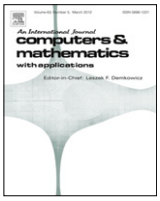
Computers and Mathematics with Applications 64 (2012) 3828–3839

Contents lists available at SciVerse ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa



The SIMTHESys multiformalism modeling framework

M. Iacono [a,*], E. Barbierato [b], M. Gribaudo [c]

[a] Seconda Università di Napoli, Dip. di Studi Europei e Mediterranei, Caserta, Italy

[b] Università di Torino, Dip. di Informatica, Torino, Italy

[c] Politecnico di Milano, Dip. di Elettronica e Informazione, Milano, Italy

ARTICLE INFO

Keywords:
Modeling languages
Metamodeling
Performance evaluation

ABSTRACT

The usage of models is a fundamental activity in designing and verifying a system. Mastering different modeling techniques and scaling their application to complex systems is not an easy task and requires both advanced skills and proper tools. One of the means that allow modelers to leverage the power of proper modeling techniques (e.g. stochastic techniques) is the application of abstractions by using high level formal modeling languages. This paper presents SIMTHESys, a framework for the development of formal modeling languages and the solution of multiformalism models by automatically generated solvers based on different solving engines.

1. Introduction

Modeling is one of the most powerful tools developed to master the complexity of reality. The scale of the systems currently in charge to support human activities and the concurrency of different non-functional specifications require that modeling techniques would offer abstraction mechanisms that allow modelers to face such complexity while exploiting mathematically founded methods. Such abstraction mechanisms can be constituted of formal modeling languages and their solution algorithms. A broad and well spread example set is given by stochastic modeling techniques.

Literature offers many such modeling formalisms, more or less abstract and fit to model systems from a given point of view: examples are Petri nets, fault trees, stochastic automata, process algebras, queuing networks and many others that have been given a stochastic variant or interpretation. Nevertheless, all these useful and powerful abstractions are built over common low-level tools, such as Markov chains. Their existence enables modelers to cope with more difficult problems keeping the same well-known advantages.

SIMTHESys (Structured Infrastructure for Multiformalism modeling and Testing of Heterogeneous formalisms and Extensions for SYStems) is a framework for the definition of new formalisms and the generation of related solvers, that allow the combination of more formalisms in the same models. Formalisms and models definition is supported by a family of languages based on XML. SIMTHESys is mainly a conceptual framework that assists the process of defining new formalisms. As the definition of new formalisms would not be useful without the availability of related solvers, SIMTHESys is complemented with a solver generation tool (namely SIMTHESys*ER*), capable of automatically building simple multiformalism solvers based on generic elementary solvers (solving engines, in SIMTHESys terminology). SIMTHESys supports the design and the experimentation of modeling abstractions that can be used to ease the modeling process by encapsulating different solution methods and techniques into high level, customizable modeling languages.

* Corresponding author.
E-mail address: mauro.iacono@unina2.it (M. Iacono).

0898-1221/$ – see front matter © 2012 Elsevier Ltd. All rights reserved.
doi:10.1016/j.camwa.2012.03.009

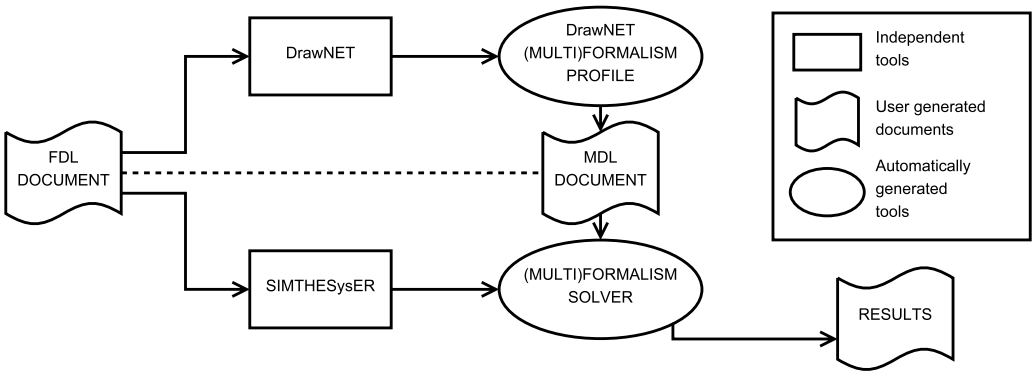


Fig. 1. The SIMTHESys workflow.

Stochastic formalisms are supported by the framework in the current phase of the project, though any kind of formalisms can be implemented in line of principle.

The goal of this paper is to introduce SIMTHESys, its main ideas and its contribution to the field of multiformalism modeling: a formal description of the framework is thus out of the scope of this paper. Currently, being the focus of our research the exploration of a new multiformalism modeling paradigm, optimization of solvers (indeed an important topic) is out of the scope of our research and will be faced in the future.

In this paper the SIMTHESys modeling framework is presented: Section 2 explains how formalisms and models can be designed, and with which limitations and possibilities; Section 3 examines the architecture that allows solvers generation and the model solution process; Section 4 shows some examples of use of SIMTHESys; finally Section 5 presents conclusions and future work. For this purpose, some general information about the framework and other similar approaches is needed.

1.1. Framework description

A sound description of the framework should deal with both model description and analysis aspects. The SIMTHESys framework consists of the modeling stack, the solving stack and SIMTHESys*ER*. The two goals of the framework (producing solvers for new formalisms and describing models to be solved) are accomplished by two complementary processes, that are both represented in the SIMTHESys workflow in Fig. 1. In the figure, tools and documents that are involved in the two processes are depicted with different symbols. The solver generation process, described in Section 3.3, is described by the lower part of the figure. The figure also includes DrawNET [1], an external general-purpose GUI generator that is adopted as graphical interface for the framework and is used to assist the user in writing MDL model documents by a graphical editor. DrawNET can generate a custom GUI by associating graphical primitives to the information obtained from the analysis of a FDL formalism document. Besides SIMTHESys*ER* and DrawNET, the ovals represent the automatically generated tools obtained by their execution, that form the elements of the modeling process. The dotted line stresses the fact that the MDL document must conform to the FDL document in the process.

