Modeling Hybrid Systems in SIMTHESys

Article *in* Electronic Notes in Theoretical Computer Science · October 2016

DOI: 10.1016/j.entcs.2016.09.021

3 authors:

Enrico Barbierato
Amedeo Avogadro University of Eastern Piedmont
24 PUBLICATIONS **244** CITATIONS
SEE PROFILE

Marco Gribaudo
Politecnico di Milano
123 PUBLICATIONS **1,000** CITATIONS
SEE PROFILE

Mauro Iacono
Università degli Studi della Campania "Luigi Vanvitelli
87 PUBLICATIONS **764** CITATIONS
SEE PROFILE

Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 327 (2016) 5–25

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Modeling Hybrid Systems in SIMTHESys

Enrico Barbierato[1] Marco Gribaudo[2]

Politecnico di Milano,
via Ponzio 34/5, 20133 Milano (Italy)

Mauro Iacono[3]

Seconda Università degli Studi di Napoli
viale Ellittico, 81100 Caserta (Italy)

Abstract

Hybrid systems (HS) have been proven a valid formalism to study and analyze specific issues in a variety of fields. However, most of the analysis techniques for HS are based on low-level description, where single states of the systems have to be defined and enumerated by the modeler. Some high level modeling formalisms, such as Fluid Stochastic Petri Nets, have been introduced to overcome such difficulties, but simple procedures allowing the definitions of domain specific languages for HS could simplify the analysis of such systems. This paper presents a stochastic HS language consisting of a subset of *piecewise deterministic Markov processes*, and shows how SIMTHESys - a compositional, metamodeling based framework describing and extending formalisms - can be used to convert into this paradigm a wide number of high-level HS description languages. A simple example applying the technique to solve a model of the energy consumption of a data-center specified using Queuing Network and Hybrid Petri Nets is presented to show the effectiveness of the proposal.

Keywords: Performance evaluation, hybrid systems, metamodeling.

1 Introduction and related works

HS represent an extremely flexible formalism and a successful research field, as a wide number of problems belonging to different areas - such as biology, networks, telecommunications and many others - have been studied and analyzed through hybrid models. This article considers a subset of the Piecewise Deterministic Markov Processes [12], a set of stochastic models characterized by randomness appearing in the form of point events.

The relevance of HS in different fields is testified by the longevity of related studies and the diversity in approaches that can be found in literature. A good

[1] enrico.barbierato@polimi.it
[2] gribaudo@elet.polimi.it
[3] mauro.iacono@unina2.it

http://dx.doi.org/10.1016/j.entcs.2016.09.021
1571-0661/

survey summarizing the main issues in the first three decades of studies is provided by [2], which gives a very good general introduction of the field, with both conceptual definitions and examples. In [27] a control systems oriented point of view on the field is given, including a wide bibliography. Non expert readers can find a good introduction in [25]. In [2] a list of software solutions offering tools to study HS is available: here we will just mention Ptolemy [24], HyTech [17], UPPAAL [22] and Prism [21], as we explored their characteristics to have a conceptual benchmark while shaping our approach.

This work aims at providing a basis to allow flexible modeling of HS by means of a framework enabling the definition of custom modeling formalisms, fitting at best the applications. The main contribution of this work is not the definition of just another hybrid modeling formalism, but to show a generic method from which high level hybrid models can be analyzed by mean of a suitable interchange format. At the best of our knowledge, this is an original contribution to the field.

For the purposes of this work, besides the mathematical aspects characterizing the possible descriptions of hybrid systems (see [12]), the aspects related to their operational semantics play a relevant role in supporting the development of a general modeling framework for a class of formalisms suitable for their representation and analysis. In this sense, [23] offers an interesting perspective. Important contributions are also given by the theory of Hybrid Automata (HA) [16] and the theory of Continuous and Hybrid Petri Nets [1], which also inspired our research.

The roots of the paper can be found into our studies on Hybrid Petri Nets (HPN) [14][13][15] and on multiformalism modeling [26][4]. The approach presented in this paper distills the essential aspects of Hybrid Petri Nets semantics and analysis, abstracting them to obtain the mechanisms to define custom formalisms with hybrid characteristics.

The present approach is founded onto the SIMTHESys multiformalism modeling framework. SIMTHESys (Structured Infrastructure for Multiformalism modeling and Testing of Heterogeneous formalisms and Extensions for SYStems) is a framework including solution engines to evaluate performance indexes by simulation or with suitable numerical techniques. The framework allows the definition of custom modeling languages, named *formalisms*, which are organized into families according to the types of techniques that are needed to analyze a model written following the corresponding semantic. Currently, SIMTHESys supports: the design of classical formalisms such as SPN, Tandem Finite Capacity Queueing Networks (TFCQN) and Gordon and Newell Queueing Networks (GNQN), and multiformalism models based on them [20]; multiformalism performance oriented models allowing testing against some conditions [5]; the formal definition of formalism enriched with an exception handling mechanism [8]; two approaches (of increasing complexity) adding software rejuvenation features to performance models [3] [19]; a performance evaluation oriented modeling language for SOA BPEL applications [9]; a performance evaluation oriented modeling language for Map-Reduce applications [6][7]; formalisms suitable for the implementation of product-form solution theory based analysis techniques, allowing compositional multiformalism modeling [10].

The paper is organized as follows: Section 2 introduces a running example to present the work along the paper; Section 3 introduces the reference formal definitions for the target HS class; Sections 4 and 5 present our approach; Sections 6 and 7 introduce the basis and the results of this work; Section 8 demonstrates the approach on the running example; finally, conclusions are drawn.

2 A running example

To start pointing out the nature of the systems this work aims to consider and analyze, let us focus on a running example, consisting of the characterization of the simplified data center shown in Fig.1a. In particular, the focus is on the study of the total energy required to run the servers, considering both IT resources and cooling. The more the servers are loaded, the more the temperature increases. In normal operating conditions, the data center is cooled using an air flow generated by a fan. If temperature reaches a given threshold, the data center turns on an air conditioning system (AC) to cool down the server room. If cooling down is insufficient and the room reaches a critical temperature, servers are shut down to prevent hardware damage. Moreover, the AC might fail to start, forcing a preventive shut down of all the system to allow for repair. It is supposed that user demands alternate at Poisson rates λ_L and λ_H between a high and a low workload.

The temperature of the room is modeled by a continuous variable x. Temperature can rise due to the heating produced by the servers at different rates depending on the workload: a_H during the high demand period, and a_L otherwise. Temperature can decrease due to the effect of both the fan (decreasing at rate d_L) and the AC (rate d_H). The fan is always active, while the AC is turned on at rate μ_{AC} only if the temperature reaches the threshold f_{Max}. When the temperature drops below threshold f_{Min}, the system can switch back to fan cooling at rate μ_{Fan}. The room has a base temperature f_{Room}, and must stay below a critical temperature $f_{Critical}$. The rate at which the start of AC might fail is μ_S. In this case the servers are immediately shut down to allow repair, which takes an exponentially distributed amount of time characterized by rate μ_R. Repairing will leave a random temperature in the room, to account for the fact that during the process both the AC and the servers might be started and stopped several times. To model this issue, we will not consider the room temperature during the repair process, and we will set the level x at which the system will restart to be distributed according to a truncated normal distribution centered on the average room temperature, and characterized by a given variance γ. The system will always restart with the low demand, to allow testing before increasing the demand again.

