative-reactive synchronization discipline complying with the bounded capacity assumption is adopted, which is based on the systematic use of prioritized, weighted passive actions of the form $<a, *_{l,w}>$. The idea is that the nonpassive actions probabilistically determine the type of action to be executed at each step, while the passive actions of the determined type probabilistically react in order to identify the subterms taking part in the synchronization. In order for two equally typed actions to synchronize, in this approach one of them must be passive and the rate of the resulting action is given by the rate of the nonpassive action multiplied by the local execution probability of the passive action. Finally, in [40] equal instantaneous activities can synchronize, while time passages cannot. Therefore, when both E_1 and E_2 can let time pass, in this approach the overall time passage is the maximum of the two local, exponentially distributed time passages.

2.2 Semantics: Interleaving and Memoryless Property

The semantics for SPA is defined in an operational fashion by means of a set of axioms and inference rules that formalize the meaning of the algebraic operators. The result of the application of such rules is a labeled transition system (LTS), where states are in correspondence with process terms and transitions are labeled with actions. As an example, the axiom for the action prefix operator

$$<a, \lambda>.E \xrightarrow{a,\lambda} E$$

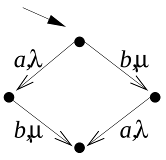
establishes that term/state $<a, \lambda>.E$ can evolve into term/state E by performing action/transition $<a, \lambda>$. As another example, the inference rule for the functional relabeling operator

$$\frac{E \xrightarrow{a,\lambda} E'}{E[\varphi] \xrightarrow{\varphi(a),\lambda} E'[\varphi]}$$

establishes that, whenever term/state E can evolve into term/state E' by performing action/transition $<a, \lambda>$, term/state $E[\varphi]$ can evolve into term/state $E'[\varphi]$ by performing action/transition $<\varphi(a), \lambda>$.

The most complicated inference rules are those for the alternative composition operator and the parallel composition operator. As far as the alternative composition operator is concerned, the problem is that, in the case of terms like $<a, \lambda>.E + <a, \lambda>.E$, the transition generation process must keep track of the fact that the total rate is $2 \cdot \lambda$ by virtue of the race policy. In [48] it is proposed to use labeled multitransition systems, so that a single transition labeled with $<a, \lambda>$ is generated for the term above, which has multiplicity two. In [45], instead, it is proposed to decorate the transitions with an additional distinguishing label, whose value depends on whether the transitions are due to the left hand side or the right hand side summand of the alternative compositions. As far as the parallel composition operator is concerned, the related inference rules must embody the desired synchronization discipline.

The resulting LTS is an interleaving semantic model, which means that every parallel computation is represented through a choice between all the sequential computations that can be obtained by interleaving the execution of the actions of the subterms composed in parallel. As an example, the parallel term $<a, \lambda>.\underline{0} \parallel_{\emptyset} <b, \mu>.\underline{0}$ and the sequential term $<a, \lambda>.<b, \mu>.\underline{0} + <b, \mu>.<a, \lambda>.\underline{0}$ are given the same LTS up to state names:



This is correct from the functional viewpoint, because an external observer, who is not aware of the structure of the systems represented by the two terms, sees exactly the same behavior. Moreover, this is correct from the performance viewpoint as well, by virtue of the memoryless property of the exponential distribution. For instance, if in the parallel term action $<a, \lambda>$ is completed before action $<b, \mu>$, then state $\underline{0} \parallel_{\emptyset} <b, \mu>.\underline{0}$ is reached and the time to the completion of action $<b, \mu>$ is still exponentially distributed with rate μ. In other words, the interleaving style fits well with the fact that the execution of an exponentially timed action can be considered as being started in the state in which it terminates.

The LTS produced by applying the operational semantic rules to a process term represents the integrated semantic model of the process term. It can undergo to integrated analysis techniques, like integrated model checking [8] and integrated equivalence checking (see Sect. 2.3), to detect mixed functional-performance properties, like the probability of executing a certain sequence of activities. From the integrated semantic model, two projected semantic models can be derived. The functional semantic model is a LTS obtained by discarding information about action rates; it can be analyzed through traditional techniques like model checking [30] and equivalence/preorder checking [28]. The performance semantic model is a LTS obtained by discarding information about action types, which happens to be a continuous time Markov chain (CTMC). In the approach of [26], where prioritized, weighted immediate and passive actions are considered, the projected semantic models are generated after pruning the lower priority transitions from the integrated semantic model. Furthermore, the performance semantic model can be generated only if the integrated semantic model is performance closed, i.e. has no passive transitions. If this is the case, the performance semantic model is a CTMC whenever the integrated semantic model has only exponentially timed transitions or both exponentially timed and immediate transitions (in which case states having outgoing immediate transitions are removed as their sojourn time is zero). If instead the integrated semantic model has only immediate transitions, then it is assumed that the execution of each of them takes one time unit so that the performance model turns out to be a discrete time Markov chain (DTMC). CTMCs and DTMCs can then be analyzed through standard techniques [69], mainly based on rewards [52], to derive performance measures.

2.3 Equivalences: Congruence and Lumpability

SPA terms can be equated on the basis of their functional and performance behavior. The mostly used method is that, inspired by [57], of the Markovian bisimulation equivalence [48, 45, 27], based on the ability of two terms of simulating each other behavior. The idea is that, given an equivalence relation $\mathcal{B}$ over process terms, $\mathcal{B}$ is a Markovian bisimulation if, for each pair $(E_1, E_2) \in \mathcal{B}$, action type a, and equivalence class C of $\mathcal{B}$, the total rate with which E_1 reaches states in C by executing actions of type a is equal to the total rate with which E_2 reaches states in C by executing actions of type a. The Markovian bisimulation equivalence is then defined as the union of all the Markovian bisimulations.

The Markovian bisimulation equivalence enjoys several properties. First, it is a congruence w.r.t. all the operators as well as recursive constant defining equations [48, 45, 27, 26]. This ensures substitutivity, i.e. compositionality at the semantic level: given a term, if any of each subterms is replaced by a Markovian bisimulation equivalent subterm, the new term is Markovian bisimulation equivalent to the original one. Second, the Markovian bisimulation equivalence complies with the ordinary lumping for MCs [64], thus ensuring that equivalent terms possess the same performance characteristics [48, 27]. Third, the Markovian bisimulation equivalence is the coarsest congruence contained in the intersection of the bisimulation equivalence [60] and the ordinary lumping, which means that it is the best Markovian equivalence we can hope for in a bisimulation setting [10]. Fourth, the Markovian bisimulation equivalence has a sound and complete axiomatization – with $<a, \lambda_1>.E + <a, \lambda_2>.E = <a, \lambda_1 + \lambda_2>.E$ as typical axiom besides the usual expansion law for the parallel composition operator – which provides an alternative characterization easier to understand [45].

