QoS-Based Model Driven Assessment of Adaptive Reactive Systems

Diego Perez-Palacin
Dpto. de Informática e
Ingeniería de Sistemas
Universidad de Zaragoza,
Zaragoza, Spain
diegop@unizar.es

Raffaela Mirandola
Politecnico di Milano
Dipartimento di Elettronica
e Informazione
Milano, Italy
mirandola@elet.polimi.it

José Merseguer
Dpto. de Informática e
Ingeniería de Sistemas
Universidad de Zaragoza,
Zaragoza, Spain
jmerse@unizar.es

Vincenzo Grassi
Università di Roma Tor Vergata
Dipartimento di Informatica,
Sistemi e Produzione
Roma, Italy
vgrassi@info.uniroma2.it

***Abstract*—Adaptable reactive software systems continuously interact with their environment responding to external stimuli and triggering events that may be perceived by their users. Designing and maintaining such systems is a challenging task. A key issue to be faced concerns the assessment of their effectiveness, in terms of the ability to meet their required Quality of Service. This paper introduces a model driven approach to support this assessment, with a focus on performance and dependability attributes. Our approach takes advantage from an existing intermediate modeling language and introduces the necessary extensions to cope with reactive systems. The presented model driven framework exploits the idea of defining a model transformation chain that maps a design oriented model of the system to an analysis oriented model that lends itself to the application of a suitable analysis methodology. We identify some key concepts that should be present in the design model of an adaptable reactive system, and show how to devise a transformation from such a model to a target analysis models.**

I. Introduction

Dealing with reactive software systems [14] that maintain ongoing interactions with their environment being able to react to external stimuli is becoming an important feature for software systems, due to the emergence of more and more classes of applications that are highly complex and distributed, and operate in heterogeneous and rapidly changing environments like those from the mobile and pervasive computing domains [9].

Reactive systems cannot be described in terms of a single function that maps inputs to outputs. The response that a reactive system provides to an input event, for example, depends on the current state of the system, which, in turn, is a function of the already received inputs [14]. Additionally, the system may offer services or behaviors that can be invoked, also by third parties, in the form of events to which the system reacts. Some examples illustrate them. Real-time systems, after time-outs, awake tasks by sending events, so they react to accomplish their works. An agent may be requested to perform some behavior by means of a call to an event it offers. Service software reacts to incoming calls and manages them to offer adequate responses.

At this regard, we identified three necessary and intrinsic abilities of systems that show reactive behavior:

(a) Their ability to suspend their execution until the eventual reception of a signal or event.
(b) Their ability to accept and manage a signal/event while they are doing some computing work.
(c) Their ability to send signals/events to their components or to another systems for them to react.

Designing and maintaining such systems is a challenging task. A key issue to be faced concerns the assessment of their effectiveness, in terms of the ability to meet their functional and non-functional requirements concerning the delivered quality of service (QoS). In this respect, our goal is to support the design and management of reactive software systems by means of the model-based analysis of their effectiveness, focusing in particular on their ability to meet non-functional requirements related to performance and dependability attributes.

Model-Driven Development (MDD) techniques typically focus on a transformation path, supported by automatic transformation tools, from high level to platform specific models (down to the executable code) of a software system [4], [5], [17]. Modeling frameworks for dynamically changing software systems have been already proposed. Some of them are mainly targeted to the analysis of functional requirements [3], [8], [15], [18], and hence are not suitable for the effectiveness analysis of such systems with respect to performance or dependability.

The idea of exploiting MDD methodologies for QoS assessment has emerged in recent years (see, for example, [4], [6], [19] and papers in [1]). Indeed, the construction of a QoS analysis model can be seen as a special type of model transformation whose source is a design oriented model of the system (produced during the design process by the system designers), while the target is a suitable analysis oriented model, which lends itself to the application of sound analysis methodologies. Existing MDD-based methodologies for the generation of QoS analysis models do not consider the modeling of adaptable reactive systems. Moreover, they often devise the transformation path as a single step transformation from the source design oriented model to the target analysis oriented model. This single step transformation could be excessively complex, for several reasons: the large semantic gap between the source and target models, the different notations that could be used in the source model, and the different target notations one could be interested in, to support different kinds of analysis (e.g., queueing networks, Petri Nets, Markov processes).

To face these problems, bridge models expressed in some suitable intermediate language has been already proposed in the literature to support the generation of analysis oriented models from design oriented models [11], [22]. KLAPER intermediate language [11], for example is also used in the Q-ImPrESS european project [23] which aims at bringing service orientation to different application domains guaranteeing end-to-end quality of service. In this paper we take advantage of this research and use a two-step transformation path from design oriented to analysis oriented models centered around the construction of a bridge model expressed in the D-KLAPER intermediate modeling language. To this end D-KLAPER supports the abstract and simplified representation of concepts we may expect be expressed in the source design oriented model of an adaptable reactive system.

A positive consequence of this splitting centered around a bridge model is that it facilitates the re-use of work already done for one of the two parts. Indeed, given a particular notation used for the design of adaptable systems (e.g., a notation based on a suitable customization of UML [16]), the QoS assessment of models expressed in this notation can take advantage of an already defined transformation from bridge models to some kind of analysis model: what remains to be done is the (presumably) simpler transformation from the source design model to the bridge model, rather than the more complex thorough transformation from the source model to the analysis model. Similarly, once a transformation from a specific kind of design model to the bridge model has been defined, the set of QoS analysis methodologies that can be used (e.g., analytic, or simulation-based) can be extended by simply defining a new transformation from the bridge model to a new kind of analysis model.

This paper builds on and extends results presented in [10], [11], [12]. Specifically, we extend the intermediate modeling language presented there with a new simple feature aimed at modeling the specific aspects of reactive systems. The proposed extension (illustrated in Section II) follows the underlying philosophy of this intermediate language definition maintaining it as minimal as possible.

The paper is organized as follows. Section II presents the core concepts of D-KLAPER and the proposed extension. In Section III, we describe the proposed transformation path and we show how this notation can be used to model reactive systems. In Section IV, we show trough a simple example of adaptable reactive system the practical application of the presented ideas. Finally, Section V concludes the paper.

II. The D-KLAPER intermediate Model

D-KLAPER [12] is defined as a MOF (Meta-Object Facility) compliant metamodel, where MOF is the metamodeling framework proposed by the Object Management Group (OMG) for the management of models and their transformations within the MDD approach to software development [17]. We point out that D-KLAPER is not intended to be directly used by system designers. Indeed, in the modeling framework we envisage, D-KLAPER plays a role analogous to that played by the bytecode language in a Java environment. Hence, D-KLAPER provides a purposely minimal set of elementary and abstract concepts and notations. More expressive concepts and notations used by system designers to build their models should then be mapped to D-KLAPER, with the support of automatic model transformation tools. In particular, D-KLAPER is built around these two elementary abstract concepts: (i) a software system (and its underlying platform) can be modeled as a set of resources which offer and require services; (ii) a system change can be modeled by a change in the binding between offered and required services.