3 The considered Hybrid Systems

As a basis for our framework, a subset of the Piecewise Deterministic Markov Processes introduced in [12] is considered. In order to simplify the description, in the following this subset will be addressed as Hybrid System Modeling Language

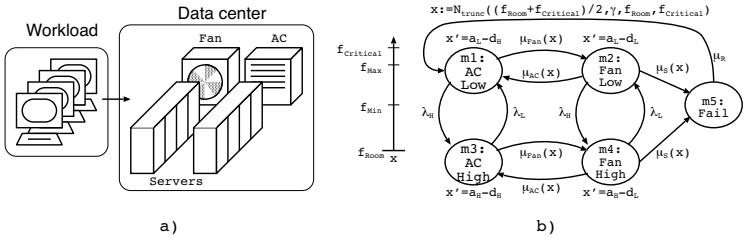

Fig. 1. A simplified data center example: a) the system, b) the corresponding HSML model.

(HSML), and HS will be used to denote hybrid systems in general: i.e. real systems that could be modeled using PDMPs. Our class of HS is characterized by a discrete and finite set of modes $\mathcal{M} = \{m_1, \ldots, m_M\}$. For each mode m_i, the system is characterized by a finite number d_i of continuous *variables* $x_{i,j}$ (with $1 \leq j \leq d_i$), each defined on a compact subset of $\mathbb{R}$ characterized by a lower boundary $l_{i,j}$ and an upper boundary $u_{i,j}$. The *continuous domain* of a mode m_i is called D_i, and defined as:

$$D_i = \bigtimes_{j=1}^{d_i} [l_{i,j}, u_{i,j}] \tag{1}$$

where $\times$ represents the cartesian product of the corresponding sets. Each state is thus defined by a tuple σ:

$$\sigma = (m_i, x_{i,1}, \ \ldots \ , \ x_{i,d_i}) \ \text{ with } m_i \in \mathcal{M} \text{ and } x_{i,j} \in [l_{i,j}, u_{i,j}] \tag{2}$$

To simplify the notation, we define with $\mathbf{x}_i = (x_{i,1}, \ \ldots \ , \ x_{i,d_i})$, and the state becomes $\sigma = (m_i, \mathbf{x}_i)$. The state space of the model $\mathcal{S}$ is defined as:

$$\mathcal{S} = \{\sigma\} = \bigcup_{i=1}^{M} (\{m_i\} \times D_i) \tag{3}$$

An HSML model is defined by a tuple $(\mathcal{S}, \Phi, E, \Lambda, \Psi)$, where $\mathcal{S}$ is the state space, as defined in Eq.3. $\Phi = \{\phi_1, \ldots, \phi_M\}$ are the *continuous evolution functions*. Specifically, for every mode $m_i \in \mathcal{M}$, function ϕ_i is defined as:

$$\phi_i : D_i \times \mathbb{R} \to D_i \tag{4}$$

Let us suppose that in a time interval $[a, b]$ the system visits only continuous states belonging to the same mode m_i. Let us denote with $\sigma(t) = (m_i, \mathbf{x}_i(t))$ the state at time $a \leq t \leq b$. The continuous part of the states evolves according to function ϕ_i. In particular:

$$\sigma(\alpha) = (m_i, \mathbf{x}_i) \implies \sigma(\beta) = (m_i, \phi_i(\mathbf{x}_i, \beta - \alpha)), \quad \forall a \leq \alpha \leq \beta \leq b \tag{5}$$

It can be easily shown that by definition we must have $\phi_i(\mathbf{x}_i, 0) = \mathbf{x}_i$. The random evolution of the model is governed by a set of N possible *events* $E = \{e_1, \ldots, e_N\}$.

The rate at which events can occur is defined by function Λ:

$$\Lambda : S \times E \to \mathbb{R}_0^+ \tag{6}$$

In particular, in any time interval $\Delta t \to 0$ we have that an event $e_k \in E$ can occur in state σ with probability:

$$Pr\{e_k \text{ occurs in state } \sigma \text{ during } \Delta t\} = \Lambda(\sigma, e_k) \cdot \Delta t + o(\Delta t) \tag{7}$$

Finally, Ψ describes the effects of an event to the state of the system. It is defined as:

$$\Psi : S \times E \times S \to [0, 1] \tag{8}$$

Specifically, $\Psi(\sigma'|e_k, \sigma)$ defines the probability that the system will jump to state σ', conditioned to the occurrence of event e_k in state σ. More formally, we have that:

$$\begin{aligned} &Pr\{\sigma' \text{ with } m'_j = m_j, x'_{j,1} \leq x_{j,1}, \ \ldots \ , \ x'_{j,d_j} \leq x_{j,d_j} | e_k, \sigma\} = \\ &= \Psi(m_j, x_{j,1}, \ \ldots \ , \ x_{j,d_j} | e_k, \sigma) \end{aligned} \tag{9}$$

To simplify the notation, it is possible to write $\Psi(e_k, \sigma) = DIST(\sigma)$, where $DIST(\sigma)$ is a valid probability distribution that might be parameterized on the current state σ. For example, $\Psi(e_k, m_i, \mathbf{x}_i) = DET(m_l, \mathbf{x}_i)$ denotes an event e_k that changes deterministically (DET) the mode from m_i to m_l leaving the continuous variables unchanged (provided that $d_i = d_l$). It is a shorthand notation for:

$$\Psi(m_j, \mathbf{x}_j | e_k, m_i, \mathbf{x}_i) = \begin{cases} 1 \text{ if } m_j = m_l \wedge \mathbf{x}_i \leq \mathbf{x}_j \\ 0 \ otherwise \end{cases} \tag{10}$$

where $\mathbf{x}_i \leq \mathbf{x}_j$ denotes the element-wise comparison of the elements of vectors $\mathbf{x}_i$ and $\mathbf{x}_j$.

3.1 Application to the running example

Let us model the data center described in Section 2 using the proposed HSML. As shown in Fig.1b, the model is characterized by $M = 5$ modes: `AC-Low` (m_1), `Fan-Low` (m_2), `AC-High` (m_3), `Fan-High` (m_4) and `Fail` (m_5). Modes m_1 to m_4 have a single fluid component x_1, bounded between $l_{i,1} = f_{Room}$ and $r_{i,1} = f_{Critical}$, while mode m_5 do not have any continuous variable associated.