There are some variants of the Markovian bisimulation equivalence. In the approach of [26], the Markovian bisimulation equivalence is extended to deal with prioritized, weighted immediate and passive actions. In the approach of [40], a weak Markovian bisimulation equivalence is defined that abstracts from instantaneous τ activities. In [48], a different weak Markovian bisimulation equivalence is proposed that, in some cases, abstracts from exponentially timed τ actions. Finally, an alternative view is taken in [17]: following the testing approach of [32], it is proposed of equating two terms whenever they have the same probability to pass the same tests within the same average time. The resulting equivalence, called Markovian testing equivalence, is coarser than the Markovian bisimulation equivalence, abstracts from internal immediate actions and in some cases from internal exponentially timed actions, and possesses an alternative characterization in terms of extended traces. The congruence property, the axiomatization, and the relationship with the ordinary lumping for the Markovian testing equivalence are still under investigation. As far as ordinary lumping is concerned, it is known that in some cases the Markovian testing equivalence produces a more compact exact aggregation.

2.4 Performance Properties: Algebraic and Logic Approaches

SPA provides the capability of expressing the performance aspects of the behavior of complex systems, but not the performance properties of interest. In a Markovian framework, stationary and transient performance measures (system throughput, resource utilization, average buffer occupation, mean response time, etc.) are usually described as weighted sums of state probabilities and transition frequencies, where state weights are called yield rewards and transition weights are called bonus rewards [52].

In [13, 12] it is proposed to reuse the classical technique of rewards by extending the action format to include as many pairs of yield and bonus rewards as there are performance measures of interest. In this framework, at semantic model construction time every state is given a yield reward that is equal to the sum of the yield rewards of the actions it can execute. The Markovian bisimulation equivalence is then extended to take rewards into account, in a way that preserves compositionality as well as the performance measures of interest.

In [29] an alternative reward based approach is proposed, which associates certain rewards with those states satisfying certain formulas of a Markovian modal logic that characterizes the Markovian bisimulation equivalence. This approach is implemented through a high level language for enquiring about the stationary performance characteristics possessed by a process term. Such a language, whose formal underpinning is constituted by the Markovian modal logic, relies on the combination of the standard mathematical notation, a notation based on the Markovian bisimulation equivalence to focus queries directly on states, and a notation expressing the potential to perform an action of a given type.

Finally, in [8, 7, 6] it is proposed to directly express the performance properties of interest through logical formulas, whose validity is verified through an integrated model checking procedure. The continuous stochastic logic is used in this framework to inquiry about the value of stationary and transient performability measures of a system. According to the observation that the progress of time can be regarded as the earning of reward, a reward based variant of such a logic is then introduced, where yield rewards are assumed to be already attached to the states.

2.5 General Distributions

When introducing generally distributed durations in SPA, the memoryless property can no longer be exploited to define the semantics in the plain interleaving style. The reason is that the actions can no more be thought of as being started in the states where they are terminated; the underlying performance models are no longer MCs. Therefore, we have to keep track of the sequence of states in which an action started and continued its execution.

There are several approaches in the literature, among which we mention below those for which a notion of equivalence (in the bisimulation style) is developed. In [70] the problem of identifying the start and the termination of an action is solved at the syntactic level by means of suitable operators that represent the random setting of a timer and the expiration of a timer, respectively. Semantic models are infinite LTSs from which performance measures can be derived via simulation.

In [31] the problem is again solved at the syntax level through suitable clock related operators, with the difference that the semantic models are finitely represented through stochastic automata equipped with clocks.

In [25], instead, the problem of identifying the start and the termination of an action is addressed at the semantic level through the ST approach of [37]. At semantic model construction time, the start and the termination of each action are distinguished and kept connected to each other. This framework naturally supports action refinement, which can be exploited to replace a generally timed action with a process term composed only of exponentially timed actions resulting in a phase type duration that approximates the original duration.

2.6 State Space Explosion

The semantic models for SPA are state based, hence suffer from the state space explosion problem, i.e. the fact that the size of the state space grows exponentially with the number of subterms composed in parallel. In general, this problem can be tackled with traditional congruence based techniques. For instance, it is wise to build the state space underlying a process term in a stepwise fashion, along the structure imposed by the occurrences of the parallel composition operator, and minimize the state space obtained at every step according to the Markovian bisimulation equivalence. An alternative strategy is to operate at the syntactical level using the axioms of the Markovian bisimulation equivalence as rewriting rules.

More specific techniques to fight the state space explosion problem are present in the literature. Among them we mention those based on Kronecker representation [27, 67], time scale decomposition [59], product form solution [39, 65, 49], symbolic representation [44], stochastic Petri net semantics [14], and queueing network representation [9].

2.7 Tools and Case Studies

A few tools are under distribution for the modeling and analysis of systems with SPA. Among them we mention the PEPA Workbench [34], the TIPPtool [55], and TwoTowers [18].

With such tools several case studies have been conducted, which are concerned with computer systems, communication protocols, and distributed algorithms. Among such case studies we mention those related to CSMA/CD [10], token ring [10], electronic mail system [41], multiprocessor mainframe [42], industrial production cell [51], robot control [35], plain old telephone system [43], multimedia stream [23], adaptive mechanisms for transmitting voice over IP [21, 3], ATM switches [2], replicated web services [11], Lehmann-Rabin randomized algorithm for dining philosophers [10], and comparison of six mutual exclusion algorithms [13].

3 Turning SPA into an ADL

SPA supports the compositional modeling of complex systems via algebraic operators. However, this feature has not been exploited yet to enforce an easier and more controlled way of describing systems that makes SPA technicalities transparent to the designer. As an example, if a system is made out of a certain number of components, with SPA the system is simply described as the parallel composition of a certain number of subterms, each representing the behavior of a single component, with suitable synchronization sets to represent the component interactions. It is desirable to be able to describe the same system at a higher level of abstraction, where the parallel composition operators and the related synchronization sets do not come into play. It is more natural to separately define 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 view brings the advantage that the system components and the component interactions are clearly elucidated, with the synchronization mechanism being hidden (e.g. interacting actions must not necessarily have the same type). Another strength is the capability of defining the behavior – possibly parametrized w.r.t. action rates – and the interactions of a component type just once and subsequently reusing it as many times as there are instances of that component type in the system. Additionally, it is desirable that composite systems can be described in a hierachical way, and that a graphical support is provided for the whole modeling process.

Besides this useful syntactical sugar, checks are needed to detect possible mismatches when assembling components together and to identify the components that cause such mismatches. A typical example is deadlock freedom. If we put together some components that we know to be deadlock free, we would like that their combination is still deadlock free. In order to investigate that, we need suitable checks that allow deadlock to be quickly detected and some diagnostic information to be obtained for localizing the source of deadlock. As another example, in order to evaluate the performance of a system, its model must be performance closed. In this case, a check at the syntax level is helpful to easily detect and pinpoint possible violations of the performance closure.

In this section we show how SPA can be enhanced to work with at the architectural level of design. Based on ideas contained in [4, 16, 19, 20], we illustrate how SPA can be turned into a fully fledged ADL for the modeling, functional verification, and performance evaluation of complex systems. Recalled that the transformation is largely independent of the specific SPA, we concentrate on

$\mathrm{EMPA_{gr}}$ [26] – which includes prioritized, weighted immediate and passive actions and the generative-reactive synchronization discipline – and we exhibit the resulting SPA based ADL called Æmilia [15, 9]. The description of a system with Æmilia can be done in a compositional, hierachical, graphical and controlled way. First, we have to define the behavior of the types of components in the system and their interactions with the other components. The functional and performance aspects of the behavior are described through a family of $\mathrm{EMPA_{gr}}$ terms or the invocation of the specification of a previously modeled system, while the interactions are described through actions occurring in the behavior. Second, we have to declare the instances of each type of component present in the system and the way in which their interactions are attached to each other in order to allow the instances to communicate. This process is supported by a graphical notation. Then, the whole behavior of the system is a family of $\mathrm{EMPA_{gr}}$ terms transparently obtained by composing in parallel the behavior of the declared instances according to the specified attachments. From the whole behavior, integrated, functional and performance semantic models can be automatically derived, which can undergo to the analysis techniques mentioned in Sect. 2. In addition to that, Æmilia comes equipped with some architectural checks for ensuring deadlock freedom and performance closure.

