Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 275 (2011) 37–51

www.elsevier.com/locate/entcs

Defining Formalisms for Performance Evaluation With SIMTHESys

Enrico Barbierato[1]

DI, Università degli Studi di Torino
corso Svizzera, 185, 10129 Torino (Italy)

Marco Gribaudo[2]

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

Mauro Iacono[3]

DEM, Seconda Università degli Studi di Napoli
Belvedere Reale di San Leucio, 81100 Caserta (Italy)

Abstract

Tools for the analysis and modeling of complex systems must be able to support the extensibility of formalisms, reusability of models and customization of formalism compositions. From this perspective, SIMTHESys (Structured Infrastructure for Multiformalism modeling and Testing of Heterogeneous formalisms and Extensions for SYStems) is a new approach to the specification of performability oriented formalisms and the evaluation of models. Its originality emerges from the explicit definition of both syntax and evolution semantics of the considered formalism elements. The solution of models is made possible by using a set of non-specialized solving engines used to generate automatically formalism-specific reusable solvers. This paper explains how formalisms can be created in SIMTHESys by showing how three widely known modeling languages are successfully implemented.

Keywords: Performance analysis, tools, specification, performability-oriented formalism

1 Introduction

Some of the challenges that current systems can present to the scientific and professional community in terms of performance, reliability and service level agreement

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

doi:10.1016/j.entcs.2011.09.004

can be addressed by analyzing models based on many different performability formalisms.

Modeling is a complex task because of the heterogeneity of the system needed to be studied. The customization of some of the formalisms for a specific task can simplify the modeling process at a cost of creating a tool to solve a model based on an extended formalism. The idea of creating a tool for each formalism extension seems not to be feasible. Notwithstanding the huge number of known modeling formalisms, they share some basic fundamental ideas that suggests a classification in families. These can be re-grouped using a common solver that can be specialized for the different cases. The SIMTHESys approach, presented in [14] [4] moves onward from such premise to propose a new solution to the problem. SIMTHESys offers a compositional, metamodeling based framework to describe and extend formalisms [5] .

This paper aims to demonstrate how to design formalisms belonging to the Exponential Event Formalisms (EEF) family. Three cases are presented, showing how three common performance evaluation frameworks, namely Stochastic Petri Nets (SPN) [15], Tandem Finite Capacity Queueing Networks (TFCQN) [13] and Gordon and Newell Queueing Networks (GNQN) [10] can be defined using the SIMTHESys approach. Two solution components architectures have been designed for the EEF family. Respectively, they perform stochastic simulation and numerical solution. Additionally, a simple model is analyzed to show the possibilities offered by the solution components.

The paper is organized as follows. After a brief review of the SIMTHESys approach to multiformalism modeling in Section 2, in Section 3 the exponential transition based family of formalisms is introduced. Section 4 describes the implementations of SPN, TFCQN and GNQN. Section 5 presents the solution engines for the mentioned family. Conclusions and future work are described in section 6.

2 SIMTHESys overview and related works

The early experiences of Sharpe [17] [19], SMART [2] and the DEDS toolbox [1] to Mobius [3] [6] [18] [4] [5] and OsMoSys [8] [21] [16] [20] [9] (that are the closest references for this research) proved the value of a formal (multiformalism/multisolution [6]) approach to modeling and evaluation of systems.
SIMTHESys is a new framework for the definition and analysis of performance oriented formalisms. It is based on the explicit definition of both syntax and semantics of all atomic components of a formalism and on a set of non-specialized solving engines, that are used to generate automatically and transparently (multi)formalism-specific reusable solvers.

The main advantage of the SIMTHESys approach is that it allows rapid pro-

[4] Also visit the SIMTHESys web site at www.dem.unina2.it/simthesys

[5] The approach also supports multiformalism models, obtained by connecting together submodels written in different formalisms by exploiting their dynamics, but this feature is out of the scope of this paper

[6] Multiformalism refers to the possibility of using different formal languages to specify different portions of a model. Multisolution identifies the possibility of integrating different existing performance evaluation tools to compute the performance indices of a complex model.

totyping of new formalisms and solution techniques, with native multiformalism support.