Hereafter, we illustrate the main classes of the D-KLAPER metamodel illustrated in Figure 1; for a complete description of D-KLAPER the interested reader can refer to [10]. The red line in Figure 1 highlights the proposed extension.

Resource and Service metaclasses provide an abstract representation of the software system and the part of its environment consisting of the platform where it is deployed. This representation is based on the consideration that systems are often structured according to a layered architecture, where components at a given layer actually play the role of resources exploited by upper layers; hence, this overall architecture is modeled as a set of Resources which offer Services (services, in turn, may require the services of other resources to carry out their own task). A D-KLAPER Resource is thus an abstract modeling concept used to represent both software components and physical resources like processors and communication links.

Workload metaclass models the part of the system environment consisting of the demand arriving to the system from external users (which may be human beings, or other systems), represented by a set of Workloads.

The metaclasses described above share the *Behavior* metaclass, which provides a common representation for the dynamics of the activities occurring within each submodel. As shown in Figure 1, a Behavior is modeled as a directed graph of Steps. Each Step may be a:

- *Activity* step: it models an activity that may take time to be completed, and/or which may fail before completion, thus providing the basic information for performance or dependability analysis. A special kind of Activity is a ServiceCall, which models the request for the service provided by some Resource. A ServiceCall may have Parameters. D-KLAPER Parameters are intended to be an abstraction of the parameters actually used in the service requests addressed to hardware or software resources. For example, a "List" parameter sent to some list processing resource could be abstracted by an integer parameter representing its size, under the assumption that this is the only relevant information for performance analysis purposes. The relationship between a ServiceCall and the actual recipient of the call is represented by means of instances of the Binding metaclass.

- *Control* step: it models transition rules from step to step, like a branch or a fork/join.
- *Reconfiguration* step: it models a basic change operation, corresponding in D-KLAPER to the addition or removal of a Binding between a ServiceCall step and the corresponding Service. Only the behavior associated with a TriggerProcess or an AdaptationService is allowed to contain Reconfiguration steps.

The semantics of a Behavior are similar to that of other behavioral models like Execution Graphs [24] or UML Activity Diagrams [20]. As D-KLAPER is intended to support the stochastic analysis of performance or dependability attributes, timing, failure and control information associated with steps of a behavior is specified according to a stochastic setting: thus, information like the time to failure or the time to completion of an Activity are defined by suitable random variables; analogously, control information like the selection among alternative transitions, or the number of repetitions of a loop is expressed by suitable probabilities and random variables. D-KLAPER supports the specification of random variables in different ways, ranging from their mean value, to higher order moments, up to the complete distribution. It depends on the target QoS analysis methodology whether this information can be thoroughly exploited (e.g., analytic methodologies for queueing network models usually consider only mean values).

A. Metamodel Extension

The D-KLAPER metamodel has been extended with a new association between *Binding* and *Transition* shown with a red-dotted line in Figure 1. A D-KLAPER *Transition* establishes the execution order of two or more *Steps*. The new association is introduced to allow a *Step* to wait for the execution of its successor/s, so, now a successor *Step* may not execute immediately after its predecessors complete. Now, a *Step* can wait for the execution of a *ServiceCall* associated with the same *Binding* as the *Transition*, then accounting for the first one of the three abilities studied for reactive systems, (showed as (a) in section I list), i.e., to suspend system execution until the eventual reception of an event represented by the association between *Binding* and *ServiceCall*.

Summarizing, the new association allows to gain the ability (c) since the system now can send events, precisely through *ServiceCalls* which, by means of *Bindings*, are bound to *Transitions*. On the other hand, ability (b) is also gained since a *Step* can be interrupted when any of its `out` *Transitions* receives an event through their associated *Binding* due to the occurrence of the bound *ServiceCall*.

Some issues deserve to be clarified:

- A *Transition* without associated *Binding* is "taken" when its precedent *Step* (role `from`) finishes its execution.
- A *Transition* with associated *Binding* is "active" when its `from` *Step* is executing. A *Transition* with associated *Binding* is "taken" when it is active and the *ServiceCall*

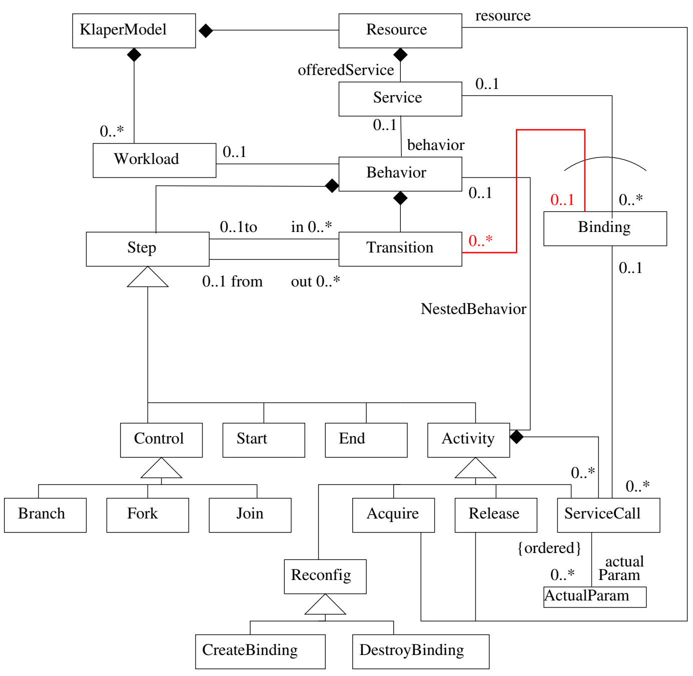


Fig. 1. D-KLAPER metamodel

associated to its same *Binding* is performed. When a *Transition* with associated *Binding* is taken, the system: 1) *interrupts* its `from` *Step*; 2) deactivates itself and all the *Transitions* with the same `from` as it; and 3) activates the `to` *Step*. So, when the *ServiceCall* associated to the *Binding* of an "active" *Transition* takes place, the execution is shifted immediately from its `from` to its `to` *Step*.
- It is worth noting that since each *Binding* can be associated with several *Transitions*, it can happen that these transitions could be concurrently active and waiting for the *ServiceCall* execution to be taken. We have decided that only one of these *Transitions* may be taken (the choice will be non-deterministic). Consider that another alternatives could be taken into account, for example: 1) to allow to fire all *Transitions* associated with the same *Binding*; 2) to perform a probabilistic choice by assigning an attribute "probability" to each *Transition*.