3.1 Components and Topology: Textual and Graphical Notations

A description in Æmilia represents an architectural type. As shown in Table 1, the description of an architectural type starts with the name of the architectural type and its numeric parameters, which often are values for exponential rates and weights. Each architectural type is defined as a function of its architectural element types (AETs) and its architectural topology. An AET is defined as a function of its behavior, specified either as a family of sequential [1] $\mathrm{EMPA_{gr}}$ terms or through an invocation of a previously defined architectural type, and its interactions, specified as a set of $\mathrm{EMPA_{gr}}$ 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 (as opposed to local) 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. Every interaction is declared to be an input interaction or an output interaction and the attachments must respect such a classification: every attachment must involve an output interaction and an input interaction of two different AEIs. An AEI can have different types of interactions (input/output, local/architectural); it must have at least one local interaction. Every local interaction must be involved in at least one attachment, while every architectural interaction must not be involved in any attachment. In order to allow several AEIs to synchronize, every local interaction can be involved in

[1] Including only $\underline{0}$, constants, action prefix operators, and alternative composition operators.

several attachments provided that no autosynchronization arises, i.e. no chain of attachments is created that starts from a local interaction of an AEI and terminates on a local interaction of the same AEI. On the performance side, we require that, for the sake of modeling consistency, all the occurrences of an action type in the behavior of an AET have the same kind of rate (exponential, immediate with the same priority level, or passive with the same priority level) and that, to comply with the generative-reactive synchronization discipline of EMPA$_{\text{gr}}$, every chain of attachments contains at most one interaction whose associated rate is exponential or immediate.

archi_type	⟨name and numeric parameters⟩
archi_elem_types	⟨architectural element types: behaviors and interactions⟩
archi_topology	
archi_elem_instances	⟨architectural element instances⟩
archi_interactions	⟨architectural interactions⟩
archi_attachments	⟨architectural attachments⟩
end	

Table 1. Structure of an Æmilia textual description

We now illustrate the textual notation of Æmilia by means of an example concerning a pipe-filter system. The system is composed of three identical filters and one pipe. Each filter acts as a service center of capacity two that is subject to failures and subsequent repairs, which is characterized by a service rate σ, a failure rate ϕ, and a repair rate ρ. For each item processed by the upstream filter, the pipe instantaneously forwards it to one of the two downstream filters according to the availability of free positions in their buffers. If both have free positions, the choice is resolved probabilistically based on $p_{routing}$. The Æmilia textual description is provided in Table 2. [2] Such a description establishes that there are three instances F_0, F_1, and F_2 of *FilterT* as well as one instance P of *PipeT*, connected in such a way that the items flow from F_0 to P and from P to F_1 or F_2. It is worth observing that the system components are clearly elucidated and easily connected to each other, and that the numeric parameters allow for a good degree of specification reuse: e.g., the behavior of the filters is defined only once. Additionally, the *accept_item* input interaction of F_0 and the *serve_item* output interactions of F_1 and F_2 are declared as being architectural. Therefore, they can be used for hierchical modeling, e.g. to describe a client-server system where the server structure is like the pipe-filter organization above.

Æmilia comes equipped with a graphical notation as well, in order to provide a visual help during the architectural design of complex systems. Such a graphical notation is based on flow graphs [60]. In a flow graph representing an architec-

[2] Wherever omitted, priority levels and weights are taken to be 1.

archi_type	$\mathit{PipeFilter}(\textbf{exp\_rate}\ \sigma_0, \sigma_1, \sigma_2, \phi_0, \phi_1, \phi_2, \rho_0, \rho_1, \rho_2;$ $\textbf{weight}\ p_{routing})$
archi_elem_types	
elem_type	$\mathit{FilterT}(\textbf{exp\_rate}\ \sigma, \phi, \rho)$
behavior	$\mathit{Filter} \overset{\Delta}{=} {<}\mathit{accept\_item}, *{>}.\mathit{Filter}' +$ ${<}\mathit{fail}, \phi{>}.{<}\mathit{repair}, \rho{>}.\mathit{Filter}$
	$\mathit{Filter}' \overset{\Delta}{=} {<}\mathit{accept\_item}, *{>}.\mathit{Filter}'' +$ ${<}\mathit{serve\_item}, \sigma{>}.\mathit{Filter} +$ ${<}\mathit{fail}, \phi{>}.{<}\mathit{repair}, \rho{>}.\mathit{Filter}'$
	$\mathit{Filter}'' \overset{\Delta}{=} {<}\mathit{serve\_item}, \sigma{>}.\mathit{Filter}' +$ ${<}\mathit{fail}, \phi{>}.{<}\mathit{repair}, \rho{>}.\mathit{Filter}''$
interactions	**input** *accept_item*
	output *serve_item*
elem_type	$\mathit{PipeT}(\textbf{weight}\ p)$
behavior	$\mathit{Pipe} \overset{\Delta}{=} {<}\mathit{accept\_item}, *{>}.({<}\mathit{forward\_item}_1, \infty_{1,p}{>}.\mathit{Pipe} +$ ${<}\mathit{forward\_item}_2, \infty_{1,1-p}{>}.\mathit{Pipe})$
interactions	**input** *accept_item*
	output $\mathit{forward\_item}_1, \mathit{forward\_item}_2$
archi_topology	
archi_elem_instances	$F_0 : \mathit{FilterT}(\sigma_0, \phi_0, \rho_0)$
	$F_1 : \mathit{FilterT}(\sigma_1, \phi_1, \rho_1)$
	$F_2 : \mathit{FilterT}(\sigma_2, \phi_2, \rho_2)$
	$P : \mathit{PipeT}(p_{routing})$
archi_interactions	**input** $F_0.\mathit{accept\_item}$
	output $F_1.\mathit{serve\_item}, F_2.\mathit{serve\_item}$
archi_attachments	**from** $F_0.\mathit{serve\_item}$ **to** $P.\mathit{accept\_item}$
	from $P.\mathit{forward\_item}_1$ **to** $F_1.\mathit{accept\_item}$
	from $P.\mathit{forward\_item}_2$ **to** $F_2.\mathit{accept\_item}$
end	

Table 2. Textual description of *PipeFilter*

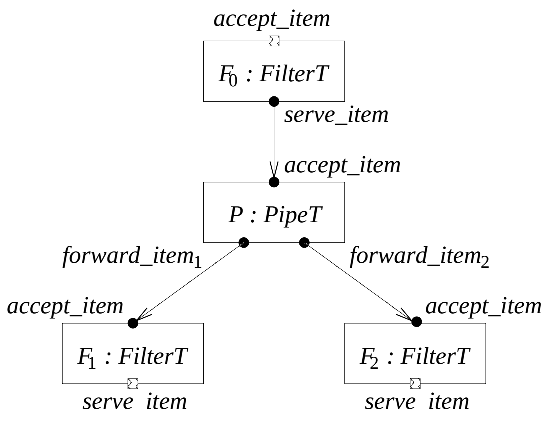


Fig. 1. Flow graph of *PipeFilter*

tural description in Æmilia, the boxes denote the AEIs, the black circles denote the local interactions, the white squares denote the architectural interactions, and the directed edges denote the attachments. As an example, the architectural type *PipeFilter* can be pictorially represented through the flow graph of Fig. 1. From a methodological viewpoint, when modeling an architectural type with Æmilia, it is convenient to start with the flow graph representation of the architectural type and then to textually specify the behavior of each AET.