1.2. Comparison with other frameworks

SIMTHESys is based on the application of metamodeling techniques to the description and the analysis of performance models (where performance should be interpreted in the broadest sense). Metamodeling is used to found formalisms extensibility and multiformalism capabilities on common features, supplied by the SIMTHESys metametamodel. The metametamodel is the foundation on which the metamodels are built. Metamodels form the layer of SIMTHESys on which the main focus is, because they are the formalisms with which models can be created. To better focus the perspective in which metamodeling is used in SIMTHESys, a light comparison with eCore [2] is useful. eCore is the metamodeling stack on which the Eclipse Modeling Framework is founded. eCore is used to allow the description of software entities independently from the platform on which they will be implemented: an eCore model is thus equivalent to an object-oriented application with its business logic, and it is used to generate plain source code in the desired language for the desired platform when needed. The focus in eCore is thus on models, and the eCore metamodel, designed to describe a generic object oriented language for software development, is generally used as it is. In the case of SIMTHESys, the purpose is to develop different metamodels, each of which is a formalism and shares the common metametamodel, so that models written according to different metamodels can interact though showing different characteristics, logic and internal organization.

To the best of our knowledge, SIMTHESys [3–6] is the only framework designed to support rapid formalism development (and automatic solver synthesis). Nevertheless, other extensible multiformalism modeling frameworks have provided the main contributions to the field. A different metamodeling approach has been adopted by AToM^3 [7], that exploits

metamodeling to implement model transformations, used to solve models by its solver. OsMoSys [8] uses metamodeling to allow the modeler to write composed multiformalism models, based on an object oriented representation of formalisms and model elements, and solves them by using a workflow based approach that integrates external solvers, properly wrapped. Within other non-metamodeling based approaches, Mobius [9] is based on a sound model organization to manage composition and multiformalism and generates executables that solve the single model applying the best solving approach in the specific case. Sharpe [10], SMART [11] and the DEDS toolbox [12] provide multiformalism modeling, with a minor stress on extensibility of the set of supported formalisms with respect to the other references presented.

1.3. A comparative analysis

A brief comparison between SIMTHESys and the most similar approaches can be based on four main points: model structure, extensibility of the supported formalisms set, multiformalism and multisolution.

From the point of view of model structure, OsMoSys and SIMTHESys share the main organization: both of them represent a main metaformalism (metametamodel) on which formalisms (metamodels) are based, and that describe the elements available in models; OsMoSys supports formalism inheritance (at formalism and element level), while both of them can extend formalisms by adding new elements. Both allow model composition by inclusion of submodels, with OsMoSys supporting generic submodels and information hiding. Both support multiformalism models with bridge formalisms, while SIMTHESys also allows interformalism connection elements and a more flexible interaction. Mobius presents a more complex model architecture, in which several different model types, organized in a logic tree, take care of different information needed to instantiate, parameterize and solve a model. AToM^3 describes models as user-defined graphs, with the support of metamodeling that supplies a grammar oriented to graph transformations.

With respect to the extension of the supported formalisms set, SIMTHESys, Mobius, OsMoSys and AToM^3 explicitly offer specific mechanisms. Being the first oriented to rapid formalism development, adding a new formalism just requires the XML description of its elements properties and semantics, in terms of use of one or more elementary solvers. Eventually, the development of a new elementary solver is requested, if the new formalism is based on a completely different logic with respect to the existing ones. The second allows specification of a new formalism by XML as well, by describing elements properties, but requires the corresponding solver to exist as an external tool and the development of a proper wrapper (namely adapter) to manage it. The latter allows a new formalism to be translated into existing ones by providing a proper model transformation description. Mobius, being oriented to models solution rather than on formalisms/models manipulations, can be extended by adding proper software modules.

With respect to multiformalism, SIMTHESys and OsMoSys support the development of multiformalism models by composition of submodels written in different formalisms by exploiting the benefits of metamodeling. AToM^3 deals with multiformalism by applying model transformations, while Mobius exploits its Abstract Functional Interface (AFI) that acts as a generalized superformalism to which all formalisms refer.

Multisolution is dealt in SIMTHESys, OsMoSys and Mobius in a different way. Although they are absolutely not comparable from the point of view of solution efficiency and sophistication, Mobius and SIMTHESys present some similarities in the generation of the final solver. Such a solver is synthesized automatically, but in the case of Mobius it is obtained in the form of an optimized executable model, based on the description given by the user. In the case of SIMTHESys, a solver that can solve all models based on the same formalism combination is generated. OsMoSys solves models by (semi-automatically) generating a business process, executed by its workflow engine, that describes the solution in terms of external solvers activations. From this point of view, it could be said that a Mobius model is executed, a SIMTHESys model is interpreted and an OsMoSys model is orchestrated.

2. Modeling in SIMTHESys

The SIMTHESys modeling framework is based on metamodeling. Metamodeling is the study of the rules and the structures that allow the specification of models. Since a model is an abstraction of the real world, a metamodel is an abstraction of a class of models. According to metamodeling theories, a certain class of models that share the same substanding logic can be described by a common metamodel, that denotes the general elements, their constraints and the relationships between them. Such a metamodel can be considered the description language for the class of models. This kind of approach supports the exploration of the structure of modeling techniques and can be applied recursively to extrapolate the structure of metamodels and to find more abstract common description rules for metamodels. A clear presentation of metaformalism concepts and applications is given in [13–15].

Metamodeling is currently a consolidated conceptual tool, that is applied in different fields (e.g. Model Driven Engineering [16], software engineering [17] and multiformalism modeling [8,18,7]). Metamodeling is in some relation with ontologies [19], as a metamodel can be considered a formally defined ontology.

In SIMTHESys, *models* are written according to metamodels known as *formalisms*, that are expressed by the metametamodel known as the *SIMTHESys metaformalism*. In the SIMTHESys framework, metamodeling is used to offer the users a consistent tool for the specification of modeling formalisms and to obtain a formalism organization that allows

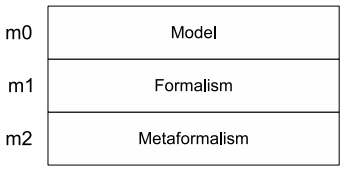


Fig. 2. SIMTHESys metamodeling stack.

formalisms extension and solvers synthesis (see Fig. 2).[1] The metamodeling architecture of the SIMTHESys framework is thus designed for three main goals:

- allowing a fast specification of new formalisms (also exploiting existing ones by extension);
- supporting multiformalism modeling;
- automatically generating solvers for (customized) formalisms (combinations).

SIMTHESys metamodeling structure is inspired to OsMoSys [8] (and DrawNET). The main difference between the two approaches lays in the description capabilities of the metaformalism in SIMTHESys with respect to the metaformalism in OsMoSys, due to the different aims of the two projects. While OsMoSys metaformalism is a consistent foundation for the description of static aspects of formalisms elements and constraints (mainly syntactical), SIMTHESys metaformalism has been designed to describe both static and dynamic aspects (specially focusing on execution semantics of the elements of a formalism), to allow a further customization of the framework.