With respect to [14], which aims to define how the SIMTHESys methodology can be used to support interaction between multiformalisms models, this paper focuses on the process of creating a new formalism by presenting the steps required to implement three known formalisms.

A SIMTHESys *formalism* is a formal description language. It is defined in terms of its *elements*, that are its atomic components. Each element is characterized by a set of attributes called *properties* and a set of descriptions of its dynamics called *behaviors*. Properties can be constant characteristics of the element, state information (useful for the solution of the model) or results (obtained by the solution process). Behaviors describe the semantics of the element (the effects of their presence in the model on the other elements, e.g. its execution policy, if applicable). A behavior is an algorithm implemented in a high-level programming language (that currently follows the syntax of Java) [7]. Every formalism has an additional *container element* that represents a (sub)model written in that formalism and whose properties include global characteristics of a model. Formalisms are described by the Formalism Description Language (FDL).

A *model* is the description of a system being evaluated, written according to a formalism. Models can be hierarchically composed, to separate reusable (sub)models of its subsystems, even if written in different formalisms [8]. Models are described by the Model Description Language (MDL) documents.

The framework is composed of three components: the SIMTHESys Metamodeling Structure (SimMS), the *Interfaces* (IFs) and the *Solving Architecture* (SA). The IFs fill the gap between the high abstraction level of formalisms and the generic applicability of the solvers. IFs constitute the foundation on which elements behaviors are built, supplying general reusable functions and access to solving engines. IFs can be seen as a middle layer supporting the interactions between SimMS and SA. They decouple the problem of solving a model in the best way available from the problem of describing it with the best suitable formalism. As a result, IFs are different from the Abstract Functional Interface (AFI) [7] considered by Mobius.

This software architecture is designed to represent efficiently the development of extensions of formalisms, the evaluation of formalism variants, and to support multiformalism [9].

The SA offers an extensible set of *Solving Engines*, suitable for the evaluation of performance indices or other significant properties of a model. Solving engines are generic solvers, that are meant to be used or composed to obtain a solution component for a certain formalism defined in a FDL document. Solution engines offer proper interfaces to the IFs and will be discussed in more detail later in the

[7] A part of the project's future work in this area will be committed to the definition of a high-level programming language to allow the developer to implement new behaviors. Such a language will then be translated in Java or C++ depending on the numerical precision requested from the solving engine.

[8] A model with heterogeneous submodels is written in a *composition formalism*, that is a formalism capable of semantically connecting concepts belonging to different formalisms.

[9] Although the focus of this paper is not on multiformalism and multisolution, many of the design choices aim at simplifying the interoperability between different formalisms.

paper.

The SIMTHESys framework has been developed to be integrated with the DrawNET tool [11][12] and the OsMoSys framework. DrawNET is a formalism definition environment that provides data structures and the automatic creation of graphical user interfaces for the rapid development of performance evaluation tools. OsMoSys is a framework that supports the definition of formalisms and models to allow the use of multiformalism and multisolution techniques.

3 Exponential Event Formalisms

The family of formalisms presented in this paper are labeled as *Exponential Event Formalisms* (EEFs). In these formalisms, primitives represent entities capable of generating *events*. Such events (that can be used to represent the firing of a PN transition, or the end of the service in a queue) are characterized by the fact that they occur after an exponentially distributed time.

Each formalism belonging to this class should define a behavior called `InitEvents`. Its purpose is to determine the events that are enabled in a given state and the constant rate λ that characterizes the exponential distribution. If more than one event is enabled at the same time, a *race policy* is used to solve the conflict: the event with the smallest firing time is selected for firing.