3.2 Translation Semantics

The semantics of an Æmilia specification is given by translation into EMPA$_{\mathrm{gr}}$. While only the dynamic operators (action prefix and alternative composition) of EMPA$_{\mathrm{gr}}$ can be used in the syntax of an Æmilia specification, the more complicated static operators (functional abstraction, functional relabeling, and parallel composition) of EMPA$_{\mathrm{gr}}$ are transparently used in the semantics of an Æmilia specification. The translation into EMPA$_{\mathrm{gr}}$ is accomplished in two steps.

In the first step, the semantics of all the instances of each AET is defined to be the behavior of the AET projected onto its interactions. Such a projected behavior is obtained from the family of sequential EMPA$_{\mathrm{gr}}$ terms representing the behavior of the AET by applying a functional abstraction operator on all the actions that are not interactions. In this way, we abstract from all the internal details of the behavior of the instances of the AET. For the pipe-filter system of Table 2 we have

$$\begin{array}{rcl} [\![\mathit{FilterT}]\!] = [\![F_0]\!] = [\![F_1]\!] = [\![F_2]\!] & = & \mathit{Filter}/\{\mathit{fail}, \mathit{repair}\} \\ [\![\mathit{PipeT}]\!] = \qquad [\![P]\!] \qquad & = & \mathit{Pipe} \end{array}$$

thus abstracting from the internal activities *fail* and *repair*.

In the second step, the semantics of an architectural type is obtained by composing in parallel the semantics of its AEIs according to the specified attachments. Recalled that the parallel composition operator is left associative, for the pipe-filter system we have

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

The use of the functional relabeling operator is necessary to make the AEIs 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 only equally typed actions to synchronize, in $[\![\mathit{PipeFilter}]\!]$ each *serve_item* action executed by $[\![F_0]\!]$ and each *accept_item* action executed by $[\![P]\!]$ is relabeled to an action with the same type a. In order to avoid interferences, it is important that a be a fresh action type, i.e. an action type occurring neither in $[\![F_0]\!]$ nor in $[\![P]\!]$. Then a synchronization on a is forced between the relabeled versions of $[\![F_0]\!]$ and $[\![P]\!]$ by means of operator $\parallel_{\{a,a_1,a_2\}}$. It is worth reminding that the transformation of *PipeFilter* into $[\![\mathit{PipeFilter}]\!]$, which can be analyzed through the techniques mentioned in Sect 2, is completely transparent to the designer.

The interested reader is referred to [9, 16] for a formal definition of the translation semantics.

3.3 Architectural Checks

Æmilia is equipped with some architectural checks that the designer can use to verify the well formedness of the architectural types and, in case a mismatch is detected, to identify the components that cause it. Most of such checks are based on the weak bisimulation equivalence [60], which captures the ability of the functional semantic models of two terms to simulate each other behaviors up to internal actions.

The first two checks take care of verifying whether the deadlock free AEIs of an architectural type fit together well, i.e. do not lead to system blocks. The first check (compatibility) is concerned with architectural types whose topology is acyclic. For an acyclic architectural type, if we take an AEI K and we consider all the AEIs $C_1, \ldots, C_n$ attached to it, we can observe that they form a star topology whose center is K, as the absence of cycles prevents any two AEIs among $C_1, \ldots, C_n$ from communicating via an AEI different from K. It can easily be recognized that an acyclic architectural type is just a composition of star topologies. An efficient compatibility check based on the weak bisimulation equivalence (together with a simple constraint on action priorities) ensures the absence of deadlock within a star topology whose center K is deadlock free, and this check scales to the whole acyclic architectural type. The basic condition to check is that every C_i is compatible with K, i.e. the functional semantics of their parallel composition is weakly bisimulation equivalent to the functional semantics of K itself. Intuitively, this means that attaching C_i to K does not alter the behavior of K, i.e. K is designed in such a way that it suitably coordinates with C_i.

Since the compatibility check is not sufficient for cyclic architectural types, the second check (interoperability) deals with cycles. A suitable interoperability check based on the weak bisimulation equivalence (together with a simple constraint on action priorities) ensures the absence of deadlock within a cycle $C_1, \ldots, C_n$ of AEIs in the case that at least one of such AEIs is deadlock free. The basic condition to check is that at least one deadlock free C_i interoperates with the other AEIs in the cycle, i.e. the functional semantics of the parallel composition of the AEIs in the cycle projected on the interactions with C_i only is weakly bisimulation equivalent to the functional semantics of C_i. Intuitively, this means that inserting C_i into the cycle does not alter the behavior of C_i, i.e. that the behavior of the cycle assumed by C_i matches the actual behavior of the cycle. In the case in which no deadlock free AEI is found in the cycle that interoperates with the other AEIs, a loop shrinking procedure can be used to single out the AEIs in the cycle responsible for the deadlock.

On the performance side, there is a third check to detect architectural mismatches resulting in performance underspecification. This check (performance closure) ensures that the performance semantic model underlying an architectural type exists in the form of a CTMC or DTMC. In order for an architectural type to be performance closed, the basic condition to check is that no AET behavior contains a passive action whose type is not an interaction, and that every set of attached local interactions contains one interaction whose associated rate is exponential or immediate.

We conclude by referring the interested reader to [9, 16] for a precise definition and examples of application of the architectural checks outlined in this section.

3.4 Families of Architectures and Hierarchical Modeling

An Æmilia description represents a family of architectures called an architectural type. An architectural type is an intermediate abstraction between a single architecture and an architectural style [66]. An important goal of the software architecture discipline is the creation of an established and shared understanding of the common forms of software design. Starting from the user requirements, the designer should be able to identify a suitable organizational style, in order to capitalize on codified principles and experience to specify, analyze, plan, and monitor the construction of a system with high levels of efficiency and confidence. An architectural style defines a family of systems having a common vocabulary of components as well as a common topology and set of contraints on the interactions among the components. As examples of architectural styles we mention main program-subroutines, pipe-filter, client-server, and the layered organization. Since an architectural style encompasses an entire family of software systems, it is desirable to formalize the concept of architectural style both to have a precise definition of the system family and to study the architectural properties common to all the systems of the family. This is not a trivial task because there are at least two degrees of freedom: variability of the component topology and variability of the component internal behavior.