2.1. The SIMTHESys metaformalism

The metaformalism specifies the grammar by which the user can describe formalisms. It is designed to allow the description of every abstract model element, in terms of the structure of its inherent information and its possible evolutions in the model according to the conditions that will be met during model evolution. The three key concepts of every SIMTHESys formalism are the element, the property and the behavior. An element is a syntactic atomic component of a formalism (e.g. a place, a transition or an arc in the PN formalism). It is characterized by a number of properties, representing its status information, and a number of behaviors, describing its dynamics. The reader could see some similarities with the concept of Object Oriented programming.

Fig. 3 gives an UML-like description of the SIMTHESys metaformalism and an example formalism (Stochastic Petri Nets) with two extensions (in gray and light gray) that will be considered in Section 2.2. In the upper part the package M2 describes in detail the parts of a formalism that have been presented. The figure presents *elements* and some specialized *formalism elements*, that indicate submodels written in a formalism, and can contain other elements as specified by *containing*. An element *has properties* and *behaviors*. A property allows the definition of a value and a type (not represented in the figure) and can be a constant, a variable or a result. A behavior defines an action that the element can perform, eventually on other elements (not represented in the figure) and it corresponds to a method in Object Oriented programming. An example of how the method bodies are defined and how methods are related to each others for the enabling and firing rules of a PN transition can be found in [6]. An element *uses* one or more *solver interfaces* to define which solving engine(s) should be used with it (it is generally the case of formalism elements), and one or more *behavioral interfaces* to reuse existing abstractions. Moreover, an element *implements* a behavioral interface by explicitly defining all or some behaviors and properties specified *in* it. The role of interfaces will be further examined in Section 3.

Such a structure allows us to satisfy the three goals of the framework: extensibility of formalisms, compositionality of models and possibility of multiformalism modeling. Extensibility of formalisms[2] is supported by the fact that every element can be enriched by adding more behaviors or properties and every formalism element can be enriched by adding more elements. Compositionality of models is provided by the fact that a formalism element can contain other formalism elements, that allows models to contain other (sub)models. Multiformalism is granted by compositionality of models and explicit specification of behaviors in every element. Potentially, this allows any element to be able to use behaviors typical of elements from different formalisms and transparently acting as a bridge, thanks to the solver generating mechanism of the framework (that builds a proper solver by examining behaviors). These three concepts will be presented again but from the point of view of the modeler in Section 2.2.

2.2. SIMTHESys formalisms

Formalisms describe a certain formal language in terms of elements, properties and behaviors. Every formalism depends on its formalism element for the specification of its general solution mechanism through the interfaces. It defines its internal

[1] Common metamodeling stacks consider four layers, since they include reality as the object to be modeled. In SIMTHESys case models are actually the reality that has to be modeled, since the focus is on formalisms: thus the models layer is indicated as M0.

[2] Note that formalism extension is a kind of inheritance in the sense of object orientation. At the moment, pure formalism and element inheritance are not supported.

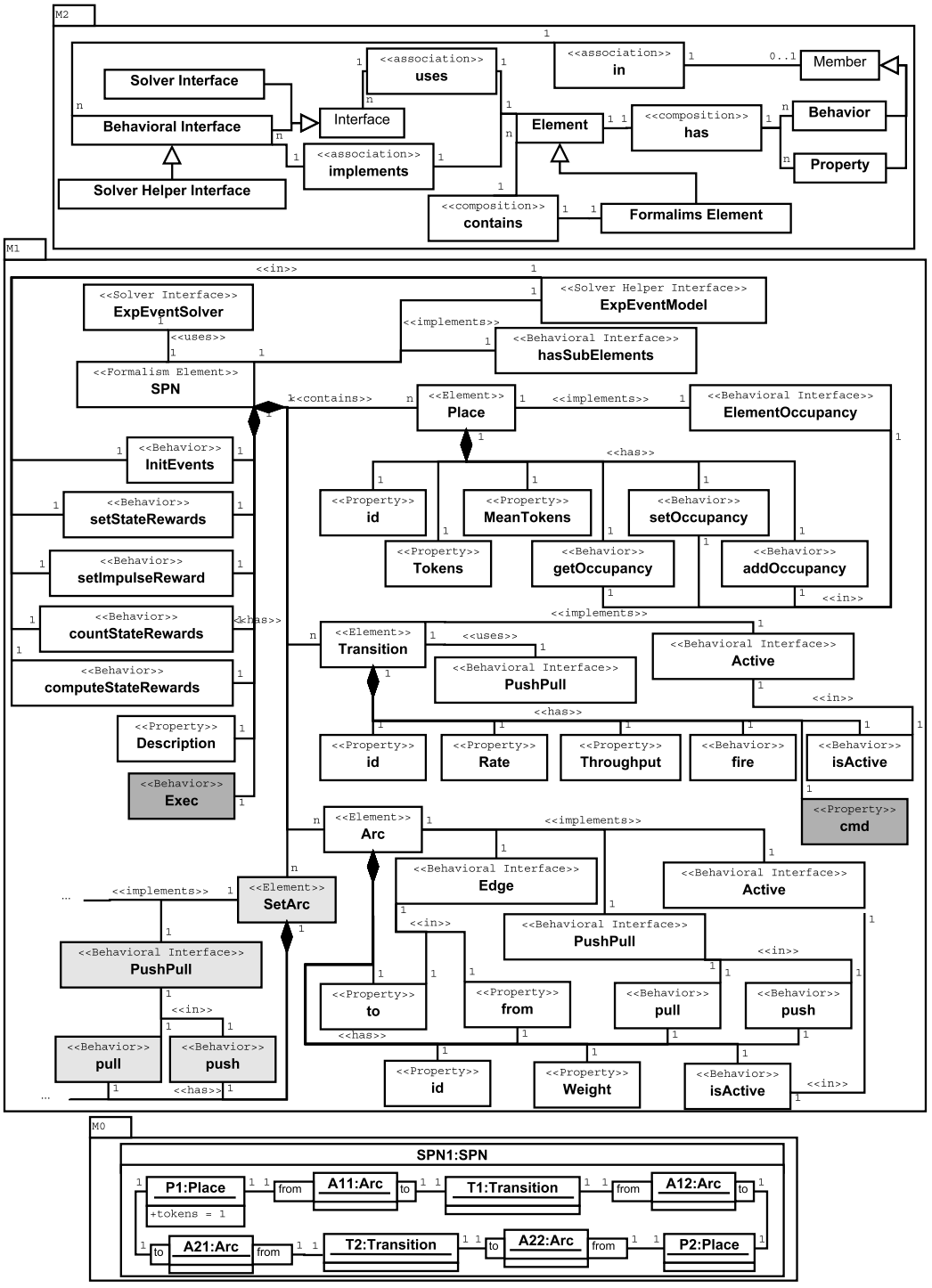


Fig. 3. Elements of the SPN formalism (with extensions) and the metaformalism.