We show through four examples, depicted in Figure 2, that the proposed extension is able to model the characterizing points of reactive systems introduced in Section I. They show two activities, `activity1` and `activity2` in sequence. In this Figure, the new associations between *Bindings* and *Transition* are depicted as dotted lines.

Part (a) in Figure 2 depicts these two activities as a fragment of a workflow with neither interruption nor suspension abilities, i.e., the typical order between activities that was already possible to model with D-KLAPER. Part (b) extends the previous one considering *interruption* of `activity1` due to the reception of a signal (`signal1`). The activation of the outgoing *Transition* from `activity1` which is associated to the *Binding* is made at the same time as the activation of the activity itself. Therefore, if `signal1` is received (i.e. it is executed the *ServiceCall* associated with the same *Binding*) while `activity1` is executing, the *Transition* associated with the *Binding* is taken, then deactivating immediately the execution of `activity1` and starting also instantaneously the execution of `activity2`. In general, the activation of an outgoing *Transition* with associated *Binding* from a *Step* is made at the same time as the activation of the Step itself. Note that the interruption due to a signal could lead the execution to another *Step* different to `activity2`, but it has been used the same to keep the example simple. If `signal1` is not received, then `activity1` finishes its execution, the *Transition* without associated *Binding* is taken then deactivating both `activity1` and the *Transition* with associated *Binding*, and just after `activity2` is activated and starts its execution.

Part (c) in Figure 2 extends part (a) considering *suspension* of `activity2` until an event/signal reception. Suspension is modeled with a *Step* whose outgoing *Transitions* are associated with a *Binding* entity. In this case, `activity1` and `activity2` are connected by means of an intermediate activity called `inSuspension`. Execution of `activity1` is always completed (non interrupted) and later the execution can be suspended until the reception of `signal1`, which is necessary to proceed with `activity2`.

Part (d) in Figure 2 extends part (a) by considering the mix of both *interruption* and *suspension*. On the one hand, if `activity1` is in execution and `signal1` is received, the execution is interrupted, the *Transition* is taken and `activity2` is activated to execution. On the other hand, if `activity1` completes but `signal1` has not been received yet, then the execution is suspended until reception of such signal. In short, `activity2` immediately starts its execution if and only if `signal1` is received. Note that part (d) is almost the same as part (b) but removing the transition without associated *Binding* (second link between `Activity1` and `Activity2`) in order to gain the suspension property.

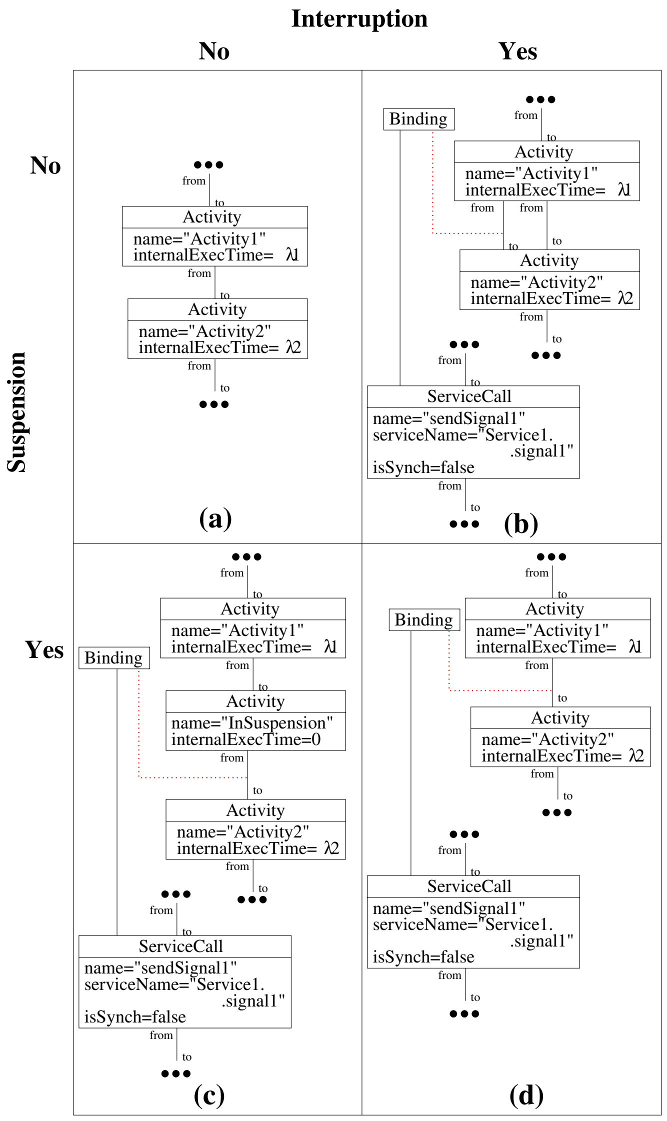


Fig. 2. D-KLAPER examples for combination of Interruption/Suspension

III. The Model-Driven Framework for Reactive Systems

In this section we first present the key points of our MDD-based approach to the generation of a performance/reliability model for a reactive system (section III-A). Then we give a short overview of the selected design model (section III-B) and of the two transformations steps (sections III-C and III-D) built around D-KLAPER.

A. The Basic Methodology

As we mentioned in the introduction, the goal of an intermediate language is splitting the complex task of deriving an analysis model (e.g., a Petri net or a queueing network) from a high level design model (expressed using some design oriented notation) into two separate and presumably simpler tasks:

- extracting from the design model only the information that is relevant for the analysis of some QoS attribute and expressing it in terms of the key concepts provided by the intermediate language;
- generating an analysis model based on the information expressed in the intermediate language.

These two tasks may be solved independently of each other. The D-KLAPER goal is to capture in a lightweight and compact model only the relevant information for the stochastic performance and reliability analysis of reactive systems, while abstracting away irrelevant details.

The input of our framework is represented by a design-level model of a reactive system. There exist some non-formal languages and notations that allow to model reactive behavior in software systems. Among them Harel statecharts [13] or UML state machines [20]. In this paper we select UML state machines as design models since they have become a de facto standard for software reactive behavior specification and we show how the proposed D-KLAPER extension is able to tackle the new design model characteristics. In the next subsection, we briefly recall the UML state machines syntax, and identify how they address the three identified abilities for reactive behavior (suspension, interruption and event sending).