An architectural type is an approximation of an architectural style, where the component topology and the component internal behavior can vary from instance to instance of the architectural type in a controlled way, which preserves the architectural checks. More precisely, all the instances of an architectural type must have the same observable functional behavior and conforming topologies, while the internal behavior and the performance characteristics can freely vary. An instance of an architectural type can be obtained by invoking the architectural type and passing actual AETs preserving the observable functional behavior of the formal AETs, an actual topology (actual AEIs, actual architectural interactions, and actual attachments) that conforms to the formal topology, actual names for the architectural interactions, and actual values for the numeric parameters.

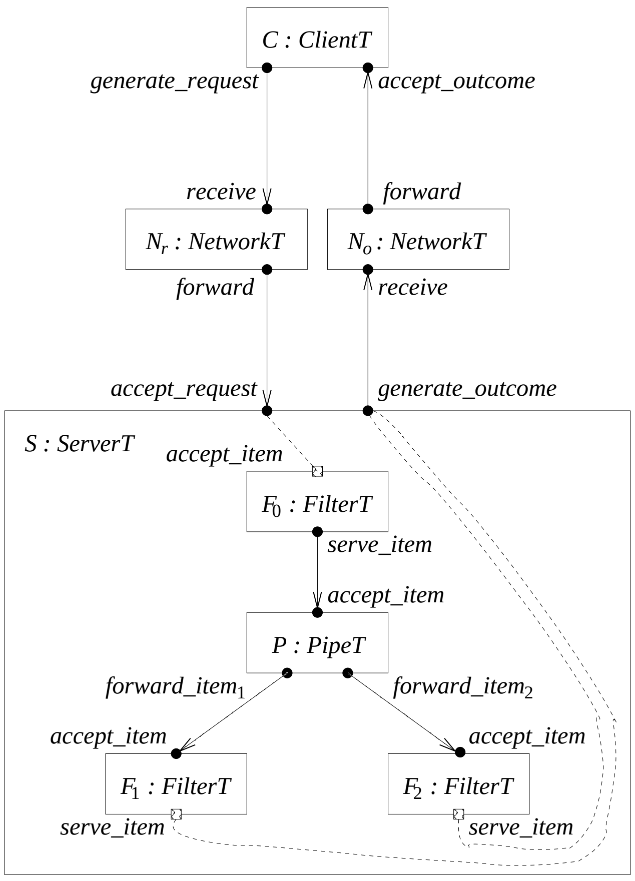


Fig. 2. Flow graph of *ClientServer*

```
archi_type                 $ClientServer(\textbf{exp\_rate}\ \lambda, \delta_r, \delta_o,$
                                       $\sigma_0, \sigma_1, \sigma_2, \phi_0, \phi_1, \phi_2, \rho_0, \rho_1, \rho_2;$
                                $\textbf{weight}\ p_{routing})$

  archi_elem_types

    elem_type              $ClientT(\textbf{exp\_rate}\ \lambda)$
      behavior             $Client \triangleq <generate\_request, \lambda>.$
                                  $<accept\_outcome, *>.Client$
      interactions         output $generate\_request$
                           input $accept\_outcome$

    elem_type              $NetworkT(\textbf{exp\_rate}\ \delta)$
      behavior             $Network \triangleq <receive, *>.<forward, \delta>.Network$
      interactions         input $receive$
                           output $forward$

    elem_type              $ServerT(\textbf{exp\_rate}\ \sigma_0, \sigma_1, \sigma_2, \phi_0, \phi_1, \phi_2, \rho_0, \rho_1, \rho_2;$
                                  $\textbf{weight}\ p_{routing})$
      behavior             $Server \triangleq PipeFilter(;$   /* actual AETs */
                                                ;   /* actual AEIs */
                                                ;   /* actual arch. interactions */
                                                ;   /* actual attachments */
                                                $accept\_request,$
                                                 $generate\_outcome,$
                                                 $generate\_outcome;$
                                                $\sigma_0, \sigma_1, \sigma_2, \phi_0, \phi_1, \phi_2, \rho_0, \rho_1, \rho_2,$
                                                 $p_{routing})$
      interactions         input $accept\_request$
                           output $generate\_outcome$

  archi_topology
       archi_elem_instances $C : ClientT(\lambda)$
                           $N_r : NetworkT(\delta_r)$
                           $N_o : NetworkT(\delta_o)$
                           $S : ServerT(\sigma_0, \sigma_1, \sigma_2, \phi_0, \phi_1, \phi_2, \rho_0, \rho_1, \rho_2, p_{routing})$
       archi_interactions
       archi_attachments   from $C.generate\_request$ to $N_r.receive$
                           from $N_r.forward$ to $S.accept\_request$
                           from $S.generate\_outcome$ to $N_o.receive$
                           from $N_o.forward$ to $C.accept\_outcome$

end
```

Table 3. Textual description of *ClientServer*

The simplest form of architectural invocation is the one in which the actual parameters coincide with the formal ones, in which case the actual parameters are omitted for the sake of conciseness. The possibility of defining the behavior of an AET through an architectural invocation as well as declaring architectural interactions can be exploited to model a system architecture in a hierarchical way. As an example, consider the pipe-filter organization of Table 2 and suppose that it is the architecture of the server of a client-server system. The flow graph description of the resulting client-server system is depicted in Fig. 2, while its textual description is reported in Table 3. The client description is parametrized w.r.t. the request generation rate λ, while the communication link description is parametrized w.r.t. the communication speed δ. As can be observed, the behavior of the server is defined through an invocation of the previously defined architectural type *PipeFilter*, where the actual names *accept_request*, *generate_outcome*, and *generate_outcome* substitute for the formal architectural interactions $F_0.accept\_item$, $F_1.serve\_item$, and $F_2.serve\_item$, respectively.

A more complex form of architectural invocation is the one in which actual AETs are passed that are different from the corresponding formal AETs. In this case, we have to make sure that the actual AETs preserves the functional behavior determined by the formal ones. To this purpose, Æmilia is endowed with an efficient behavioral conformity check based on the weak bisimulation equivalence (together with a simple constraint on action rates) to verify whether an architectural type invocation conforms to an architectural type definition, in the sense that the architectural type invocation and the architectural type definition have the same observable functional semantics up to some relabeling. The basic condition to check is that the functional semantics of each actual AET is weakly bisimulation equivalent to the functional semantics of the corresponding formal AET up to some relabeling. This behavioral conformity check ensures that all the correct instances of an architectural type possess the same compatibility, interoperability, and performance closure properties. In other words, the outcome of the application of the compatibility, interoperability, and performance closure checks to the definition of an architectural type scales to all the behaviorally conforming invocations of the architectural type.

The most complete form of architectural invocation is the one in which both actual AETs and an actual topology are passed that are different from the corresponding formal AETs and formal topology, respectively. In this case, we have to additionally make sure that the actual topology conforms to the formal topology. There are three kinds of admitted topological extensions, all of which preserve the compatibility, interoperability, and performance closure properties under some general conditions.