The state space for our running example is thus:

$$\mathcal{S} = \{m_1, m_2, m_3, m_4\} \times [f_{Room}, f_{Critical}] \cup \{m_5\} \tag{11}$$

The fluid evolution accounts for the changes in the users demand and in the type of cooling being used, and functions Φ are defined as follows:

$$\begin{aligned}
\phi_1(x_1,t) &= \min(x_1 + (a_H - d_L)\cdot t, f_{Critical}) \\
\phi_2(x_1,t) &= \max(x_1 + (a_L - d_L)\cdot t, f_{Room}) \\
\phi_3(x_1,t) &= \min(x_1 + (a_H - d_H)\cdot t, f_{Critical}) \\
\phi_4(x_1,t) &= \max(x_1 + (a_L - d_H)\cdot t, f_{Room})
\end{aligned} \tag{12}$$

The change in the modes is governed by $N = 6$ events: `High` (e_1) and `Low` (e_2) when the system jumps from low to high and from high to low demand, $\texttt{Start}_{AC}$ (e_3) and $\texttt{Stop}_A C$ (e_4) to describe the start and the stop of the AC, `Fail` (e_5) and `Repair` (e_6) to model the failures and the repairs. The AC can start only if the temperature is more than f_{Max}, and it can stop only if it drops below f_{Min}. Since the AC can fail only when it is starting, event e_5 can only take place if the temperature is greater than f_{Max}. If we call $\mathbf{1}(Y)$ the indicator function that returns 1 if predicate Y is true, and 0 if it is false, function Λ is defined as follows:

$$\begin{aligned}
\Lambda(m_i,x_1,e_1) &= \lambda_H \cdot \mathbf{1}(i \in \{1,2\}) \\
\Lambda(m_i,x_1,e_2) &= \lambda_L \cdot \mathbf{1}(i \in \{3,4\}) \\
\Lambda(m_i,x_1,e_3) &= \mu_{AC} \cdot \mathbf{1}(i \in \{2,4\} \wedge x_1 \geq f_{Max}) \\
\Lambda(m_i,x_1,e_4) &= \mu_{Fan} \cdot \mathbf{1}(i \in \{1,3\} \wedge x_1 \leq f_{Min}) \\
\Lambda(m_i,x_1,e_5) &= \mu_S \cdot \mathbf{1}(i \in \{2,4\} \wedge x_1 \geq f_{Max}) \\
\Lambda(m_i,x_1,e_6) &= \mu_R \cdot \mathbf{1}(i = 5)
\end{aligned} \tag{13}$$

Finally, function Ψ changes the fluid component x_1 only in mode m_5 to account for the restart after the failure. TO summarize:

$$\Psi(e_1,m_i,x_1) = \begin{cases} DET(m_3,x_1) & m_i = m_1 \\ DET(m_4,x_1) & m_i = m_2 \end{cases} \tag{14}$$

$$\Psi(e_2,m_i,x_1) = \begin{cases} DET(m_1,x_1) & m_i = m_3 \\ DET(m_2,x_1) & m_i = m_4 \end{cases}$$

$$\Psi(e_3,m_i,x_1) = \begin{cases} DET(m_2,x_1) & m_i = m_1 \\ DET(m_4,x_1) & m_i = m_3 \end{cases}$$

$$\Psi(e_4,m_i,x_1) = \begin{cases} DET(m_1,x_1) & m_i = m_3 \\ DET(m_2,x_1) & m_i = m_4 \end{cases}$$

$$\begin{aligned}
\Psi(e_5,m_i,x_1) &= DET(m_5) \\
\Psi(e_6,m_5) &= DET(m_1) \times N_{trunc}((f_{Room} + f_{Critical})/2, \gamma, f_{Room}, f_{Critical})
\end{aligned}$$

where $N_{trunc}((f_{Room} + f_{Critical})/2, \gamma, f_{Room}, f_{Critical})$ denotes that the next state will have the mode deterministically set to m_1, and the temperature randomly distributed according to a normal distribution with mean $(f_{Room} + f_{Critical})/2$ and variance γ, truncated between f_{Room} and $f_{Critical}$.

4 Defining high-level hybrid system languages

The example provided in Section 2 shows how models for very simple systems may require a very complex description. The proposed formalization is very hard to use from a modeler's perspective due to several reasons. The simplest to address is the use of numbers to identify modes and events: this however can be overcame quite easily by using proper symbolic representations of the considered quantities. More subtle is the difficulty in defining in a consistent way all the functions involved (i.e. Λ, Φ and Ψ). Moreover, due to the similarity of the considered formalism to automata, both the number of states and events grows exponentially in the complexity of the model.

Thus we propose an integration of the considered type of HS in the SIMTHESys framework to automatically generate all the elements of the HSML tuple from an high-level description of the considered system. In order to describe how hybrid continuous-discrete paradigms can be implemented and translated into the considered HSML language in a way that allows the solution of multiformalism models, we will first recall how the object-oriented SIMTHESys methodology works on standard discrete multiformalism system. In particular, we will present how SPN and simple exponential Queuing Networks (QN), can be implemented in SIMTHESys. We will then extend such concepts to include continuous state components and we will apply them to implement a multiformalism language allowing cooperation between the hybrid counterparts of the two reference discrete formalisms. For what concerns SPN, we will focus on Hybrid Petri Nets (HPN), a special type of Fluid Stochastic Petri Nets (FSPN) [18]. For what concerns QN, the use of HSML allows extend the proposed formalism including non-exponential FCFS service centers. Even if such extension does not describe proper HS since the apparent state space of the model is only discrete, the inclusion of non-exponential transitions must be implemented using supplementary variables, thus leading to a model requiring the considered underlying HSML definition to be properly solved.

Note that a preliminary approach to modeling HS in SIMTHESys was given in [3] to add rejuvenation to PN or TFCQN models. However, in that case, the continuous part was limited to model aging of the components. This limitation allowed such models to be solved by first translating them into labeled transition systems [10], and then by externally attaching a solver handling the related continuous part. The approach proposed in this paper greatly extends such results, by allowing the representation of a more general class of hybrid systems.

5 SIMTHESys approach to multiformalism modeling

The SIMTHESys framework consists of i) a metamodeling [4] structure, ii) a set of interfaces and iii) a solving architecture. A user builds a model as composition of *submodels* where each one may be written in a (possibly different) formalism and

[4] Metamodeling studies the rules and the structures that specify models. A metamodel can be defined an abstraction of a class of models, considering that a model abstracts the real world.

tied together in a composition by using metamodeling approach.

In SIMTHESys, formalisms are described by Formalism Description Language (FDL, based on XML) documents, which define all their modeling primitives. The FDL follows an object-oriented approach, in which elements can be characterized by interfaces that allow the interchange of objects as long as they provide the same interface.

The basic part of a formalism is the *Element*, defining all the atomic primitives describing a model. *Formalism Elements* are used to define submodels, and can contain other elements. An *Element* is characterized by *Properties* and *Behaviors*. Properties associate values of given types to the elements of a formalism. Behaviors define the actions that the element performs. For example, the following properties characterize the *Place* element of a SPN:

```
<propertyType name="id" type="String" default="" storage="static"/>
<propertyType name="Tokens" type="int" default="0" storage="dynamic"/>
<propertyType name="MeanTokens" type="Result" storage="computed"/>
```

where *static* denotes a constant value, *dynamic* indicates a value instantiated by the model that may change during the system evolution, and *computed* refers to a value calculated by the solving engine. Each property automatically defines *getter* and *setter* methods that can be called by the code defining the behaviors to read or write the corresponding value. For example *getTokens()* returns the value of the *Tokens* property associated with a place. The value of the same property can be set using the *setTokens(int v)* method. With regard to *Behaviors*, the following example determines the current number of tokens in a SPN *Place*:

```
<behavior name="getOccupancy" return="int">
<code>
       return getTokens();
</code>
</behavior>
```

Interfaces provide the capability to share common sets of behaviors used by different elements. The main idea of interfaces consists of binding formalisms and solving engines. SIMTHESys exploits three types of interfaces: solver interfaces (to define which solving engine should be used), solver helper interfaces (to guarantee that a set of behaviors and properties are mandatory in all models produced in the specified formalism) and behavioral interfaces (to reuse existing abstractions).