dynamic behavior by a proper specification of its behaviors and of its elements behaviors. Composition of models of the same formalism is enabled by specifying the formalism model itself (renamed in this case *submodel element*) as an element of the formalism. Every formalism can specify how to interface with its submodels by exploiting the bridging elements and their behaviors. The interface with submodels can be defined as a whole or with inner elements by defining proper export mechanisms, in the form of additional or specialized elements of the formalism itself. Multiformalism features can be enabled by specifying different formalism elements as elements of a formalism. A formalism that is designed to

interface only other different formalisms is named *composition formalism*. The elements for which behaviors are designed for the interaction with elements of different formalisms are named *hybrid elements*. Since composition formalisms are not substantially different from other formalisms (and are actually a subset of them), hybrid elements can also be added to existing formalisms by formalism extension. The described behavior-based approach allows specifying complex interactions between heterogeneous models in a compact, atomic way that is transparent to simple modelers. It appears as a feature of a special formalism that can be applied to own models without any further knowledge of behavior design logic. Conversely, the possibility of specifying in a formalism the interaction between different formalisms offers a unique tool for the design of multiformalism models without forcing a formalism designer to redesign related solvers. Section 3 explains how this is obtained.

Each formalism is described by a Formalism Description Language (FDL) document.

In Fig. 3 the UML-like model M1 defines the SPN formalism, that is shown as example (white rectangles only). It has the *SPN* formalism element that represents the entire formalism and it is composed of elements *Place*, *Transition* and *Arc*. An SPN can contain any number of these elements. The figure describes for each element the interfaces that are used and implemented by *uses* and *implements* association arcs and their properties and behaviors by *has* composition arcs. For each property or behavior, *in* association arcs indicate whether it is specified by an interface. The FDL documents for SPN is an XML description of this UML-like model. To enable the solution of a SPN model written in this formalism, its SPN formalism element uses the *ExpEvent* solving engine. This is accomplished by means of the *ExpEventSolver* solver interface and the *ExpEventModel* solver helper interface. *ExpEventModel* requires the implementation of some behaviors to allow the solving engine to initialize, compute and set the result indexes (for this kind of engine, state and impulse reward variables). More information about the steps to integrate a formalism within the framework can be found in [20].

Note that the SIMTHESys metamodel is oriented to general graph based formalisms. This results in an explicit representation of arcs in the example (more frequently, Petri Nets are represented by omitting arcs and indicating input and output bags for each transition).

Algorithm 1 `fire`

```
1: for all a ∈ Arc where a.from = this do
2:    a.push();
3: end for
4: for all a ∈ Arc where a.to = this do
5:    a.pull();
6: end for
```

A simple behavior implementation example is in Alg. 1. The behavior states that a Transition acts soliciting all the elements that refer to it for a *push* operation, and all the elements referred by it for a *pull* operation. These operations are defined in the *PushPull* behavioral interface, used by *Transition* and implemented by *Arc*. Similarly, *isActive* will check the enabling rule by *Arcs isActive* behaviors, and enable *fire*. This simple organization is also the base for hybrid elements, that implement multiformalism: in case firing this transition should cause a queue to get a new request enqueued, it is sufficient to introduce a new *Arc*-like element implementing *PushPull* and the behavioral interface used by the queue, with a behavior capable of performing the desired interformalism action.

A hypothetical extension of SPN is also presented in Fig. 3 where gray rectangles are used to show how the extension mechanism can be used. The extension introduces set arcs (arcs that can produce tokens in a place with an arbitrary set rule) by extending the formalism with an additional *SetArc* element. It (i) implements the *PushPull* behavioral interface, (ii) extends *Transition* with the *cmd* property to specify the additional rule and (iii) extends SPN with the *Exec* behavior, capable of executing the additional rules when needed. This extension is obtained as a new instantiation of the metaformalism, by designing a new formalism that adds new elements, properties and behaviors to an SPN[3].

2.3. SIMTHESys models

Each model is described by a Model Description Language (MDL) document. In Fig. 3 the M0 package describes an example of model written in the SPN formalism just defined (details are omitted). The figure evidences the fact that places, arcs and transitions must be contained into a SPN. This model cannot contain submodels, since the formalism does not specify a self *contains* relation.

3. Solving in SIMTHESys

The solution architecture of SIMTHESys is designed for the generation of solvers according to specific needs. The formalism designer specifies the proper formalism or formalism combination. In this way, SIMTHESys*ER*, the SIMTHESys

[3] The new formalism could also be obtained by refinement of SPN: this possibility is currently unavailable but on the SIMTHESys roadmap.

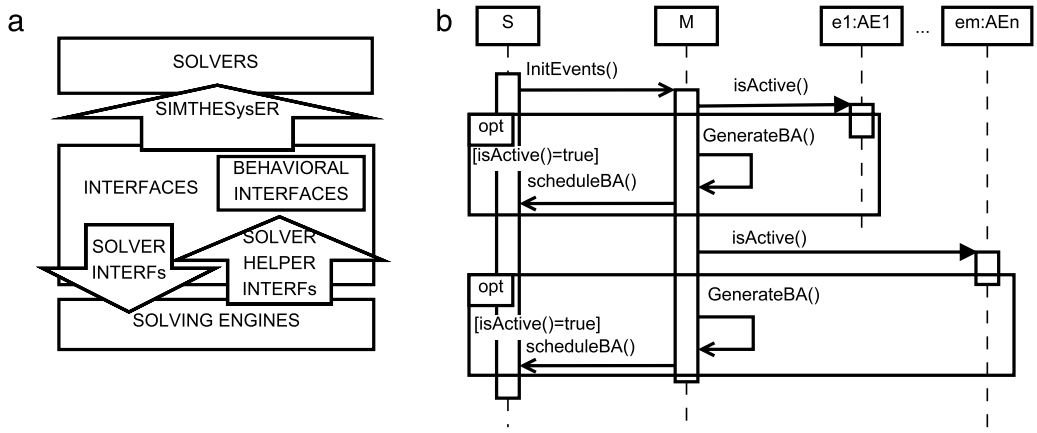


Fig. 4. (a) Solving stack and components relations. (b) Model generation sequence diagram.

framework support tool, has the capability to generate the desired solver by using *solving engines*, software components that implement basic solution algorithms. SIMTHESys*ER* (whose organization, limits and capabilities are described in [20]) exploits the *interfaces* to obtain the correct composition and to fill the semantic gap between formalisms and solving engines.