The first kind of topological extension is given by the extensible and/or connections. As an example, consider the client-server system of Table 3. Every instance of such an architectural type can admit a single client and a single server, whereas it would be useful to allow for an arbitrary number of clients (to be instantiated when invoking the architectural type) that can connect to

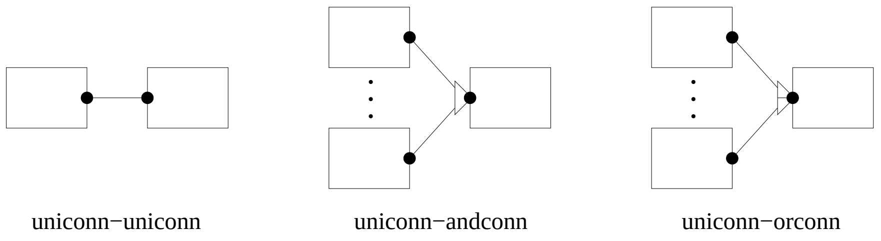


Fig. 3. Legal attachments in case of extensible and/or connections

the server. From the syntactical viewpoint, the extensible and/or connections are introduced in Æmilia by further typing the interactions of the AETs. Besides the input/output qualification, the interactions are classified as uniconn, andconn, and orconn, with only the three types of attachments shown in Fig. 3 considered legal. A uniconn interaction is an interaction to which a single AEI can be attached; e.g., all the interactions of *ClientServer* are of this type. An andconn interaction is an interaction to which a variable number of AEIs can be attached, such that all the attached AEIs must synchronize when that interaction takes place; e.g., a broadcast transmission. An orconn interaction is an interaction to which a variable number of AEIs can be attached, such that only one of the attached AEIs must synchronize when that interaction takes place; e.g., a client-server system with several clients. Every output orconn interaction must be declared to depend on one input orconn interaction, with the occurrences of the two interactions alternating in the behavior of the AET that contains them. On the semantic side, the treatment of uniconn and orconn interactions is trivial. Instead, every occurrence of an input orconn interaction must be replaced by a choice among as many indexed occurrences of that interaction as there are attached AEIs, while every occurrence of an output orconn interaction must be augmented with the same index given to the occurrence of the preceding input orconn interaction on which it depends. Such modifications, which are completely transparent to the designer, are necessary to reflect the fact that an orconn interaction expresses a choice among different attached AEIs whenever the interaction takes place.

The second kind of topological extension is given by the exogenous one. As an example, consider the pipe-filter system of Table 2. Every instance of such an architectural type can admit a single pipe connected to one upstream filter and two downstream filters, whereas it would be desirable to be able to express by means of that architectural type any pipe-filter system with an arbitrary number of filters and pipes, such that every pipe is connected to one upstream filter and two downstream filters. E.g., the flow graph in Fig. 4 should be considered as a legal extension of the flow graph in Fig. 1. The idea behind the exogenous extensions is that, since the architectural interactions of an architectural type are the frontier of the whole architectural type, it is reasonable to extend the architectural type at some of its architectural interactions with instances of the already defined AETs, in a way that follows the prescribed topology.

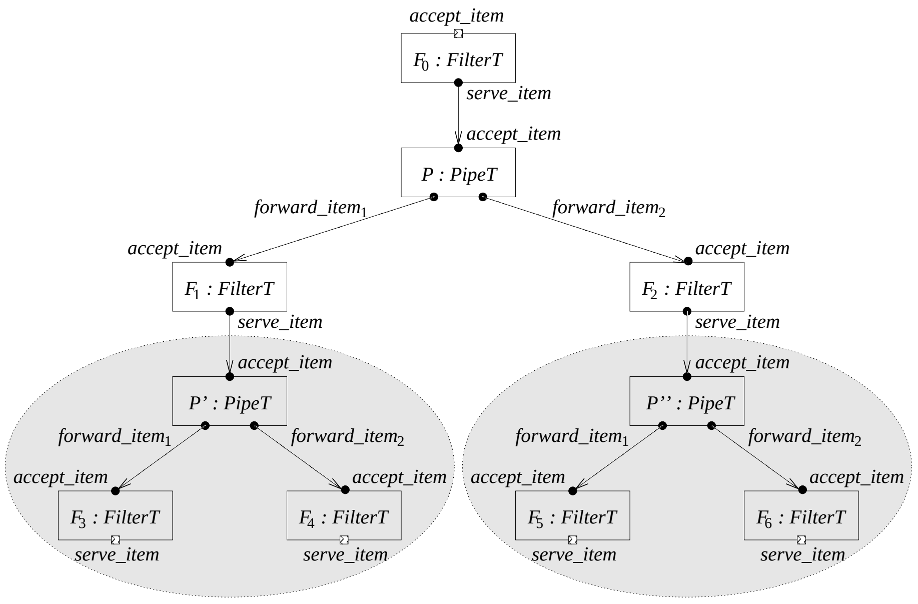


Fig. 4. Flow graph of an exogenous extension of *PipeFilter*

The third kind of topological extension is given by the endogenous one. As an example, consider the Æmilia description of a ring of stations each following the same protocol: wait for a message from the previous station in the ring, process the received message, and send the processed message to the next station in the ring. Since such a protocol guarantees that only one station can transmit at a given instant, the protocol can be considered as an abstraction of the IEEE 802.5 standard medium access control protocol for local area networks known as token ring. One of the stations is designated to be the initial one, in the sense that it is the first station allowed to send a message. Suppose that the Æmilia description declares one instance of the initial station and three instances of the normal station. Every instance of the architectural type, say *Ring*, can thus admit a single initial station and three normal stations connected to form a ring, whereas it would be desirable to be able to express by means of that architectural type any ring system with an arbitrary number of normal stations. E.g., the flow graph in Fig. 5 should be considered as a legal extension of the architectural type *Ring*. The idea behind the endogenous extensions is that of replacing a set of AEIs with a set of new instances of the already defined AETs, in a way that follows the prescribed topology. In this case, we consider the frontier of the architectural type w.r.t. one of the replaced AEIs to be the set of interactions previously attached to the local interactions of the replaced AEI. On the other hand, all the replacing AEIs that will be attached to the frontier of the architectural type w.r.t. one of the replaced AEIs must be of the same type as the replaced AEI.

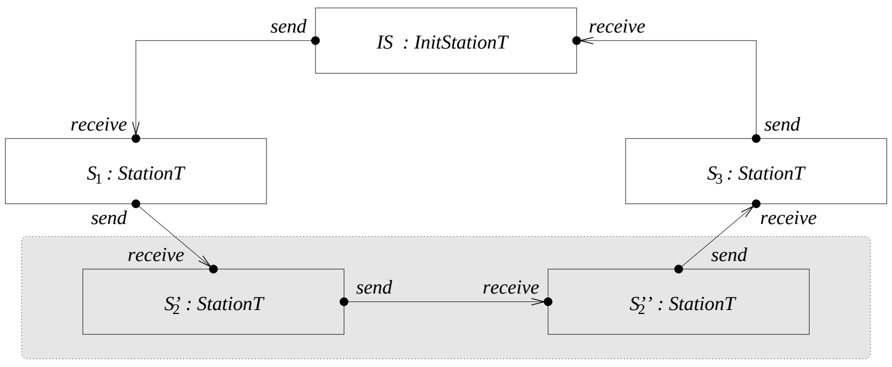


Fig. 5. Flow graph of an endogenous extension of *Ring*

We conclude by referring the interested reader to [9, 16, 19, 20] for a precise definition of the behavioral and topological conformity checks outlined in this section.

4 Conclusion

In this paper we have recalled the basic notions and the main achievements in the field of SPA and we have stressed its current transformation into a fully fledged ADL for the compositional, graphical, hierarchical and controlled modeling of complex systems as well as their functional verification and performance evaluation. Such a transformation eases the modeling process and provides an added value given by some architectural checks for detecting deadlock as well as performance underspecification, which scale over families of architectures.