The solution engines expose a behavior `Schedule` that is used by the implementations of the `InitEvents` behavior to define what must be executed whenever an event occurs. The code associated with the event updates the state of the elements to reflect the evolution of the model. The scheduled behavior updates the state of the affected elements of the model. For example a transition can move tokens among the places to which is connected by using specific arcs behaviors.

3.1 Performance Indices

EEFs may be characterized by performance indices calculated by using *State Rewards* and *Impulse Rewards*, similarly to those defined in [6].

State Rewards return the mean value of a function of the state of the model. They are used to compute the mean length of a queue, or the mean number of tokens in a PN place. Each EEF define a set of state rewards by implementing three behaviors. Firstly, `CountStateRewards` returns the number of state rewards that a model is able to expose. Secondly, `ComputeStateRewards` computes the value of all the rewards associated with the model in a given state. Finally `SetStateRewards` allows the solution engine to return the computed indices to the model.

Impulse rewards are used to compute measures related to the firings of events, such as the throughput of queues or PN transitions. Each impulse reward is characterized by a unique *reward name*, and is defined by implementing two behaviors and by passing appropriate parameters to the previously defined solution engine `Schedule`. A formalism should list the reward names of all the impulse rewards by implementing the behavior `ListImpulseRewards`. Every time an event is scheduled, the formalisms pass to the `Schedule` behavior the name of a reward that should be

updated, and the magnitude of the impulse. As for the state rewards, the behavior `SetImpulseRewards` allows the solution engine returning the computed reward to the model.

4 Case study: formalisms implementations

This section shows how the behaviors defined in Section 3 should be implemented by the TFCQN, GN-QN and SPN formalisms to compute the solution of their corresponding models using a solution engine defined for a EEF.

4.1 Tandem Finite Capacity Queueing Networks

Queueing Networks (QN) is a formalism suitable for the analysis of systems in which a number of servers are connected to serve customers, which wait in a queue. A QN is composed by two kinds of elements: the *queue* and the *arc*. A queueing network can be closed (the same N customers keep being continuously served in the network) or open (some customers join the network according to a given interarrival time distribution and some of them leave).

4.1.1 Formalism analysis

TFCQN is a variant of QN in which every queue has a finite number M of places for waiting customers (Finite Capacity). Only a single arc can leave a queue (Tandem network) [10] . If there is no room for a customer in a queue, the input stations stop serving until a place is available. This occurs as a result of a *blocking mechanism*.

The three most common are: Blocking After Service (BAS), Blocking Before Service (BBS) and Repetitive Service (RS). In the first case, the customer who does not find room in the destination queue is processed anyway and it is blocked right after service completion. In the second case, the source queue is blocked before processing the customer. In this case, either the customer enters the server (BbsSoQueue) or not (BbsSnoQueue, as a result the number of places in the queue is set to M-1 until it is unblocked). In the third case, if the destination queue does not have room left, a customer who finishes the service is reinserted in the same queue to be served again later (RSqueue).

Note that BAS queues is not supported. Even if their service time satisfies the EEF property, in case an unblocking condition is set they are supposed to transfer the waiting customer immediately to the destination queue [11] .

On an abstract level, TFCQN stations are characterized by two *structural* [12] properties: the maximum capacity of the queue and the rate of the exponential distribution corresponding to the service time. The state of each queue is uniquely specified by its length at a given instant of time. The performance indices that are

[10] Note that though probabilistic branching from service nodes allows to represent a more general model, only TFCQN have been considered for the sake of simplicity.

[11] They belong to the ExpAndImmediateEvent family (EAIEF), also available in SIMTHESys.

[12] Structural properties are static information associated with the elements of the formalism that do not change during the model evolution.

commonly computed on TFCQNs are the mean queues length and the throughput of each station.

4.1.2 Formalism implementation

TFCQN is implemented either by defining a generic queue object with an associated property that specifies the kind of block or by defining a different element type for each blocking policy. We chose the second alternative because it simplifies the coding of the behaviors. The next step to consider is to identify the kind of (discrete) event corresponding to the termination of a service provided to a customer.