A *Model* is declared by a document written in Model Description Language (MDL) and it must conforms to one (or more) formalism. In particular, a model instantiates a set of elements: for each element, it specifies the initial value for all the dynamic properties, and assigns a constant value to all the static properties.

5.1 SIMTHESys Solving Engines

Beside the FDL Analyzer, SIMTHESys provides also a set of six solution techniques for two formalism families (exponential events and exponential and immediate events based), and exploits six different solution engines. The six solution techniques are based either on discrete event simulation or on state space generation. The foundation of the latter engines consists of a state snapshot logic and

uses a behavior to retrieve the requested information from the high-level formalism model. In the first phase, the process builds up the transition graph of the model (which in most of the case is a CTMC) by running all the states considering in each step those events that are enabled, stopping when all the produced states can be found in a snapshot. In the next phase, the generator matrix is derived from the transition graph. Finally, the steady state solution vector is computed. Discrete event simulation engines instead use behaviors to choose one possible evolution, and produce performance indices by averaging several traces.

5.2 Formalism families

A *formalism family* is a set of Formalisms that can be handled over the same solving engines. They also share a set of similarities, which can be exploited to automatically generate the multiformalism behavior from the specification of the way in which primitives of single formalisms components works. Any formalism that is part of a family can be solved by any solution engine supporting that family by using a dedicated group of *solver* and *solver helper* interfaces.

6 Exponential Event Formalisms

To show how a new formalism family should be designed, we can consider the Exponential Event Formalisms (EEF) family as a reference, as it is the basis for well known formalisms. As the name suggests, EEF are formalisms in which the state of the model is modified through events that occur after random exponentially distributed times. Moreover, all the events that can occur have the property that they can be enabled or disabled depending on the model state. The model is composed by "containers" that can hold a different number of objects, which moves among the various modeling primitives available in the formalism that belong to the EEF family. The first behavior defined in a FDL is called 'InitEvents' and it must check which elements are currently enabled ("active") in a given model state. Each solution engine exploits a behavior called 'Schedule' that is used by 'InitEvents'. Each implementation of 'Schedule' behavior must specify what will be executed whenever an event occurs using a program routing written in a suitable programming language (Java in the current implementation). The portion of code that is executed reflects the evolution of the model by updating the state of the elements. All languages that want to automatically interact together must define an 'isActive' behavior for all elements that represent events. The languages also must expose a 'fire' behavior that is called whenever the event occurs to determine the how the state of the model will change. All elements that represent containers must implement either 'push' or 'pull' behaviors to move objects among them.

Finally, the performance indexes are collected by a another set of behaviors, which are called in turn by the solving engines to compute and update the measurements. To simplify the presentation, we will not enter in detail of these behaviors: interested readers can refer to [20] for further details.

To be more formal, with reference to the framework defined in this paper, EEF

encompasses all the formalisms that can be described by a sub-set of the HSML $(\mathcal{S}, E, \Lambda, \Psi)$, where $\mathcal{S} = \{m_i\}$ contains its (finite and discrete) state space, E contains all the events that can modify the state, and occur at an exponential (and possible state dependent rate) $\Lambda : S \times E \to \mathbb{R}_0^+$. When an event occurs, the state is probabilistically changed according to a distribution $\Psi : S \times E \times S \to [0, 1]$. The state space $\mathcal{S}$ is computed as the set of all the possible combinations of values the properties of all the elements of the model may assume. The 'InitEvents' behavior, checks which events $e \in E$ are enabled in a state $m \in \mathcal{S}$ calling the 'isActive' behavior of the modeling primitives that can cause it (for example, transitions in SPN, or queues in QN). If the event can occur, it is scheduled using the 'Schedule' behavior, passing the rate at which the event occurs computed with $\lambda(m, e)$, and a piece of code that computes the possible set of next states $Next(m) = \{m' : \Psi(m, e, m') > 0\}$. Note that the EEFs can be easily mapped to a CTMC, with state space $\mathcal{S}$, in which the transition rate q_{ij} from state i to state j (with $i \neq j$) is defined as:

$$q_{ij} = \sum_{e \in E} \lambda(m_i, e)\Psi(m_i, e, m_j). \tag{15}$$

6.1 Stochastic Petri Nets

A SPN formalism defines four type of elements: places, transitions, standard arcs and inhibitor arcs. Places have a 'Tokens' property that counts the tokens inside a place. The semantic of the formalism implements some additional behaviors. Specifically the 'Place' `<elementType>` implements the 'ElementOccupancy' behavioral interface to check a place. The 'getOccupancy' and 'setOccupancy' behaviors are wrappers to implement the 'ElementOccupancy' interface, which uses the 'Tokens' property. The second element type is 'Transition', which implements 'Active' and 'FireableEvents' interfaces, and uses 'PushPull' behaviors. Other two behaviors play a crucial role in the model definition: 'isActive' implements 'Active' interface, while 'fire' implements 'FireableEvents' (and the firing rule of SPN transitions) by activating behaviors of the elements that are connected as inputs and outputs of a transition, which in turn are compliant with the 'PushPull' and 'CheckCapacity' interfaces. Only arcs and inhibitor arcs can be connected to a transition. The purpose of 'isActive' behavior is to check the enabling of the transition in the model. Following 'Place and 'Transition' elements, an 'Arc' element implements Edge', 'Active' and 'PushPull' behaviors, and uses 'ElementOccupancy' (to check a Place). The 'isActive' behavior evaluates if the enabling rule of SPN is satisfied by the input place of the arc, as required by the 'isActive' implementation of the transition, while 'Push' and 'Pull' (that implement 'PushPull') enact token consumption and production in consequence of the activation of the 'fire' behavior of the transition. The fourth and last element type is 'InhibitorArc', which is similar to Arc, but checks if a place has less than a given quantity of tokens. The `InitEvents` behavior checks which transitions are enabled and schedules consequently the firing of the eligible ones at the corresponding rate according to Algorithm 1. Note that the rate at which the event occurs corresponds to the firing rate of the correspond-

ing transition, and that the piece of code that it is executed calls the firing of the transition.

As an example of how behaviors are implemented, Algorithms 2 and 3 denote respectively the 'IsActive' and 'Fire'. In the former, a transition looks both for incoming arcs and inhibitor arcs, which in turn implement the 'IsActive' behavior by verifying that the marking of the incoming place is respectively less, or greater or equal to their weight; in the latter, each time a transition fires, it updates the marking. Finally, the performance indices are computed by defining a state reward for each place (that is, its mean number of tokens), and an impulse reward for each transition (its throughput).