Concerning future work in the area of SPA based ADLs, first of all we mention the importance of devising additional architectural checks on the performance side, that provide diagnostic information like in the case of the compatibility and interoperability checks. At the architectural level of design, it is extremely useful to be able to reinterpret the performance results in terms of components and their interactions. In order to achieve that, the performance must be calculated not on a flat model like a MC, but on a model that maintains some correspondence with the system structure, so that there is the possibility to localize bottlenecks. Some work in this direction can be found in [9], where Æmilia descriptions are translated into queueing network models.

Furthemore, SPA based ADLs should be viewed in the context of the whole software life cycle. A link should be established from higher level notations like UML, where requirements are expressed in a less formal way, as well as to object oriented programming languages, aiming at the automatic generation of code that possesses the functional and performance properties formally proved at the architectural level.

References

1. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, *"Modelling with Generalized Stochastic Petri Nets"*, John Wiley & Sons, 1995
2. A. Aldini, M. Bernardo, R. Gorrieri, *"An Algebraic Model for Evaluating the Performance of an ATM Switch with Explicit Rate Marking"*, in Proc. of the *7th Int. Workshop on Process Algebra and Performance Modelling (PAPM 1999)*, Prensas Universitarias de Zaragoza, pp. 119-138, Zaragoza (Spain), 1999
3. A. Aldini, M. Bernardo, R. Gorrieri, M. Roccetti, *"Comparing the QoS of Internet Audio Mechanisms via Formal Methods"*, in ACM Trans. on Modeling and Computer Simulation 11:1-42, 2001
4. R. Allen, D. Garlan, *"A Formal Basis for Architectural Connection"*, in ACM Trans. on Software Engineering and Methodology 6:213-249, 1997
5. J.C.M. Baeten, W.P. Weijland, *"Process Algebra"*, Cambridge University Press, 1990
6. C. Baier, B. Haverkort, H. Hermanns, J.-P. Katoen, *"On the Logical Characterisation of Performability Properties"*, in Proc. of the *27th Int. Coll. on Automata, Languages and Programming (ICALP 2000)*, LNCS 1853:780-792, Geneve (Switzerland), 2000
7. C. Baier, B. Haverkort, H. Hermanns, J.-P. Katoen, *"Model Checking Continuous-Time Markov Chains by Transient Analysis"*, in Proc. of the *12th Int. Conf. on Computer Aided Verification (CAV 2000)*, LNCS 1855:358-372, Chicago (IL), 2000
8. C. Baier, J.-P. Katoen, H. Hermanns, *"Approximate Symbolic Model Checking of Continuous Time Markov Chains"*, in Proc. of the *10th Int. Conf. on Concurrency Theory (CONCUR 1999)*, LNCS 1664:146-162, Eindhoven (The Netherlands), 1999
9. S. Balsamo, M. Bernardo, M. Simeoni, *"Combining Stochastic Process Algebras and Queueing Networks for Software Architecture Analysis"*, to appear in Proc. of the *3rd Int. Workshop on Software and Performance (WOSP 2002)*, Rome (Italy), 2002
10. M. Bernardo, *"Theory and Application of Extended Markovian Process Algebra"*, Ph.D. Thesis, University of Bologna (Italy), 1999
11. M. Bernardo, *"A Simulation Analysis of Dynamic Server Selection Algorithms for Replicated Web Services"*, in Proc. of the *9th Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2001)*, IEEE-CS Press, pp. 371-378, Cincinnati (OH), 2001
12. M. Bernardo, M. Bravetti, *"Reward Based Congruences: Can We Aggregate More?"*, in Proc. of the *1st Joint Int. Workshop on Process Algebra and Performance Modelling and Probabilistic Methods in Verification (PAPM/PROBMIV 2001)*, LNCS 2165:136-151, Aachen (Germany), 2001
13. M. Bernardo, M. Bravetti, *"Performance Measure Sensitive Congruences for Markovian Process Algebras"*, to appear in Theoretical Computer Science, 2002
14. M. Bernardo, N. Busi, M. Ribaudo, *"Integrating TwoTowers and GreatSPN through a Compact Net Semantics"*, to appear in Performance Evaluation, 2002
15. M. Bernardo, P. Ciancarini, L. Donatiello, *"ÆMPA: A Process Algebraic Description Language for the Performance Analysis of Software Architectures"*, in Proc. of the *2nd Int. Workshop on Software and Performance (WOSP 2000)*, ACM Press, pp. 1-11, Ottawa (Canada), 2000

16. M. Bernardo, P. Ciancarini, L. Donatiello, *"Architecting Software Systems with Process Algebras"*, Tech. Rep. UBLCS-2001-07, University of Bologna (Italy), 2001
17. M. Bernardo, W.R. Cleaveland, *"A Theory of Testing for Markovian Processes"*, in Proc. of the *11th Int. Conf. on Concurrency Theory (CONCUR 2000)*, LNCS 1877:305-319, State College (PA), 2000
18. M. Bernardo, W.R. Cleaveland, W.S. Stewart, *"TwoTowers 1.0 User Manual"*, `http://www.sti.uniurb.it/bernardo/twotowers/`, 2001
19. M. Bernardo, F. Franzè, *"Architectural Types Revisited: Extensible And/Or Connections"*, in Proc. of the *5th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2002)*, LNCS 2306:113-128, Grenoble (France), 2002
20. M. Bernardo, F. Franzè, *"Exogenous and Endogenous Extensions of Architectural Types"*, in Proc. of the *5th Int. Conf. on Coordination Models and Languages (COORDINATION 2002)*, LNCS 2315:40-55, York (UK), 2002
21. M. Bernardo, R. Gorrieri, M. Roccetti, *"Formal Performance Modelling and Evaluation of an Adaptive Mechanism for Packetised Audio over the Internet"*, in Formal Aspects of Computing 10:313-337, 1999
22. H. Bohnenkamp, *"Compositional Solution of Stochastic Process Algebra Models"*, Ph.D. Thesis, RWTH Aachen (Germany), 2001
23. H. Bowman, J.W. Bryans, J. Derrick, *"Analysis of a Multimedia Stream using Stochastic Process Algebra"*, in Proc. of the *6th Int. Workshop on Process Algebra and Performance Modelling (PAPM 1998)*, pp. 51-69, Nice (France), 1998
24. J.T. Bradley, *"Towards Reliable Modelling with Stochastic Process Algebras"*, Ph.D. Thesis, University of Bristol (UK), 1999
25. M. Bravetti, *"Specification and Analysis of Stochastic Real-Time Systems"*, Ph.D. Thesis, University of Bologna (Italy), 2002
26. M. Bravetti, M. Bernardo, *"Compositional Asymmetric Cooperations for Process Algebras with Probabilities, Priorities, and Time"*, in Proc. of the *1st Int. Workshop on Models for Time Critical Systems (MTCS 2000)*, Electronic Notes in Theoretical Computer Science 39(3), State College (PA), 2000
27. P. Buchholz, *"Markovian Process Algebra: Composition and Equivalence"*, in Proc. of the *2nd Int. Workshop on Process Algebra and Performance Modelling (PAPM 1994)*, pp. 11-30, Erlangen (Germany), 1994
28. W.R. Cleaveland, J. Parrow, B. Steffen, *"The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems"*, in ACM Trans. on Programming Languages and Systems 15:36-72, 1993
29. G. Clark, *"Techniques for the Construction and Analysis of Algebraic Performance Models"*, Ph.D. Thesis, University of Edinburgh (UK), 2000
30. E.M. Clarke, O. Grumberg, D.A. Peled, *"Model Checking"*, MIT Press, 1999
31. P. D'Argenio, *"Algebras and Automata for Timed and Stochastic Systems"*, Ph.D. Thesis, University of Twente (The Netherlands), 1999
32. R. De Nicola, M.C.B. Hennessy, *"Testing Equivalences for Processes"*, in Theoretical Computer Science 34:83-133, 1983
33. D. Ferrari, *"Considerations on the Insularity of Performance Evaluation"*, in IEEE Trans. on Software Engineering 12:678-683, 1986
34. S. Gilmore, *"The PEPA Workbench User Manual"*, `http://www.dcs.ed.ac.uk/pepa/tools.html`, 2001
35. S. Gilmore, J. Hillston, D.R.W. Holton, M. Rettelbach, *"Specifications in Stochastic Process Algebra for a Robot Control Problem"*, in Journal of Production Research 34:1065-1080, 1996