Independently from the selected notation, design models may lack performance and/or reliability information which is necessary to derive meaningful analysis models. Therefore, these models must be annotated with missing information about non-functional attributes. In the case of UML design models, annotations can be added following the MARTE-DAM profile [7].

At this point, we generate D-KLAPER models starting from the design models with performance/reliability annotations using model-to-model transformations and following the main steps illustrated in [11].

Finally, we can generate from the D-KLAPER model a performance and/or reliability model expressed in some machine interpretable notation, and then we can solve it using suitable solution methodologies. In our framework, we take advantage from the already defined translations from D-KLAPER into formalisms that can adequately represent reactive behavior, such as Petri nets [21], and we modify and update the translation process to effectively support the reactive property added to D-KLAPER.

The predictions obtained from the analysis of performance and/or reliability models obtained at this step can be exploited to perform *what-if* experiments and to drive design decisions leading to meet the desired quality requirements.

B. UML State machines as Reactive Systems Models

A UML state machine is made of *states* and *transitions*. There are different kind of states (e.g., pseudostate, simple or composite). States own outgoing transitions that target another states. So *transitions* link states and are made of two parts. The *reactive part* that specifies the event that triggers the transition, and the *proactive part* that specifies the event that will be send. When the *reactive* part is empty, it is called *automatic* and taken as soon as the activity completes its execution. In a state, it can also be specified an *activity*, which is meant to spend some computation time.

A UML state machine can specify the three abilities identified for reactive behavior:

- There can be *sent events* between state machines. That events can be produced among others by the proactive part of a transition.
- *Execution interruption* is modeled by a state that when executing an *activity* accepts an event (obviously an event that can trigger one of its outgoing transitions).
- The execution can be *suspended* in a state, but it is necessary condition that all its outgoing transitions own reactive part, i.e., a trigger event. Assuming that, there are two ways for modeling suspension: (a) If the state has not *activity*, then the execution is suspended upon entrance in the state; (b) If it owns *activity*, then the execution is suspended from the activity termination to the arrival of an event triggering whatever transition.

Note that the purpose of this work is not the comprehensive transformation of every characteristic of UML state machines into D-KLAPER model, but the enhancement of D-KLAPER to deal with reactive systems. Hence, the use of simple UML state machines is enough to show the three studied properties of reactive systems. Herein, it is carried out the transformation of such simple UML state machines; being out of the scope of this work complex state machines, such as those with composite states, concurrent regions, deferred events, or history states.

C. Transforming UML State machines into D-KLAPER Models

In this section we present how a generic simple state of a UML state machine, see Figure 3, is translated into a D-KLAPER model, see Figure 4. Considering that a UML state machine is an aggregation of states, together with its outgoing transitions, we could easily obtain the D-KLAPER model corresponding to a UML state machine made of simple states.

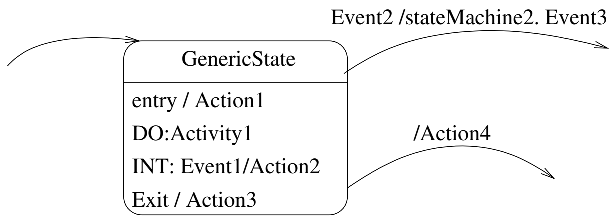


Fig. 3. UML model of a generic state

We describe the translation through Figure 4, that obviously follows the execution model of a simple state proposed by UML. First, upon state entrance, the *entry* action has to be executed, so it is converted into a D-KLAPER activity that has an `internalExecTime` equals to zero. Then, the *doActivity* is also converted into a D-KLAPER activity but in this case the `internalExecTime` has to be greater than zero. The translation of an event-driven outgoing transition is more laborious:

- the reactive part is represented in D-KLAPER by a link to a *Binding*.
- the proactive part in UML can be specified either by an *action* or by the *sending* of an event. In D-KLAPER, the former will be obviously translated as the *entry* actions, and the latter with a `ServiceCall`.
- finally, the *exit* action specified in the state is also translated as a part of the transition, see Figure 4.

The translations for internal transitions and automatic outgoing transitions are the same as the previous one but considering that:

- the *internal* does not executes the *exit* action.
- the *automatic* has not reactive part.

Finally, it is interesting to remark that:

- wherever an *action* (entry or exit) can be specified, then it can be substituted by the *sending* of an event. In fact, that is what happened in the proactive part of a transition.
- the translation of the *exit* action will appear as many times as outgoing transitions exist in the state (both automatic and event-driven).

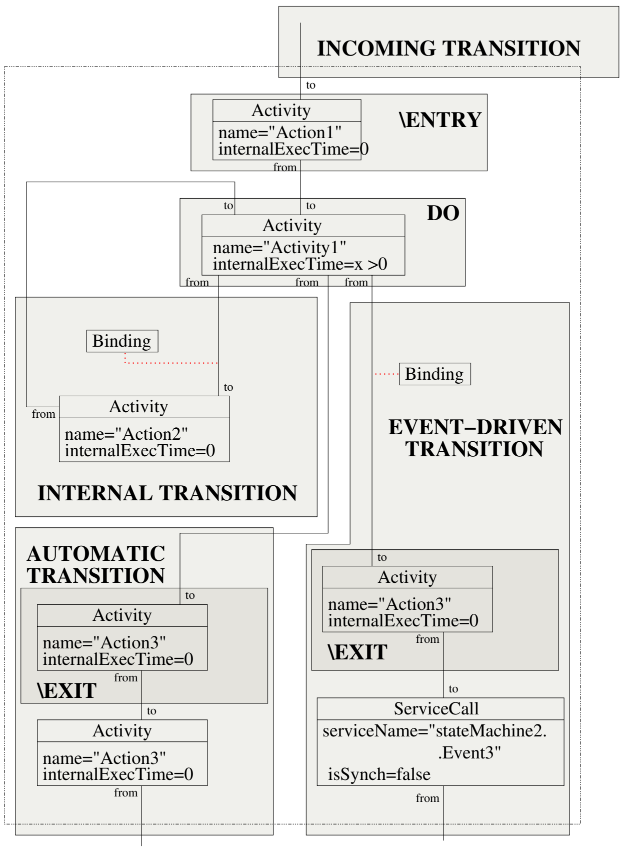


Fig. 4. D-KLAPER model of a generic state

D. *Transforming D-KLAPER Models into Petri Nets*

D-KLAPER can be transformed to a number of analysis models, for example Deterministic and Stochastic Petri nets (DSPN) [2], that (1) will allow the evaluation of performance and reliability and (2) since they are a formal method, the source model can also gain a representation with formal execution semantics.