The resulting solvers can evaluate all models written by users that are compliant with the chosen formalism or formalism combination. Abusing naming conventions, it can be noted that while solvers generate results, SIMTHESys*ER* generates metaresults, being solvers the results of the analysis of FDL specifications. The structure of the solving stack and the relationships between its components are depicted in Fig. 4(a).

An exponential event solver *S* generates its internal representation of a model M as depicted in Fig. 4(b). *S* (exploiting *ExpEventModel*, in this example) requests M to identify the enabled events by calling *InitEvents( )*. This behavior loops through all the elements implementing the *isActive* behavior and invokes its occurrence. Behaviors are invoked by generating a *Behavior Adapter* using the *Generating Behavior* function. The solver can determine the changes in the model caused by an event and an exponential time after which the event occurs using the *Schedule* function. Repeated application of this sequence allows the solver to get all the needed information to run.

Being *InitEvents* defined by the formalisms as a behavior of its model element, the whole mechanism can be modified if needed, to match the logic of different formalisms.

3.1. Solving engines

Each solving engine is designed to provide a basic solution method to the framework. A solving engine evaluates one or more metrics, that are available to the framework by means of a proper *solver interface*. Formalisms designed over the same solving engines are said to belong to the same *formalism family*. The set of the available solving engines is open, so that the framework can be enriched by the development of completely new engines or by including external solvers. There is virtually no limitation over engines, provided that they offer a solver interface, but the more specialized is the solution method, the less wide the related formalism families can be.

The framework currently provides six solving engines for stochastic evaluation. These engines support two formalism families, named Exponential Events (EEF) and Exponential and Immediate Events (EIEF). They allow the computation of state rewards and impulse rewards performance indexes (similarly to [21]). State rewards compute performance measures as mean values of some function of the state of a model (mean length of a queue, mean number of tokens in a Petri net place). Impulse rewards compute performance indices in terms of the number of times in which an event occurs (throughput of queues or of Petri net transitions).

Se and *Sg* are classic event based simulators based on [22], with exponential interarrival times (without and with the possibility of immediate non stochastic events respectively). The execution is performed by a proper behavior that identifies all enabled events and draws the needed exponentially distributed samples.

Ce_s and Cg_s are designed to generate Continuous Time Markov Chains (CTMCs) and perform steady state analysis (respectively without and with the possibility of immediate non stochastic events). The engines are based on a state snapshot logic and exploit a behavior to obtain the needed information from the high-level formalism model. This process starts from the generation of the initial state, then it produces the transition graph of the CTMC by executing all the states including enabled transitions in each step, until all states that are generated are already present in a snapshot. The transition graph is then used to obtain the generator matrix, and the steady state solution vector is computed.

Ce_t and Cg_t are designed to generate Continuous Time Markov Chains (CTMCs) and perform transient analysis (respectively without and with the possibility of immediate non stochastic events), with similar approaches with respect to Ce_s and Cg_s.

Since the nature of the solving engines is independent from formalisms and from the solvers generation process and thanks to the presence of the interfaces, other engines of different nature based on different solution paradigms can be integrated in the framework.

3.2. Interfaces

The possibility of sharing common sets of behaviors between different elements is formalized through the concept of interfaces. Interfaces constitute the intermediate layer of the SIMTHESys solving stack. Interfaces are divided into three categories: *solver interfaces*, *solver helper interfaces* and *behavioral interfaces*. Interfaces are used to bind solving engines and formalisms.

Solver interfaces export the capabilities of solving engines towards formalisms. A formalism should use a solver interface in order to be able to define which engine is to be used when generating the solver and to specify the implementation of the methods used to invoke the engine. An example of use of a solver interface is in Fig. 3. In the figure, the SPN formalism element uses the *ExpEventSolver* solver interface (that is supported by *Se*, Ce_s and Ce_t).

Solver helper interfaces offer solving engines the hooks to obtain the information related to a specific model using a certain (multi)formalism. A formalism implements a solver helper interface by guaranteeing that a number of behaviors and properties are mandatory in all models written in the specified (multi)formalism. The behaviors offer the solver the means to access the information needed for the solution process by specifying how it can be computed by model parameters and state. The properties instruct the solver about how to construct the data structures needed by the solution process (e.g. the state space). Generally, a formalism family shares the same solver helper interface. An example of use of a solver helper interface is in Fig. 3. In the figure, the SPN formalism element implements the *ExpEventModel* solver helper interface (that is required by *Se*, Ce_s and Ce_t), by implementing behaviors *InitEvents*, *setStateRewards*, *setImpulseRewards*, *countStateRewards* and *computeStateRewards*. For the exponential event solving engines, *countStateRewards* is used to return the number of state rewards of a model; *computeStateRewards* is used to compute all the rewards in a given state (by using other behaviors of the various model elements to compute the values related to the specific primitives, such as the tokens in a place element to evaluate the *MeanTokens* property); *setStateRewards* is used by the solving engine to return computed indices (e.g. setting *MeanTokens*); *listImpulseRewards* is used to list the reward names of all the impulse rewards associated to a model; and *setImpulseRewards* is used similarly to *setStateRewards*.

Behavioral interfaces offer reusable facilities, in terms of pre-built sets of behaviors and properties. Such interfaces can provide functionalities that are typical of specific concepts (e.g. the arc in Petri nets-like models, or the communication protocols primitives in a formalism for computer networks modeling). They are used to implement easily similar modeling patterns in similar formalisms, or can aggregate complex sequences of simple behaviors into higher level behaviors, to offer the modeler an easier modeling paradigm for more abstract modeling domains. An example of implementation of a behavioral interface is in Fig. 3. In the figure, the *Transition* element uses the *PushPull* behavioral interface (typical of elements that take and produce a number of items by input and output arcs, and implemented by the *Arc* element) and implements the *Active* behavioral interface (typical of elements that need to be scheduled by event based solving engines) by implementing the *isActive* behavior.

The use of interfaces is of paramount importance for decoupling formalisms and solving engines and for allowing the design of hybrid formalism elements, thus the enactment of multiformalism modeling. In the first case, the same formalism can be used to build different solvers founded on different solving engines, simply by using a different solver interface. Also, different formalisms can be mapped onto a same solving engine, by implementing the same solver helper interface. In the second case, a hybrid element, allowing a multiformalism solver to be built, can be created by composing in it two different behavioral interfaces, typical of two formalisms, and describing the composition by properly mapping to each other related behaviors.