Algorithm 1 `InitEvents`

```
1: for all T ∈ Transition do
2:   if T.IsActive() then
3:     solver.Schedule(T.rate,"T.Fire()");
4:   end if
5: end for
```

Algorithm 2 `IsActive`

```
1: for all a ∈ Arc ∪ InhibitorArc where a.to = this do
2:   if NOT a.IsActive() then
3:     return false;
4:   end if
5: end for
6: return true;
```

Algorithm 3 `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
```

6.2 Queueing Networks

Queueing Networks (QN) is a formalism used to analyze a system where a number of servers are connected to serve customers, waiting in a queue. Queuing networks are composed by just two element types: queuing stations and interconnection arcs. In this case queuing nodes are characterized by the 'Length' property that counts the number of jobs currently in execution, and by the event that fires when the job currently in service finishes. The 'InitEvents algorithm is shown in 4. The 'isActive' behavior is implemented by checking if the length of the corresponding queue is greater than 0. The 'Queue' element consists of the 'getOccupancy' and

'setOccupancy' that are wrappers to implement the 'ElementOccupancy' interface and work on the 'Length' property. The 'Arc' subElement implements 'Push' and 'Pull'. Specifically, it uses the 'Push' behavior of the connecting arc to address the customer to the next station and in turn calls the 'AddOccupancy' behavior of the 'Queue' at the other end of the 'arc'. The 'AddOccupancy(c)' behavior adds c customers to the length of each queue.

Algorithm 4 `InitEvents`

```
1: for all q ∈ Queue do
2:   if q.IsActive() then
3:     for all a ∈ Arc where a.from = q do
4:       solver.Schedule(a.prob · q.rate,"q.Fire(a)");
5:     end for
6:   end if
7: end for
```

The event is scheduled at rate that is computed as $a.prob \cdot q.rate$: the first term represents the routing probability and it is used to allow a probabilistic choice among several destination. The notation $a.prob$ states that it is collected from a static property of the arc. The latter, $q.rate$, is instead a property of the queue and corresponds to the exponential service rate of the considered station. As for SPN, the code containing the definition of what happens when the scheduled event is executed, relies on the execution of a 'Fire' behavior. However, in this case, the 'Fire' behavior of a 'Queue' includes a parameter defining the destination of the service. The 'Fire' behavior is described by Algorithm 5.

Algorithm 5 `Fire`(a)