A DSPN system is a 8-tuple $\mathcal{S} = (P, T, \Pi, I, O, H, W, M^0)$, where P is the set of places, T is the set of transitions (immediate and timed), $P \cap T = \emptyset$; $\Pi : T \rightarrow \mathbb{N}$ is the priority function that assigns a priority level to each transition. $I, O, H : T \rightarrow 2^P$ are the input,output, inhibition functions, respectively, that map transitions onto the power set of P; $W : T \rightarrow \mathbb{R}$ is the weight function that assigns rates to exponentially distributed transitions, constant delays to deterministic transitions and weights to immediate transitions. $M^0 : P \rightarrow \mathbb{N}$ is the initial marking of the net.

The Petri nets in Figure 5 correspond to the translation of the D-KLAPER in Figure 2 which indeed introduced the three abilities we identified for reactive behavior.

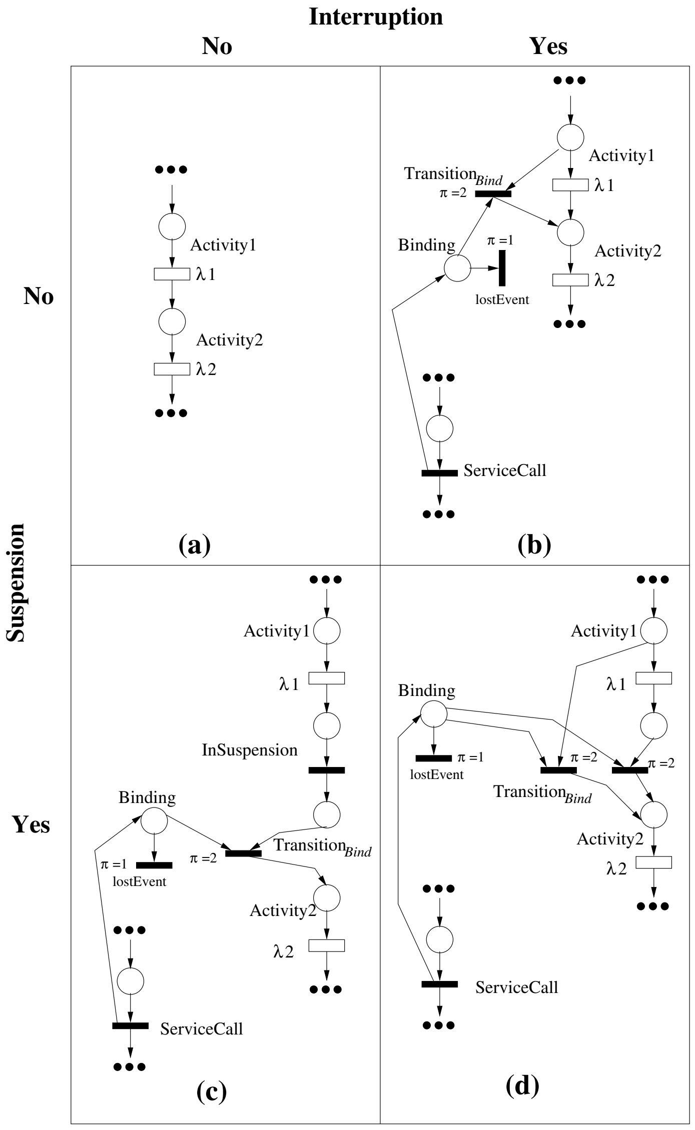


Fig. 5. Petri net examples for combination of Interruption/Suspension/EventSending (transformation of Figure 2)

The *event sending* ability is represented in parts (b), (c) and (d) in Figure 5 by means of `ServiceCall` transitions, which are in charge of create a token in the `Binding` place. The *suspension* ability is represented in parts (c) and (d) by means of `Transition`$_{Bind}$. In these parts, `Transition`$_{Bind}$ are the only ones that link `Activity1` with `Activity2` Petri net fragments. Therefore the execution flow cannot reach `Activity2` until the firing of such transitions, which are further waiting for the creation of a token in `Binding` places. *Interruption* ability is represented in parts (b) and (d) through arcs between `Activity1` places and `Transition`$_{Bind}$. In these parts, if a token in `Binding` place is created while `Activity1` is executing, the token will be removed from the input place of `Activity1`, then interrupting it, and a token wil be created in `Activity2` place. The translation of the rest of D-KLAPER metaclasses, which are not illustrated in this paper, follows the ideas introduced in [21]. It is worth noting that, on the one hand, the size of the resulting Petri net is linear in the size of the D-KLAPER model. On the other hand, in order to perform a state-space-based analysis, the size of that state-space is potentially exponential in the size of the Petri net.

IV. Example application

In this section, we illustrate reactive behavior with a simple example of a dynamic software system. UML state machines and a sequence diagram describe the behavioral design of the system, they are also extended with a profile, MARTE-DAM [7], that introduces performance and reliability system views. These diagrams are translated into a D-KLAPER model which preserves the desired reactive properties and also accounts for performance and reliability. Finally, the translation of D-KLAPER model into Petri Nets and their analysis help to verify, in early life-cycle stages, whether the system fulfills some performance and cost requirements taking also into account some dependability properties such as availability/reliability.

A. Structural Specification

A UML component diagram extended with MARTE-DAM annotations [7] is depicted in Figure 6(a). Component *C1* offers service *S0* and perform calls to *S1*, indeed there are two choices to invoke *S1*:

- a) as an Internet service that comes at a price;
- b) as a COTS component that has already been integrated and executes for free, therefore being the default option.

C1 is made of four classes as detailed in Figure 6(b). The *C2* COTS component reliability specification warns about a Mean Time To Failure (MTTF) equal to 10^5 time units (tu), and a Mean Time to Repair (MTTR) of $5 \cdot 10^3$ tu.

B. Reactive Specification

Sequence diagram in Figure 7 offers a high-level view of the interactions in the system. When a Client asks for *S0*, the *S0server* class manages the request, then creating an instance of a *Main* object. The *Main* object cooperates with the system *Monitor* and *ProviderSelector* to effectively resolve the request. Figures 8, 9 and 10 respectively depict the UML state machines of the *Main*, *Monitor* and *ProviderSelector* classes. The behavior of *ProviderSelector* represents the Adaptation-Service of the system since it decides whether *S1* service calls will be requested to *C2* or to *InternetProvider*.

The *Main* state machine will help to illustrate *suspension* and *event sending*. *Suspension* is accomplished by *CallingS1* state, when it is reached, the object will wait for the eventual arrival either of `restart` or `S1response`. Note that being the entry action execution immediate, the object is truly waiting for an event to react. Regarding *event sending* there are several examples in this state machine, e.g., before entering in *Calling* state, *Main* sends the `start` event to the *Monitor*.