3.3. Solvers

As for models, general characteristics about solvers cannot be given, since their capabilities and use obviously depend on the solving engine(s) and the formalism(s) to which they are related. A SIMTHESys solver is a stand-alone software tool that is able to parse MDL documents describing models that conform to the specific (multi)formalism for which the solver has been generated. Besides the MDL documents, some solver could also need additional input data (e.g. if is designed to work in-the-loop with the real system that the model represents), in form of other documents or data streams. A solver generates results in form of documents.

Solvers are generated by SIMTHESys*ER*, a dedicated tool[4] that is one of the components of the SIMTHESys framework. The solver is synthesized by combining the specified formalism definitions, interfaces and solving engines as shown in Fig. 5 (more details about the steps to be carried out to get automatically a solver for the new multi-formalism can be found in [20]). SIMTHESys*ER* exploits the definition of the main formalism element of the (multi)formalism to select the needed solving

[4] The tool, written in Java with some components in C++ to enhance performance, is currently available and freely downloadable from the SIMTHESys web site www.dem.unina2.it/simthesys: see it also for examples and references.

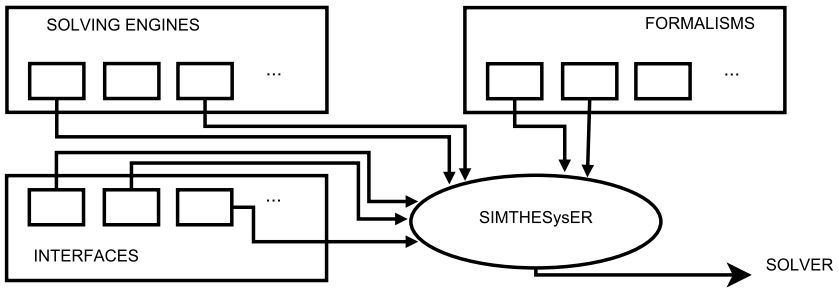


Fig. 5. The components needed to build a solver.

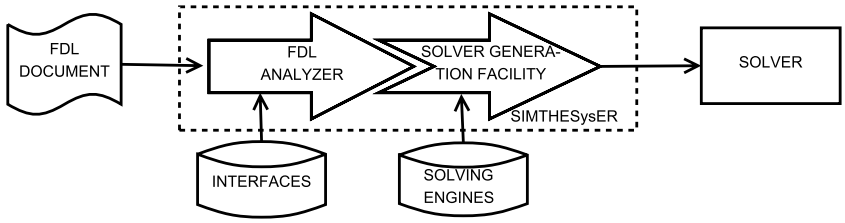


Fig. 6. The creation process of a solver.

engines by using the information obtained from the solver interfaces. The main cycle of the solver is built on the solver helper interface(s), while the internal mechanisms of the solver are obtained from the behavioral interfaces. This synthesis process is performed by the two main internal components of SIMTHESys*ER*, the FDL Analyzer and the Solver Generation Facility (see Fig. 6). The FDL Analyzer reads a FDL document and generates the MDL parser, building the classes that implement the description of the elements of the formalism(s) by exploiting the interfaces. The Solver Generation Facility consists of scripts that generates the code for the evaluation of formalism specific performance metrics, the code of main cycle of the solver by combining the interfaces and the solving engines, and the final solver.

Currently, with the set of solving engines discussed in 3.1, SIMTHESys*ER* has been used for the automatic generation of solvers for Stochastic Petri Nets (SPN) [23], Stochastic Petri Nets with exception handling, Generalized Stochastic Petri Nets (GSPN) [23], Generalized Stochastic Petri Nets with exception handling, Markov Chains (MC), Queuing Networks (QN) [24], Finite Capacity Queuing Networks (FCQN) [25], Fault Trees (FT) [26], a stochastic verification formalism inspired to stochastic automata [4] and custom multi-formalisms based on their composition.

4. Using SIMTHESys

This section describes an example of a new formalism that can be used to add the features of software rejuvenation [27] to a performance model written in any formalisms that can be solved by the EEF or the EIEF engines. The importance of including all the elements related to software rejuvenation in a performance model in modular way was addressed in [28] and later extended in [29]. Specifically, the latter addressed the possibility of identifying blocks that can describe the various features of the rejuvenation, and use them to produce high-level models of the feature. However, in that work the various features had to be converted manually by the user in Fluid Stochastic Petri Nets [30], and, although suggested, no automatic conversion procedure was proposed.

In this section, those concepts are extended by creating a new formalism called *SR* (Software Rejuvenation), that adds specific primitives to model the preventive rejuvenation of software systems. In the case presented in this paper, phase-type distributions [31] are used. In order to show the flexibility of the proposed approach, the scope is limited to the software system, where rejuvenation is implemented by restarting the application (note that this approach can represent a limitation when modeling a real system). The formalism includes three new primitives: the *Degradation*, the *Rejuvenation* and the *Crash* and a new arc called *RateControl*. The *Degradation* models the system degradation, i.e. the aging of the system. It is implemented by a sequence of exponential events according to the Markov Chain shown in Fig. 7. Each state i represents a different degradation level ϕ_i: in particular $0 \leq \phi_i \leq 1$ is a factor that slows down the exponential firing time of the events to which the degradation is connected using a *RateControl* arc. The mean sojourn in each state i is exponentially distributed with a mean time of $1/\lambda_i$. At the end of stage i, the models jump to the next state $i+1$, up to a maximum degradation level represented by state n where the system remains until it crashes or it is rejuvenated.

The system can experience failures, that are modeled by the *Crash* primitive. Each crash primitive alternates between two states: working and under repair. The sojourn time in the first state, which represents the *Mean Time To Failure* (MTTF), is distributed according to a phase type distribution with a total of n phases. The jump rate from phase i to $i+1$ is λ_i^C, and the probability of crashing is ψ_i^C. Similarly, the sojourn time in the second state, that represents the *Mean Time To Repair*

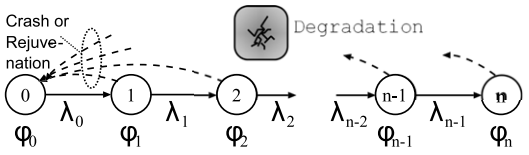


Fig. 7. Degradation.

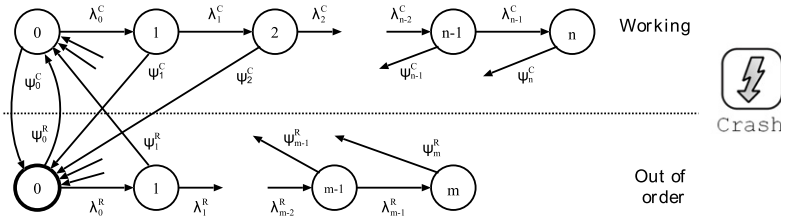


Fig. 8. Crash.