```
1: a.Push();
2: length = length - 1;
```

Finally, the following performance metrics are computed: the throughput of the queues as impulse rewards, and the average number of jobs in a queue as state rewards.

Note that, since the both SPN and QN use the same behaviors to check if nodes are enabled and move objects among the primitives, the two formalisms can easily interact and be interconnected together to analyze multiformalism models.

7 A Hybrid Formalisms Family for SIMTHESys

EEFs can be extended to support HS: this new formalism family will be addressed as *Hybrid Formalism Family* (HFF). In particular, it must account for the fact that states are composed of a discrete and a continuous part. The latter is characterized by a continuous domain that can change depending on the former. It must also consider the fact that the rate at which events occur can depend on the continuous part of the state, and whenever the system changes mode (its discrete state), all the continuous variables could be altered in a deterministic or stochastic way.

HFF still uses the 'InitEvents' behavior in order to take account of the events that are enabled (or can be enabled due to the evolution of the continuous variables) in a state. In addition, HFF requires supported formalisms to implement two new behaviors: 'getFluidDomain' returns the continuous variables that must be considered in the corresponding discrete state. If the current state corresponds to mode m_i, it returns the value of variable D_i in the HSML state definition. Behavior 'getFluidEvolution' returns the function describing how the continuous variable evolve with time in the considered mode. If the current state is mode m_i, it returns a function encoding ϕ_i. The parameters used by 'Schedule' behavior must take into account Λ and Ψ. In particular, since Λ for HSML can depend on the continuous state, the 'Schedule' behavior implemented by the solvers of the HFF models must accept a function instead of a constant to specify the firing rate of the events. The code being executed, must implement the change in the mode of function Ψ. The change to the continuous variables must instead be passed as an extra parameter to the 'Schedule' behavior. It should not only update the discrete properties of the model to determine the next mode, but it must also adjust the values of the continuous variables of the current mode to account for the new variables that will be available in the destination mode.

7.1 Hybrid Petri Nets

Hybrid Petri Nets (HPN) adds to conventional SPN new elements to model HS. Several slightly different HPN dialects have been defined: in this work we will focus on the primitives shown in Figure 2. In particular, HPN are characterized by a set of *discrete places* P_D and a set of *discrete transitions* T_D. Each transition $\tau \in T_D$ is characterized by a possibly state dependent exponential firing rate $\mu_\tau(\sigma)$. HPN also contains a set of *fluid places* P_F (usually represented as double circles) and a set of *fluid transitions* T_F (drawn as double boxes). HPN contains several different type of arcs: *discrete* and *inhibitor arcs* (collected respectively in sets A_D and A_H) have the conventional meaning defined for SPN. *Test arcs* A_T, usually represented as a line with an arrow on both sides, consist of a primitive enabling transitions when the connected place has more tokens than the weight of the arc. For standard SPN, they can be considered as a short-hand notation for two arcs, with the same weight, connecting one transition to one place in opposite direction. For HPN however they have a special meaning and must be explicitly included in the formalism. *Fluid arcs* A_F, usually represented as pipes, allow the flow of fluid between fluid places and fluid transitions. A fluid arc $a \in A_F$ is characterized by a fluid weight $w_a \in \mathbb{R}$.

Fluid places are characterized by a level property, accounting for the continuous value contained in the place. This level is usually addressed as *fluid* as opposed to tokens contained in conventional discrete places. Each fluid place has an implicit lower boundary at level 0; in order to allow for model solution, the level of a fluid place $p \in P_F$ is usually upper bounded by a specified level B_p. An enabled fluid transition $\tau \in T_F$ transfers fluid at rate $r_\tau(\sigma)$ among the fluid places to which it is connected. In particular, if arc $a \in A_F$ starts from fluid place $p \in P_F$ and ends in an enabled transition $\tau \in T_F$, it removes from place a fluid at rate $-w_a \cdot r_\tau(\sigma)$ units

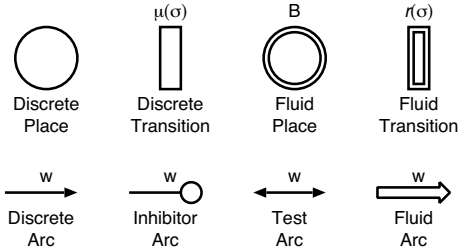

Fig. 2. HPN modeling primitives

per second. If arc a' starts from τ and ends in p, then the enabled transition pumps $w_{a'} \cdot r_\tau(\sigma)$ fluid units per second int place p. Fluid transitions can be enabled or disabled by discrete places using inhibitor or test arcs. If the input place has not enough fluid to satisfy the outgoing request, or if an output place is full, the rate of the corresponding transition is reduced to allow flow conservation. This procedure, called *rate adaption* [1] is quite complex, and its description is outside the scope of this work. In this paper, it is simply assumed that the fluid rate $r_\tau(\sigma)$ accounts for rate adaption in its state dependency.

The evolution and the semantic of the discrete part of HPN is identical to the one of SPN: for this reason it can be implemented exactly as outlined in Section 6.1. In the corresponding HSML, the state space $\mathcal{S}$ is defined as the set of the possible different configurations that the discrete part of the model (i.e. the marking of the discrete places) may assume, and it can be generated in SIMTHESys with the same technique used to visit the state space of SPN models. The `InitEvents` behavior checks which transitions are enabled. The rate at which events can occur, $\lambda(\sigma, e)$ depends on the level of the fluid places. Moreover, if event $e_\tau \in E$ corresponds to the firing of a transition $\tau \in T_D$, then $\lambda(\sigma, e_\tau) = \mu_\tau(\sigma)$ will be defined as the (possibly fluid dependent) rate of τ. The 'Schedule' behavior is then called for each active transition $\tau \in T_D$, using as rate the function $\lambda(\sigma, e_\tau)$. The code associated to the event updates the mode of the HSML by changing the token in the discrete places as for conventional SPN. Fluid variables remain unchanged, so 'Schedule' behavior receives parameter $\Psi(e_\tau, \sigma) = \mathrm{DET}(\sigma)$.

In HPN the continuous variables are identical in all the states and correspond to the level of the fluid places of the model. For this reason, all discrete states m_i have the same *continuous domain* D_i:

$$D_i = \bigtimes_{p \in P_F} [0, B_p] \tag{16}$$

This implies that the behavior 'getFluidDomain' always returns D_i. The evolution of the continuous variables depends on the enabled fluid transitions, which in turns depend on the marking of places (both discrete and continuous). Behavior 'getFluidEvolution' returns the function that describes the fluid evolution, considering input and output fluid arcs for each continuous place. In particular, let us call $\mathcal{E}(\sigma) \in T_F$ the set of fluid transitions enabled in state σ. Let us call $\mathbf{x} \in D_i$ the set

of the continuous values contained all the fluid places of the model when behavior 'getFluidEvolution' is called. Function ϕ can be computed as the solution of the following differential equation:

$$\begin{cases} \dfrac{d\phi(\mathbf{x},\tau)}{dt} = \displaystyle\sum_{\tau\in\mathcal{E}(\sigma)} r_\tau(\sigma)\cdot\left(W_\tau^{in} - W_\tau^{out}\right) \\ \phi(\mathbf{x},0) = \mathbf{x} \end{cases} \tag{17}$$