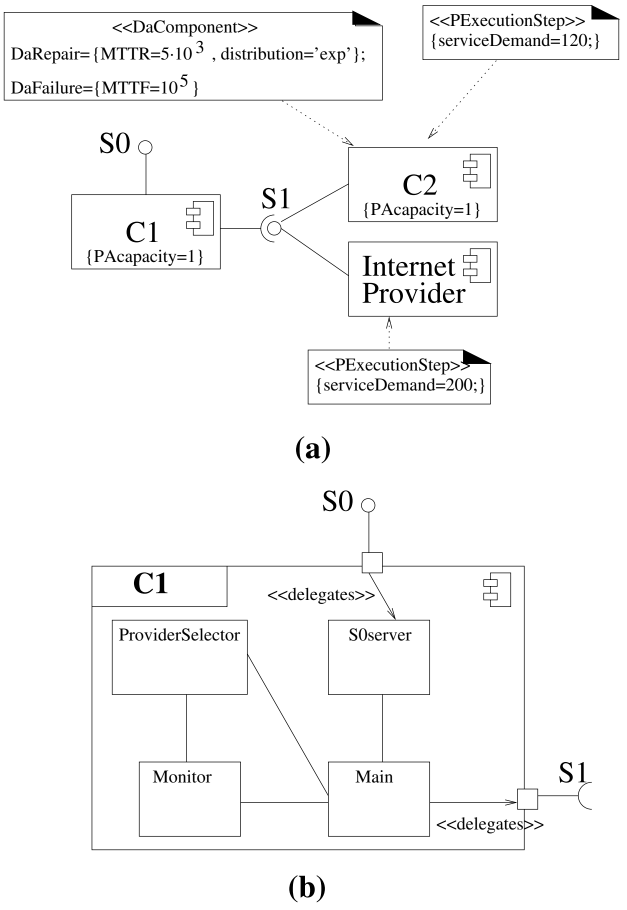


Fig. 6. Components diagrams.

The other ability we identified for reactive behavior, i.e. *interruption*, is illustrated in the *Monitor* state machine. When *monitoring*, the `stop` event can interrupt the time-out execution to bring the *Monitor* to idle.

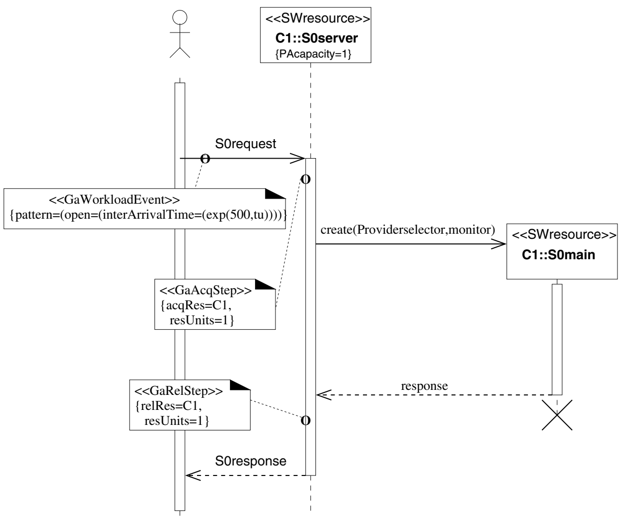


Fig. 7. Sequence diagram representing the S0 requests.

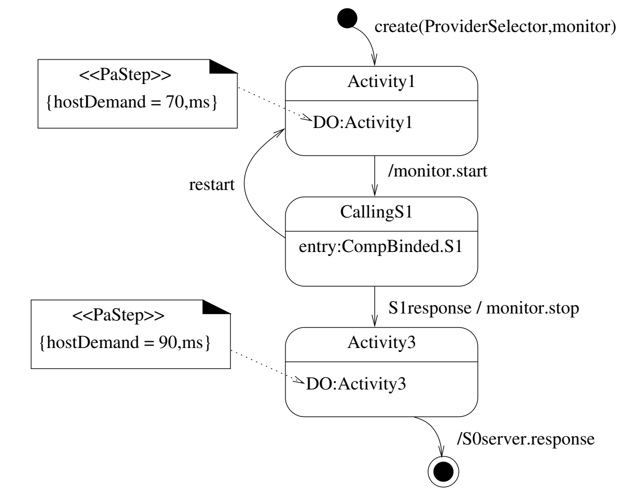


Fig. 8. Main UML State machine

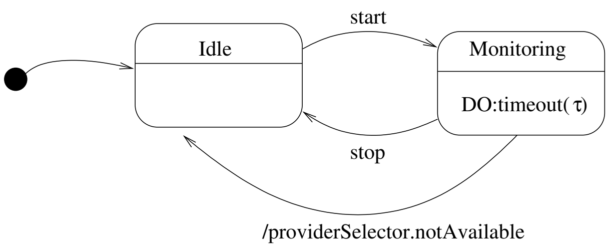


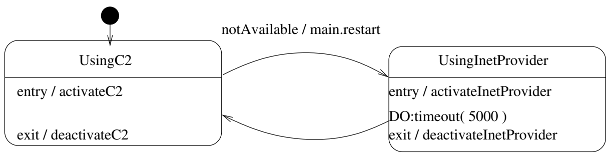
Fig. 9. Monitor state machine

C. Translation into D-KLAPER Models

Figure 11 depicts the D-KLAPER corresponding to the sequence diagram in Figure 7, here it is important to note how the system workload is represented.

Figures 12, 13 and 14 depict D-KLAPER models for the UML state machines of the *Main*, *monitor* and *ProviderSelector* respectively. In the D-KLAPER model of the *Main* class, Figure 12, the call and the response to the external service *S1* (`CompBinded.S1` and `S1response` in the UML state machine) are translated as service calls. In the D-KLAPER model of the *ProviderSelector* class, Figure 14, the dashed part represents the necessary bindings for *S1* to be called, note that this is not yet specified in the UML model. However we assume this behavior is associated with the entry and exit actions in the *ProviderSelector* state machine states. Hence, note that this fact implies a D-KLAPER manual translation.