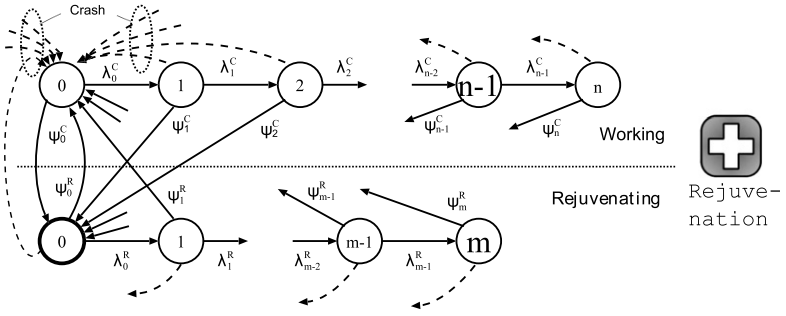


Fig. 9. Rejuvenation.

(MTTR), is represented also by a phase type distribution, with parameters m, λ_i^R and ψ_i^R. The Markov chain that describes the Crash primitive is shown in Fig. 8.

Finally, to reduce the faults, the system might execute some preventive restoration by restarting the service. This is modeled by the *Rejuvenation* component, which alternates between two states that represent the normal operation and the duration of the Rejuvenation. The sojourn time in both states is also phase type distributed with parameters n, λ_i^C, ψ_i^C, m, λ_i^R and ψ_i^R respectively, and their underlaying Markov chain is represented in Fig. 9. The various elements included into a SR sub-model can interact together according to the rules described in Table 1. Specifically, each time a *Crash* occurs, the system identifies the fault and restarts the system. During this period, the system is not working, and degradation or rejuvenation are blocked. As soon as the system resume working, both degradation and rejuvenation restart their timing by resetting their phase to the initial one. It is assumed that crashes cannot occur during the repair of a faulty system. Under the rejuvenation, the degradation is reset to its initial phase to model the effect of restarting the software. A crash may happen during the rejuvenation; this occurrence does not reset the phase of the crash and it does not block the corresponding event. However, the interaction between the degradation and the crash has been modeled. The actual rate at which the phase can change (and the event can occur in a phase) for the crash block, has been made inversely proportional to the degradation: in this way, the more the system is degraded, the higher is the possibility of having a crash. Under this hypothesis, the rejuvenation has also an indirect effect that reduces the crash probability by reducing the rate at which the faulty event can happen.

The previous primitives can be employed to control a model with respect of other formalisms by using still the same set of properties that characterize the software rejuvenation. For example, in Fig. 10, the SR primitives are used to control a simple server (with a single buffer position), modeled by an SPN. In this case, a timed transition *Srv* represents the service of a customer, whose arrival is determined by the firing of transition *Arr*. The service transition is controlled by the SR sub-model which introduces the software rejuvenation. The relations between the two models are defined by the inhibitor (*CheckLT*), test (*CheckGE*) and rate control (*RateControl*) arcs. In particular, the rate control arc that connects the degradation to the *Srv* transition models the effect of the aging of the component: transition *Srv* is slowed down by the degradation factor corresponding to the current degradation phase. When the system is stopped, either for a crash or for an ongoing restart, the server transition cannot fire due to presence of the *CheckLT* arc that connects the SR block to the transition. Since the SR block implements the *getOccupancy* behavior by returning 1 if the system is either crashed or rejuvenating, this correctly models the desired feature. Finally, in order to model the fact that the system can only crash or degrade when it is working, two *CheckGE* arcs connect the place containing the client currently in service (the token that enables transition

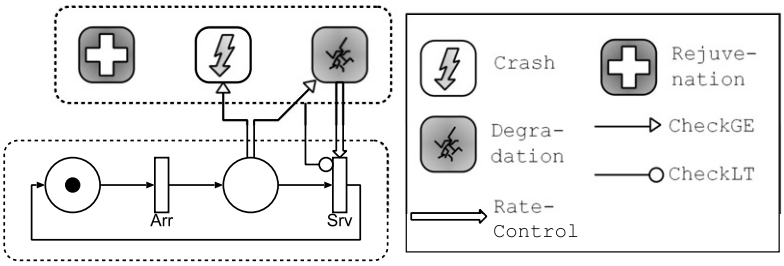


Fig. 10. Model.

Table 1
Interaction among the events of the SR formalism.

Event	Deg. phase	Rej. phase	Rej. state	Deg.	Rej.	Crash
Crash	0	0	Normal	Blocked	Blocked	NA
Rejuv.	0	NA	NA	Blocked	NA	Blocked

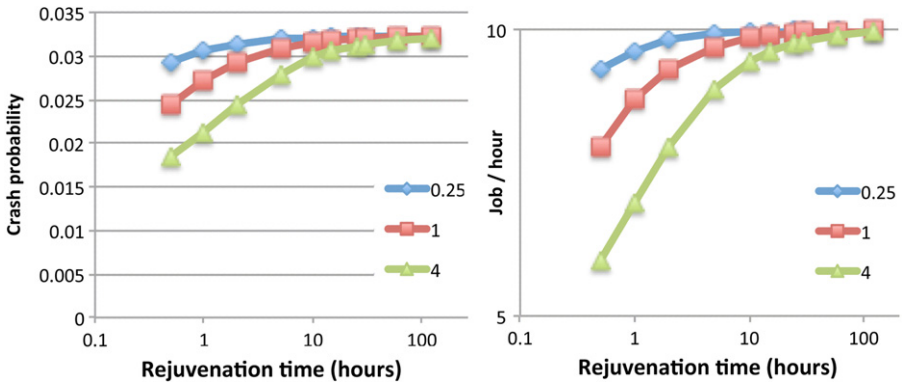


Fig. 11. Crash probability and throughput for different times between the rejuvenation, and different rejuvenation lengths.

Srv) to the *Crash* and the *Degradation* elements of the SR model. Since all these interactions are governed by behaviors, the same primitives can be used to control other formalisms, like a queuing network, in the same way.

To prove the efficacy of the model using the methods described in Sections 2 and 3, Fig. 11 shows the crash probability and the response time that have been obtained by varying both the time between two consecutive rejuvenation, and the duration of the rejuvenation. As it can be seen, the throughput increases, but also the crash probability becomes larger by increasing the time between two resets.

5. Conclusions and future work

This paper has outlined the SIMTHESys approach, and shown its applicability by presenting a way to add software rejuvenation to a model written in a different formalism. The approach, although relatively new, has proven to be very flexible in several tests due to the architecture of the behaviors that make the framework parameterizable. Currently, SIMTHESys supports the extension of common formalisms like Petri Nets, Fault Trees and Queuing Networks, to include new features such as testing specific conditions, throwing and catching exceptions, and implementing software rejuvenation policies. Our current work focuses on the development of new solution components that allow the construction of formalisms which are not only based on exponential events, but also on other features including continuous variables (i.e.fluid and hybrid models), or spatial distributions (i.e. Markovian agents and cellular automata).