36. R.J. van Glabbeek, S.A. Smolka, B. Steffen, *"Reactive, Generative and Stratified Models of Probabilistic Processes"*, in Information and Computation 121:59-80, 1995
37. R.J. van Glabbeek, F.W. Vaandrager, *"Petri Net Models for Algebraic Theories of Concurrency"*, in Proc. of the *Conf. on Parallel Architectures and Languages Europe (PARLE 1987)*, LNCS 259:224-242, Eindhoven (The Netherlands), 1987
38. N. Götz, *"Stochastische Prozeßalgebren – Integration von funktionalem Entwurf und Leistungsbewertung Verteilter Systeme"*, Ph.D. Thesis, University of Erlangen (Germany), 1994
39. P.G. Harrison, J. Hillston, *"Exploiting Quasi-Reversible Structures in Markovian Process Algebra Models"*, in Computer Journal 38:510-520, 1995
40. H. Hermanns, *"Interactive Markov Chains"*, Ph.D. Thesis, University of Erlangen (Germany), 1998
41. H. Hermanns, U. Herzog, J. Hillston, V. Mertsiotakis, M. Rettelbach, *"Stochastic Process Algebras: Integrating Qualitative and Quantitative Modelling"*, Tech. Rep. 11/94, University of Erlangen (Germany), 1994
42. H. Hermanns, U. Herzog, V. Mertsiotakis, *"Stochastic Process Algebras as a Tool for Performance and Dependability Modelling"*, in Proc. of the *1st IEEE Int. Computer Performance and Dependability Symp. (IPDS 1995)*, IEEE-CS Press, pp. 102-111, Erlangen (Germany), 1995
43. H. Hermanns, J.-P. Katoen, *"Automated Compositional Markov Chain Generation for a Plain-Old Telephone System"*, in Science of Computer Programming 36:97-127, 2000
44. H. Hermanns, J. Meyer-Kayser, M. Siegle, *"Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains"*, in Proc. of the *3rd Int. Workshop on the Numerical Solution of Markov Chains (NSMC 1999)*, Zaragoza (Spain), 1999
45. H. Hermanns, M. Rettelbach, *"Syntax, Semantics, Equivalences, and Axioms for MTIPP"*, in Proc. of the *2nd Int. Workshop on Process Algebra and Performance Modelling (PAPM 1994)*, pp. 71-87, Erlangen (Germany), 1994
46. U. Herzog, *"Formal Description, Time and Performance Analysis – A Framework"*, in *Entwurf und Betrieb verteilter Systeme*, Informatik Fachberichte 264, Springer, 1990
47. U. Herzog, *"EXL: Syntax, Semantics and Examples"*, Tech. Rep. 16/90, University of Erlangen (Germany), 1990
48. J. Hillston, *"A Compositional Approach to Performance Modelling"*, Cambridge University Press, 1996
49. J. Hillston, N. Thomas, *"Product Form Solution for a Class of PEPA Models"*, in Performance Evaluation 35:171-192, 1999
50. C.A.R. Hoare, *"Communicating Sequential Processes"*, Prentice Hall, 1985
51. D.R.W. Holton, *"A PEPA Specification of an Industrial Production Cell"*, in Computer Journal 38:542-551, 1995
52. R.A. Howard, *"Dynamic Probabilistic Systems"*, John Wiley & Sons, 1971
53. K. Kanani, *"A Unified Framework for Systematic Quantitative and Qualitative Analysis of Communicating Systems"*, Ph.D. Thesis, Imperial College (UK), 1998
54. J.-P. Katoen *"Quantitative and Qualitative Extensions of Event Structures"*, Ph.D. Thesis, University of Twente (The Netherlands), 1996
55. U. Klehmet, V. Mertsiotakis, *"TIPPtool – User's Guide"*, `http://www7.informatik.uni-erlangen.de/tipp/tool.html`, 1998
56. L. Kleinrock, *"Queueing Systems"*, John Wiley & Sons, 1975

57. K.G. Larsen, A. Skou, *“Bisimulation through Probabilistic Testing”*, in Information and Computation 94:1-28, 1991
58. S.S. Lavenberg editor, *“Computer Performance Modeling Handbook”*, Academic Press, 1983
59. V. Mertsiotakis, *“Approximate Analysis Methods for Stochastic Process Algebras”*, Ph.D. Thesis, University of Erlangen (Germany), 1998
60. R. Milner, *“Communication and Concurrency”*, Prentice Hall, 1989
61. D.E. Perry, A.L. Wolf, *“Foundations for the Study of Software Architecture”*, in ACM SIGSOFT Software Engineering Notes 17:40-52, 1992
62. M. Rettelbach, *“Stochastische Prozeßalgebren mit zeitlosen Aktivitäten und probabilistischen Verzweigungen”*, Ph.D. Thesis, University of Erlangen (Germany), 1996
63. M. Ribaudo, *“On the Relationship between Stochastic Process Algebras and Stochastic Petri Nets”*, Ph.D. Thesis, University of Torino (Italy), 1995
64. P. Schweitzer, *“Aggregation Methods for Large Markov Chains”*, in Mathematical Computer Performance and Reliability, North Holland, pp. 275-286, 1984
65. M. Sereno, *“Towards a Product Form Solution for Stochastic Process Algebras”*, in Computer Journal 38:622-632, 1995
66. M. Shaw, D. Garlan, *“Software Architecture: Perspectives on an Emerging Discipline”*, Prentice Hall, 1996
67. M. Siegle, *“Beschreibung und Analyse von Markovmodellen mit grossem Zustandsraum”*, Ph.D. Thesis, University of Erlangen (Germany), 1995
68. C.U. Smith, *“Performance Engineering of Software Systems”*, Addison-Wesley, 1990
69. W.J. Stewart, *“Introduction to the Numerical Solution of Markov Chains”*, Princeton University Press, 1994
70. B. Strulo, *“Process Algebra for Discrete Event Simulation”*, Ph.D. Thesis, Imperial College (UK), 1994