Finally, the provider components D-KLAPER models appear in Figure 15. These two models are obtained automatically translating the component diagram in Figure 6.



notAvailable / main.restart
UsingC2
entry / activateC2
exit / deactivateC2
UsingInetProvider
entry / activateInetProvider
DO:timeout(5000)
exit / deactivateInetProvider

Fig. 10. ProviderSelector state machine

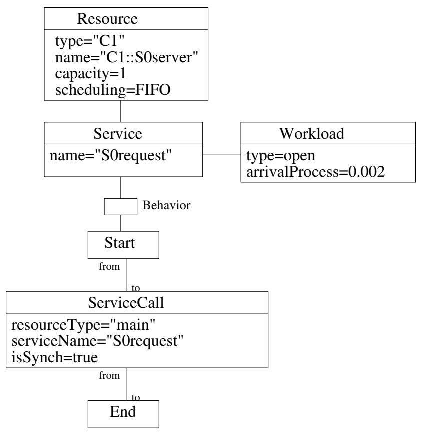


Fig. 11. D-KLAPER model representing the sequence diagram in Fig. 7.

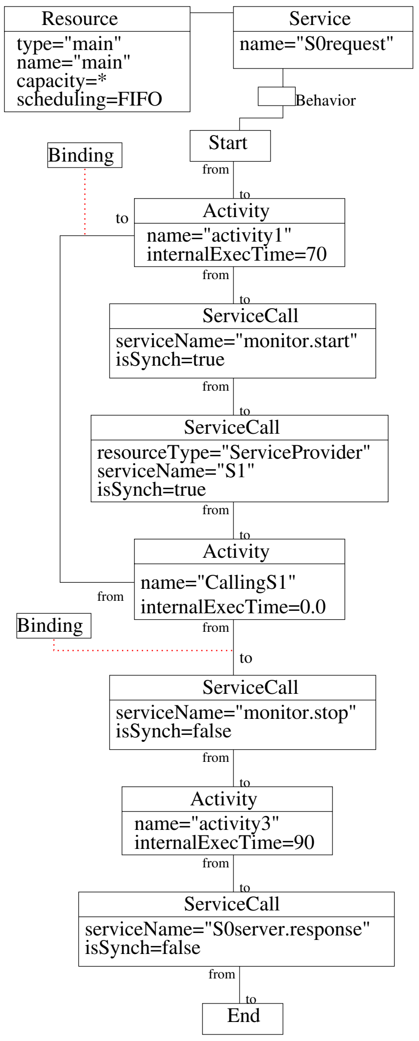


Fig. 12. D-KLAPER model representing the main class.

D. Translation into Petri Nets and Evaluation

Now, we proceed to evaluate the example, so to acquire knowledge and validate some non-functional properties. Concretely, the mean response time of *S0* and the mean monetary

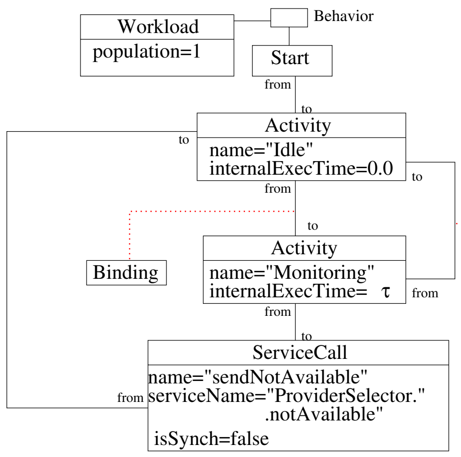


Fig. 13. D-KLAPER model of the monitor state machine (in Figure 9).

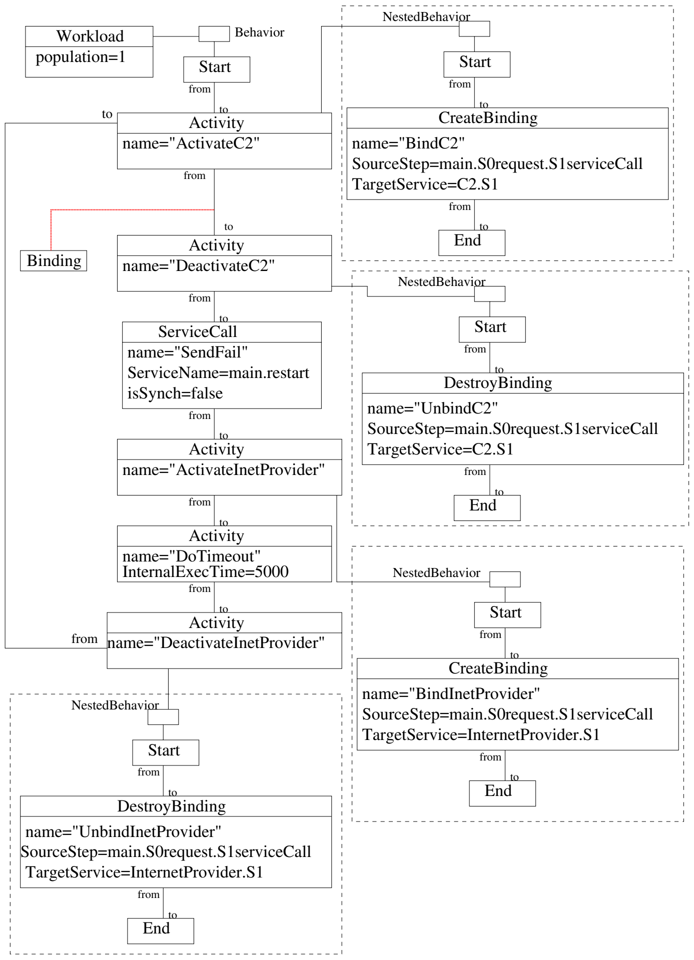


Fig. 14. D-KLAPER model of the providerSelector state machine (in Figure 10).

cost for an *S0* execution. Requirements of the system established that "the mean response time of *S0* has to be less than 700 time units (tu)" and "the mean cost for serving an *S0* request must not exceed two monetary units (mu)". A system restriction says that the *InternetProvider* offers each *S1* service call at a cost of 10 mu.

The D-KLAPER models in Figures 11 to 15 have been translated into a Deterministic and Stochastic Petri Net (DSPN) [2], following the patterns in Figure 5 and ideas from [21]. The obtained DSPN is then used to evaluate system performance and execution costs.

Variable τ in Figure 9 represents a threshold for the system to acknowledge *C2::S1* calls; upon expiration the monitor assumes that *C2* is no longer available. The higher τ is, the more *C2* will be used, then it may happen to the system to wait for *C2* while in fact it is unavailable. However, the lower τ is, the more the *InternetProvider* will be used, in this case the monitor may predict *C2* unavailability when it can be only performing an unusual slow service.

Figure 16 (a) depicts *S0* mean response time w.r.t. τ. The performance requirement is met from $\tau = 440$ (693.7tu) to τ=1640 (699.58tu) and a minimum is obtained around $\tau = 840$. Hence, timeouts lower than 440 confuse the system as explained in the previous paragraph, i.e., predicting erroneous *C2* unavailabilities. However, timeouts higher than 1640 lead the system to wait for *C2* even when it is dropped.