References

[1] M. Gribaudo, D.C. Raiteri, G. Franceschinis, Drawnet, a customizable multi-formalism, multi-solution tool for the quantitative evaluation of systems, in: QEST, 2005, pp. 257–258.
[2] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework, second ed., Addison–Wesley Professional, 2008.
[3] E. Barbierato, M. Gribaudo, M. Iacono, Defining formalisms for performance evaluation with simthesys, Electric Notes Theoretical Computer Science 275 (2011) 37–51.
[4] E. Barbierato, M. Gribaudo, M. Iacono, Exploiting multiformalism models for testing and performance evaluation in simthesys, in: Proceedings of 5th International ICST Conference on Performance Evaluation Methodologies and Tools — VALUETOOLS 2011, 2011.

[5] E. Barbierato, M. Gribaudo, M. Iacono, S. Marrone, Performability modeling of exceptions-aware systems in multiformalism tools, in, in: K. Al-Begain, S. Balsamo, D. Fiems, A. Marin (Eds.), ASMTA, in: Lecture Notes in Computer Science, vol. 6751, Springer, 2011, pp. 257–272.
[6] M. Iacono, M. Gribaudo, Element based semantics in multi formalism performance models, in: MASCOTS, IEEE, 2010, pp. 413–416.
[7] J. de Lara, H. Vangheluwe, Atom3: A tool for multi-formalism and meta-modelling, in: R.-D. Kutsche, H. Weber (Eds.), FASE, in: Lecture Notes in Computer Science, vol. 2306, Springer, 2002, pp. 174–188.
[8] V. Vittorini, M. Iacono, N. Mazzocca, G. Franceschinis, The osmosys approach to multi-formalism modeling of systems, Software and System Modeling 3 (1) (2004) 68–81.
[9] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J.M. Doyle, W.H. Sanders, P. Webster, The mobius modeling tool, in: Proceedings of the 9th International Workshop on Petri Nets and Performance Models, PNPM'01, IEEE Computer Society, Washington, DC, USA, 2001, p. 241.
[10] K.S. Trivedi, Sharpe 2002: Symbolic hierarchical automated reliability and performance evaluator, in: DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks, IEEE Computer Society, Washington, DC, USA, 2002, p. 544.
[11] G. Ciardo, R.L. Jones III, A.S. Miner, R.I. Siminiceanu, Logic and stochastic modeling with smart, Performance Evaluation 63 (2006) 578–608.
[12] F. Bause, P. Buchholz, P. Kemper, A toolbox for functional and quantitative analysis of deds, in: Proceedings of the 10th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools, TOOLS '98, Springer-Verlag, London, UK, 1998, pp. 356–359.
[13] J. Bézivin, On the unification power of models, Software and System Modeling 4 (2) (2005) 171–188.
[14] J.P. Van Gigch, System Design Modeling and Metamodeling/John P. van Gigch, Plenum Press, New York, 1991.
[15] M.A. Jeusfeld, M. Jarke, J. Mylopoulos (Eds.), Metamodeling for Method Engineering, MIT Press, Cambridge, MA, USA, 2009.
[16] J.D. Poole, Model-Driven Architecture: Vision, Standards, and Emerging Technologies. Position Paper Submitted to ECOOP 2001, 2001.
[17] O.M. Group, Unified modeling language standards version 2.3. URL http://www.omg.org/spec/UML/2.3/.
[18] H.M. Gholizadeh, M.A. Azgomi, A meta-model based approach for definition of a multi-formalism modeling framework, International Journal of Computer Theory and Engineering 2 (1) (2010) 87–95.
[19] M. Uschold, M. Grninger, M. Gruninger, Ontologies: principles, methods and applications, Knowledge Engineering Review 11 (1996) 93–136.
[20] E. Barbierato, M. Gribaudo, M. Iacono, Simthesyser: a tool generator for the performance evaluation of multiformalism models, Tech. Rep., Università degli Studi di Napoli, Belvedere Reale di San Leucio 81100 Caserta, Italy, February 2012.
[21] D.D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J.M. Doyle, W.H. Sanders, P.G. Webster, The mobius framework and its implementation, 2002.
[22] S. Robinson, Simulation: The Practice of Model Development and Use, John Wiley & Sons, Inc., New York, NY, USA, 2004.
[23] D. Kartson, G. Balbo, S. Donatelli, G. Franceschinis, G. Conte, Modelling with Generalized Stochastic Petri Nets, John Wiley & Sons, Inc., New York, NY, USA, 1994.
[24] S. Balsamo, V.D.N. Personè, P. Inverardi, A review on queueing network models with finite capacity queues for software architectures performance prediction, Performed Evaluation 51 (2–4) (2003) 269–288.
[25] W. Gordon, G. Newell, Closed queueing systems with exponential servers, Operations Research 15 (2) (1967) 254–265.
[26] D.C. Raiteri, M. Iacono, G. Franceschinis, V. Vittorini, Repairable fault tree for the automatic evaluation of repair policies, in: DSN, IEEE Computer Society, 2004, pp. 659–668.
[27] S. Garg, A. Puliafito, M. Telek, K. Trivedi, Analysis of software rejuvenation using markov regenerative stochastic petri net, in: Software Reliability Engineering, 1995. Proceedings, Sixth International Symposium on, 1995, pp. 180–187.
[28] A. Bobbio, S. Garg, M. Gribaudo, A. Horvath, M. Sereno, M. Telek, Modeling software systems with rejuvenation, restoration and checkpointing through fluid stochastic petri nets, in: Petri Nets and Performance Models, 1999. Proceedings. The 8th International Workshop on, 1999, pp. 82–91.
[29] A. Bobbio, S. Garg, M. Gribaudo, A. Horvath, M. Sereno, M. Telek, Compositional fluid stochastic petri net model for operational software system performance, in: Software Reliability Engineering Workshops, 2008. ISSRE Wksp 2008. IEEE International Conference on, 2008, pp. 1–6.
[30] G. Horton, V.G. Kulkarni, D.M. Nicol, K.S. Trivedi, Fluid stochastic petri nets: Theory, applications, and solution techniques, European Journal of Operational Research 105 (1) (1998) 184–201.
[31] C.A. O'Cinneide, Characterization of phase-type distributions, Communications in Statistics. Stochastic Models 6 (1) (1990) 1–57.