where $W_\tau^{in} = |w_{a:\tau\to p}|$ is a vector with as many components as fluid places, and for each $p \in P_F$, term $w_{a:\tau\to p}$ denotes the weight of the fluid arc starting from τ and directed to p or zero if such arc does not exist. In a similar way, W_τ^{out} accounts for the arcs ending on fluid transition τ.

7.2 Queuing Networks with general service distribution

QN with general service distributions are solved with the supplementary variable approach [11]. In particular, each non-exponential queue has associated a fluid variable that represents the 'clock' of the corresponding service time distribution. Behavior 'getFluidDomain' returns a domain with as many components as the non-exponential queues that are currently working (i.e. not idle) in the state. Behavior 'getFluidEvolution' instead returns a linear increase with rate 1 for each continuous variable corresponding to the clock of a non-idle queue. The 'Schedule' behavior is called for each non-idle queue, with firing rate argument Λ set to the hazard rate of the corresponding non-exponential service time distribution, and parameter Ψ set to a function that puts to zero the supplementary variable corresponding to the queue that has finished service, and leaves the other clocks unchanged.

8 Application

We now extend the case study presented in Section 2 to consider a more complex workload model. We use a standard closed QN model to describe the IT components of the data-center and a HPN to model the air conditioning system. The model of the considered system is shown in Figure 3. The workload of the data-center is characterized by N users that submit requests to the system. Each users has a think time Z which is spent in the infinite server station *Terminals* of the QN part of the model. Jobs requires both compute resources at queue *Compute* characterized by a given average service time S_C, and storage resources at queue *Storage* with service time S_S. Requests always ends after a visit to the compute node, with probability p_{job}, otherwise they perform another storage-compute cycle. The data-center is composed by c_C compute servers, and c_S storage servers, which are modeled with c-servers queues.

The temperature is now modeled by fluid place *Temperature*, and it rises considering the heat produced by the workloads of the compute (fluid transition $Heat_C$) and the storage ($Heat_S$) servers. The cooling effects of fan and AC are modeled

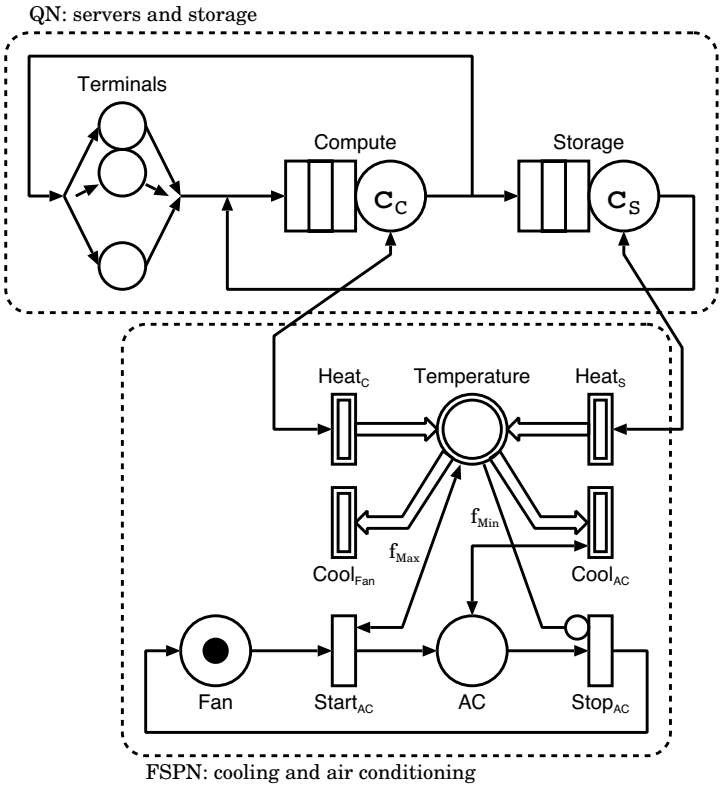

Fig. 3. A small datacenter QN-HPN model.

respectively by fluid transitions $Cool_{Fan}$ and $Cool_{AC}$. When place *Fan* is marked, the data-center is cooled down using just fans. As soon as the temperature becomes larger than f_{Max}, the token moves to place AC thanks to the exponential transition $Start_{AC}$ to denote the activation of the air conditioning system. The firing of the exponential transition $Stop_{AC}$ models the return to fan-based cooling whenever the temperature drops below threshold f_{Min}. To simplify the presentation, the case in which the AC can fail starting has not been taken in account. Instead, the domain of the fluid places is set to $[t_{Room}, t_{Critical}]$, and the failure event is evaluated as the probability of reaching $f_{Critical}$ with the considered configuration. Note that a description of the considered system directly in HSML would have required $N(N+1)$ different modes.

Power consumption is computed using a simple linear model, supposing that the power consumption is linearly proportional to the utilization of the system:

$$\begin{aligned} P = {} & c_C \cdot P_{C\text{-}idle} + (P_{C\text{-}Max} - P_{C\text{-}idle})U_C + \\ & + c_S \cdot P_{S\text{-}idle} + (P_{S\text{-}Max} - P_{S\text{-}idle})U_S + \\ & + P_{Fan} + Pr\{\#AC = 1\} \cdot P_{AC} \end{aligned} \tag{18}$$

where $P_{C\text{-}idle}$ and $P_{C\text{-}Max}$ are respectively the idle and maximum power of each compute server, $P_{S\text{-}idle}$ and $P_{S\text{-}Max}$ the same values for the storage servers, $0 \leq U_C \leq c_C$ and $0 \leq U_S \leq c_S$ are the utilizations of the compute and storage servers, P_{Fan}

and P_{AC} are the power consumptions of the fan and of the air conditioning systems, and $Pr\{\#AC = 1\}$ is the probability that place AC is marked. Parameters used in the model are summarized in Table 1. The components of the tuple $(\mathcal{S}, \Phi, E, \Lambda, \Psi)$ have been manually generated following the procedure described in Section 5. The solution has been computed discretizing the temperature range in 61 equally spaced steps: in this way the HSML model has been transformed into an ordinary CTMC and solved using conventional techniques [5]. The solution of the model required few seconds on a 2011 *MacBook Air Intel Core-i5* PC with 4GB of RAM.

Table 1
Model parameters

Workload	$N = 1 \dots 25$		
Compute Servers	$c_C = 3 \dots 5$	Storage Servers	$c_S = 2$
Compute Av. Serv.	$S_C = 2$ s.	Storage Av. Serv.	$S_S = 0.5$ s.
Think time	$Z = 1$ min.	Prob. end of a job	$p_{job} = 0.1$
Compute Heating	$Heat_C = 0.3\ \frac{\text{deg.}}{\text{job·s.}}$	Storage Heating	$Heat_C = 0.1\ \frac{\text{deg.}}{\text{job·s.}}$
Fan Cooling	$Cool_{Fan} = 0.5\ \frac{\text{deg.}}{\text{s.}}$	AC Cooling	$Cool_{AC} = 1.5 \dots 2\ \frac{\text{deg.}}{\text{s.}}$
Start AC rate	$Start_{AC} = 10\ \text{s.}^{-1}$	Stop AC rate	$Stop_{AC} = 10\ \text{s.}^{-1}$
Start AC temp.	$f_{Max} = 25^o$	Stop AC temp.	$f_{Min} = 20^o$
Room temp.	$f_{Room} = 18^o$	Critical temp.	$f_{Critical} = 30^o$
Compute idle power	$P_{C\text{-}idle} = 70$ Watt	Storage idle power	$P_{S\text{-}idle} = 70$ Watt
Compute max. power	$P_{C\text{-}Max} = 140$ Watt	Storage max. power	$P_{S\text{-}Max} = 80$ Watt
Fan power	$P_{Fan} = 100$ Watt	AC power	$P_{AC} = 1 \dots 1.35$ KW.

The model has been evaluated with an increasing workload $N = 1 \dots 25$, for different numbers of compute servers $c_C = 3 \dots 5$. The configuration with $c_C = 5$ has been evaluated with two configurations exploiting two different AC units: one capable of cooling 1.5 deg. / s. and requiring a power of $P_{AC} = 1000$ Watt (denoted in Figures 4-8 with $c = 5$), and a second more powerful system capable of cooling 2 deg. / s., but requiring $P_{AC} = 1350$ Watt (denoted with $c = 5+$).

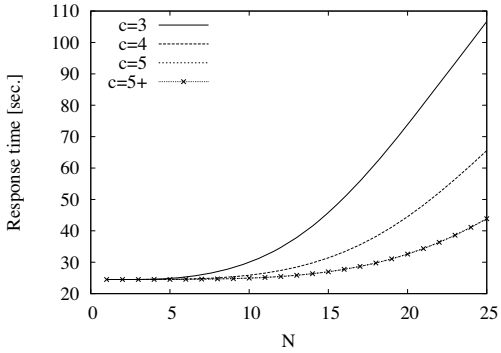

Fig. 4. Response time of the system as function of the workload for a different number of compute servers.

Figure 4 shows the average system response time that, as expected, decreases when increasing the number of compute servers. It is interesting to note that the

[5] Please note that a general solution technique for HSML models is an important topic which outside the scope of this work. Here, an ad-hoc solution has been implemented to show the feasibility of the multiformalism approach

type of AC device used has no influence on this part of the model (that is the lines for $c = 5$ and $c = 5+$ are superposed), since there is no feedback from the HPN sub-model to the QN component.

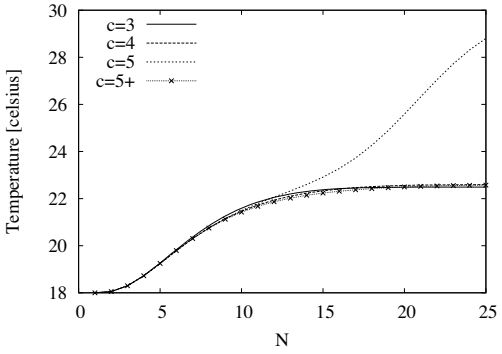

Fig. 5. Average temperature of the system as function of the workload for a different number of compute servers.

The average room temperature is shown in Figure 5. As expected, the average temperature increases with workload, and tends to reach the asymptote $(t_{Max} + t_{Min})/2 = 22.5$ due to the control that activates and turns off the air conditioning. It is interesting to note that the average temperature depends just on the workload, and not on the number of servers used in the configuration.

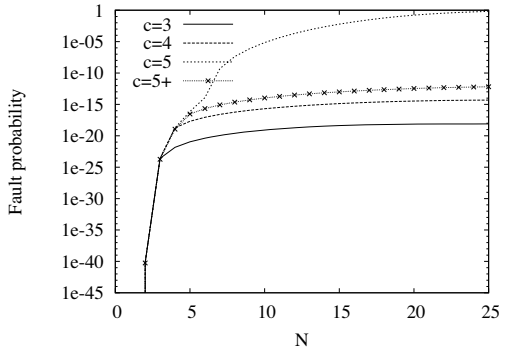