Figure 16 (b) depicts the mean cost of executing *S0* w.r.t. τ. The function decreases because, as explained, *C2*, the free component, is more used for higher values of τ. The requirement is fulfilled for τ values upper than 520, since then, the mean cost is less than 2 mu.

Considering these two graphs, we observe that the non-functional requirements are met from τ=520 to τ=1640.

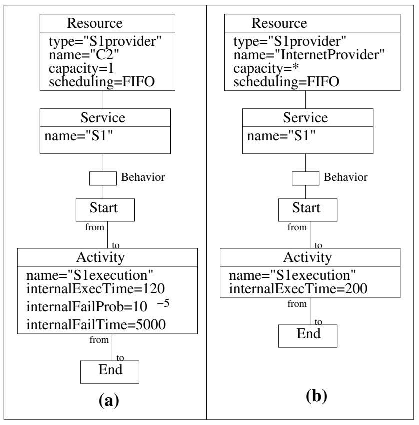
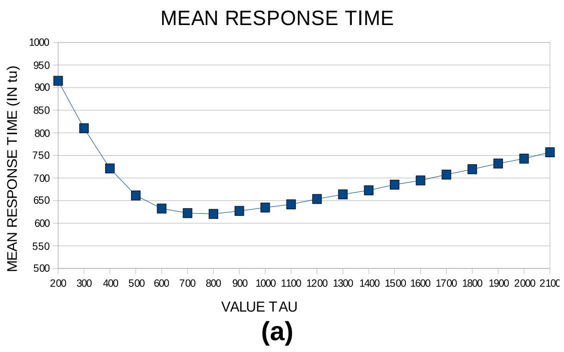


Fig. 15. D-KLAPER models representing *S1* providers: (a) C2 (b) Internet-ServiceProvider

V. Conclusions

In this paper we have presented a Model-Driven approach whose goal is to support the QoS assessment of adaptable reactive systems. Our approach builds on the existence of intermediate modeling languages and extends one of them, to capture the core features (from a performance/dependability viewpoint) of an adaptable reactive system model. We are working on the complete automation of the proposed approach, since this represents a key point for its successful application and complete validation by applying it to industrial case



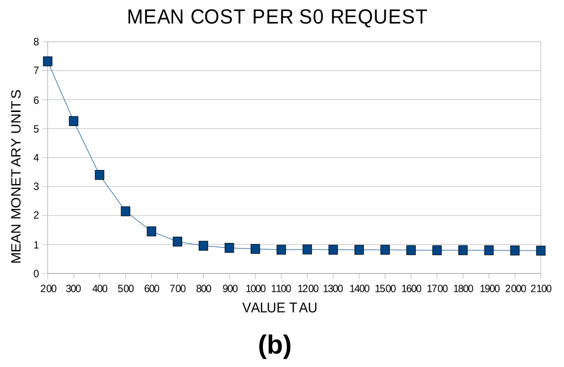


Fig. 16. Results of the system evaluation

studies.

Acknowledgment

The work of R. Mirandola and V. Grassi has been partially supported by the EU FP7 Q-ImPrESS project. The work of D. Perez-Palacin and J. Merseguer has been supported by the project DPI2006-15390 of the Spanish Ministry of Science and Technology.

References

[1] WOSP conference series, 1998-2010.

[2] M. Ajmone Marsan and G. Chiola. On petri nets with deterministic and exponentially distributed firing times. In *Advances in Petri Nets 1987, covers the 7th European Workshop on Applications and Theory of Petri Nets*, pages 132–145, London, UK, 1987. Springer-Verlag.

[3] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *FASE*, pages 21–37, 1998.

[4] D. Ardagna, C. Ghezzi, and R. Mirandola. Rethinking the use of models in software architecture. In *QoSA*, volume 5281 of *Lecture Notes in Computer Science*, pages 1–27, 2008.

[5] C. Atkinson and T. Kühne. Model-driven development: A metamodeling foundation. *IEEE Softw.*, 20(5):36–41, 2003.

[6] S. Becker, H. Koziolek, and R. Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1), 2009.

[7] S. Bernardi, J. Merseguer, and D. Petriu. A dependability profile within marte. *Software and Systems Modeling*, 2009.

[8] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS*, pages 28–33. ACM, 2004.

[9] B. H. C. Cheng and al. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009.

[10] V. Grassi, R. Mirandola, and E. Randazzo. Model-Driven assessment of QoS-Aware Self-Adaptation. In *Software Engineering for Self-Adaptive Systems*, pages 201–222, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] V. Grassi, R. Mirandola, and A. Sabetta. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *Journal of Systems and Software*, 80(4):528–558, 2007.

[12] V. Grassi, R. Mirandola, and A. Sabetta. A model-driven approach to performability analysis of dynamically reconfigurable component-based systems. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, pages 103–114, New York, NY, USA, 2007. ACM.

[13] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[14] D. Harel and A. Pnueli. On the development of reactive systems. In *In K. R. Apt, editor, Logics and Models of Concurrent Systems, volume F-13 of NATO ASI Series*, pages 477–498, 1985.

[15] F. Irmert, T. Fischer, and K. Meyer-Wegener. Runtime adaptation in a service-oriented component model. In *SEAMS*, 2008.

[16] M. H. Kacem, M. N. Miladi, M. Jmaiel, A. H. Kacem, and K. Drira. Towards a uml profile for the description of dynamic software architectures. In *COEA*, LNI, pages 25–39. GI, 2005.

[17] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture(TM): Practice and Promise*. Addison Wesley Object Technology Series, 2003.

[18] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE*, pages 259–268, 2007.

[19] A. D. Marco and R. Mirandola. Model transformation in software performance engineering. In *QoSA*, volume 4214 of *Lecture Notes in Computer Science*, pages 95–110, 2006.

[20] Object Management Group. UML 2.0 superstructure specification, 2002.

[21] D. Perez-Palacin and J. Merseguer. Performance evaluation of self-reconfigurable service-oriented software with stochastic petri nets. *Electronic Notes in Theoretical Computer Science*, 261:181 – 201, 2010. Proceedings of the Fourth International Workshop on the Practical Application of Stochastic Modelling (PASM 2009).

[22] D. B. Petriu and C. M. Woodside. An intermediate metamodel with scenarios and resources for generating performance models from uml designs. *Software and System Modeling*, 6(2):163–184, 2007.

[23] Q-ImPrESS. Quality impact prediction for evolving service-oriented software.

[24] C. U. Smith and L. G. Williams. *Performance and Scalability of Distributed Software Architectures: an SPE Approach*. Addison Wesley, 2002.