Fig. 6. Fault probability of the system as function of the workload for a different number of compute servers.

Figure 5 also shows that the temperature for the case $c = 5$ (five compute servers, lower power AC system) tends to rise more than in the other configurations. Figure 6 shows the probability of reaching the critical temperature for the various configurations. It is clear that while for most of the cases this probability is negligible, the configuration $c = 5$ has a large value, meaning that the AC unit that can be used with $c = 3$ or $c = 4$ is not enough to cool down the room with $c = 5$, and a larger (but more power demanding) model is required, as shown by the curve labeled as $c = 5+$.

Figure 7 shows the average power consumption of the considered configurations. As expected, the power consumption increases with the number of servers, but com-

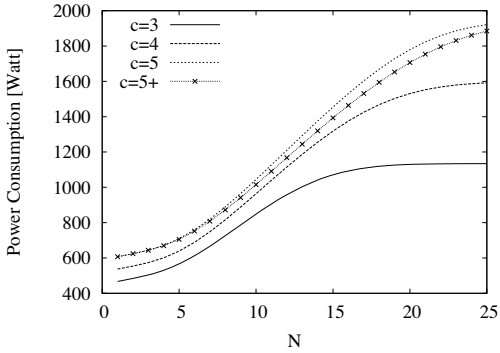

Fig. 7. Average power consumption of the system as function of workload for a different the number of compute servers.

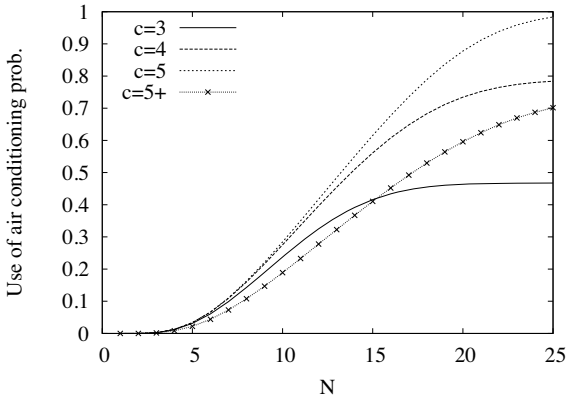

Fig. 8. Probability of switching on the air conditioning system as function of the workload for a different number of compute servers.

paring curves $c = 5$ and $c = 5+$ for workloads $N \leq 13$ where the configuration with the less powerful AC unit still provides a reliable operation, it can be seen that the use of a more powerful cooling system can reduce the overall energy requirements. This is illustrated in Figure 8 where the probability that the AC is activated is shown: the more powerful air conditioner can be activated for a shorter time period when using medium workloads, leading to a lower average power consumption even if the cooling equipment requires a larger amount of energy.

9 Conclusions and Future Work

In this paper a general solution has been presented to support the definition of custom modeling languages for HS. Besides this, the main points behind the design of such formalisms have been pointed out and examined, in order to help interested readers to autonomously produce own solutions, independently from the SIMTHESys framework. Many developments are planned after this first work on the topic, going from the use of SIMTHESys to study and define specific domain oriented formalisms to the enrichment of the HFF with additional features and

support for custom metrics.

Acknowledgment

The results of this work have been [partially] funded by EUBra-BIGSEA (690116), a Research and Innovation Action (RIA) funded by the European Commission under the Cooperation Programme, Horizon 2020 and the Ministrio de Cincia, Tecnologia e Inovao (MCTI), RNP/Brazil (grant GA/0000000650/04).

References

[1] Alla, H. and R. David, *Continuous and hybrid petri nets*, Journal of Circuits, Systems and Computers **08** (1998), pp. 159–188.

[2] Antsaklis, P. J. and X. D. Koutsoukos, "Hybrid Systems: Review and Recent Progress," John Wiley & Sons, Inc., 2005 pp. 273–298.

[3] Barbierato, E., A. Bobbio, M. Gribaudo and M. Iacono, *Multiformalism to support software rejuvenation modeling.*, in: *ISSRE Workshops* (2012), pp. 271–276.

[4] Barbierato, E., M. Gribaudo and M. Iacono, *Defining Formalisms for Performance Evaluation With SIMTHESys*, Electr. Notes Theor. Comput. Sci. **275** (2011), pp. 37–51.

[5] Barbierato, E., M. Gribaudo and 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.

[6] Barbierato, E., M. Gribaudo and M. Iacono, *A performance modeling language for big data architectures.*, in: W. Rekdalsbakken, R. T. Bye and H. Zhang, editors, *ECMS* (2013), pp. 511–517.

[7] Barbierato, E., M. Gribaudo and M. Iacono, *Performance evaluation of nosql big-data applications using multi-formalism models*, Future Generation Computer Systems **37** (2014), pp. 345 – 353.

[8] Barbierato, E., M. Gribaudo, M. Iacono and S. Marrone, *Performability modeling of exceptions-aware systems in multiformalism tools*, in: *ASMTA*, 2011, pp. 257–272.

[9] Barbierato, E., M. Iacono and S. Marrone, *PerfBPEL: A graph-based approach for the performance analysis of BPEL SOA applications.*, in: *VALUETOOLS* (2012), pp. 64–73.

[10] Barbierato, E., G.-L. D. Rossi, M. Gribaudo, M. Iacono and A. Marin, *Exploiting product forms solution techniques in multiformalism modeling*, Electronic Notes in Theoretical Computer Science **296** (2013), pp. 61 – 77.

[11] Cox, D. R., *The analysis of non-markovian stochastic processes by the inclusion of supplementary variables*, Mathematical Proceedings of the Cambridge Philosophical Society **51** (1955), pp. 433–441.

[12] Davis, M., "Markov Models & Optimization," Chapman & Hall/CRC Monographs on Statistics & Applied Probability, Taylor & Francis, 1993.

[13] Ghasemieh, H., A. Remke, B. R. Haverkort and M. Gribaudo, *Region-based analysis of hybrid petri nets with a single general one-shot transition*, in: M. Jurdzinski and D. Nickovic, editors, *Formal Modeling and Analysis of Timed Systems - 10th International Conference, FORMATS 2012, London, UK, September 18-20, 2012. Proceedings*, Lecture Notes in Computer Science **7595** (2012), pp. 139–154.

[14] Gribaudo, M. and A. Remke, *Hybrid petri nets with general one-shot transitions for dependability evaluation of fluid critical infrastructures*, in: *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on*, 2010, pp. 84–93.

[15] Gribaudo, M. and A. Remke, *Hybrid petri nets with general one-shot transitions for dependability evaluation of fluid critical infrastructures*, in: *12th IEEE High Assurance Systems Engineering Symposium, HASE 2010, San Jose, CA, USA, November 3-4, 2010* (2010), pp. 84–93.

[16] Henzinger, T. A., *The theory of hybrid automata*, in: *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96 (1996), pp. 278–.

[17] Henzinger, T. A., P.-H. Ho and H. Wong-toi, *Hytech: A model checker for hybrid systems*, Software Tools for Technology Transfer **1** (1997), pp. 460–463.

[18] Horton, G., V. G. Kulkarni, D. M. Nicol and K. S. Trivedi, *Fluid stochastic petri nets: Theory, applications, and solution techniques*, European Journal of Operational Research **105** (1998), pp. 184–201.

[19] Iacono, M., E. Barbierato and M. Gribaudo, *The SIMTHESys multiformalism modeling framework*, Computers and Mathematics with Applications (2012), pp. 3828–3839.

[20] Iacono, M. and M. Gribaudo, *Element based semantics in multi formalism performance models*, in: *MASCOTS* (2010), pp. 413–416.

[21] Kwiatkowska, M., G. Norman and D. Parker, *Probabilistic symbolic model checking with prism: a hybrid approach*, International Journal on Software Tools for Technology Transfer **6** (2004), pp. 128–142.

[22] Larsen, K. G., P. Pettersson and W. Yi, *Diagnostic Model-Checking for Real-Time Systems*, in: *Proc. of Workshop on Verification and Control of Hybrid Systems III*, 1066 (1995), pp. 575–586.

[23] Lee, E. and H. Zheng, *Operational semantics of hybrid systems*, in: M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science **3414**, Springer Berlin Heidelberg, 2005 pp. 25–53.

[24] Liu, J., X. Liu and E. A. Lee, *Modeling distributed hybrid systems in ptolemy ii*, Proceedings of the American Control Conference (2001), pp. 4984–4985.

[25] Lygeros, J., *Lecture notes on hybrid systems*, Technical report, ETH Zurich (2004).
URL http://robotics.eecs.berkeley.edu/~sastry/ee291e/lygeros.pdf

[26] Moscato, F., F. Flammini, G. D. Lorenzo, V. Vittorini, S. Marrone and M. Iacono, *The software architecture of the OsMoSys multisolution framework*, in: *ValueTools '07: Proceedings of the 2nd international conference on Performance evaluation methodologies and tools*, 2007, pp. 1–10.

[27] Zhu, F. and P. J. Antsaklis, *Optimal control of hybrid switched systems: A brief survey*, Discrete Event Dynamic Systems **25** (2015), pp. 345–364.