Table 1 presents the elements, properties and behaviors that can be used to define a TFCQN in SIMTHESys. All the queues have the same attributes and behaviors, but the latter are implemented differently. Properties reflect the description provided in Section 4.1.1.

Note that the 'Modifier' attribute associated with each property represents its role in the element definition. Structural properties have a 'const' modifier, and dynamic information are stored in properties with the 'status' modifier. Finally, performance indices have a 'computed' modifier.

The `InitEvents` behavior is defined in Algorithm 1. The object oriented like dot notation is used to name the behaviors and the properties associated to the elements of a model. The external object **solver** refers to the solution engine, that exposes the method `Schedule` to enable the events. This method has four parameters that identify: i) the rate of the exponential distribution that characterizes the firing time, ii) the piece of code that must be executed when the event occurs, iii) the name of the impulse reward that is associated to the event, and iv) the increment of the reward. The notation q.**id** is used to identify the name of the queue. Since every queue has an associated throughput, the name of the queue is used as the name of the corresponding reward. Due to the fact that throughput counts the number of services in a queue, its corresponding reward value is always 1.

Algorithm 1 `InitEvents`

```
1: for all q ∈ RSQueue ∪ BbbSoQueue ∪ BbsSnoQueue do
2:   if q.IsActive() ∧ q.CanSend() then
3:     solver.Schedule(q.rate, "q.Fire()", q.id, 1);
4:   end if
5: end for
```

The `IsActive` behavior is identical for all the types of queues, and simply checks if the length of the corresponding queue is greater than 0. The `CanSend` behavior is used to check whether a station can start its service or it is blocked because it has reached the full capacity of the destination node. In an RSQueue, the behavior always returns *true* (because the service is always enabled, and it is re-issued if the destination node is full). Regarding the other types of queue, the `CanSend` behavior is specified in Algorithm 2.

The special keyword **this** is used to identify the queue to which the behavior is associated, and the purpose of the **forall** statement is to look for all the possible

Element	Property	Type	Modifier	Behaviors
TFCQN				InitEvents, ComputeStateRewards, CountStateRewards, SetStateRewards, ListImpulseRewards, SetImpulseRewards
Arc	from to	element element	const const	HasSpace, Push
BbsSnoQueue	length meanlength capacity rate throughput	integer float integer float float	status computed const const computed	IsActive, AddOccupancy, Fire, CanSend, CanAccept
BbsSoQueue	length meanlength capacity rate throughput	integer float integer float float	status computed const const computed	IsActive, AddOccupancy, Fire, CanSend, CanAccept
RSQueue	length meanlength capacity rate throughput	integer float integer float float	status computed const const computed	IsActive, AddOccupancy, Fire, CanSend, CanAccept

Table 1
Elements of the TFCQN SIMTHESys definition

destinations of the queue (in the tandem assumption, it is at most one station). The `HasSpace` behavior associated to an arc, shown in Algorithm 3, calls the `CanAccept` behavior of the destination (specified by the property *to*).

Each queue defines a behavior `CanAccept` that returns *true* if the queue has enough space to accept an incoming customer. Concerning the RSQueue and the BbsSoQueue the algorithm checks if the total length of the queue (property *length*)

Algorithm 2 `CanSend`

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

Algorithm 3 `HasSpace`

```
1: return to.CanAccept();
```

is less than the available space (property *capacity*) as reported in Algorithm 4.

Algorithm 4 `CanAccept` - BbsSoQueue and RSQueue

```
1: return length <= capacity;
```

In the case of the BbsSnoQueue, the algorithm considers the fact that the capacity can be reduced because the destination is full, as shown in Algorithm 5.

Algorithm 5 `CanAccept` - BbsSnoQueue

```
1: if length < capacity − 1 then
2:    return true;
3: else
4:    return CanSend() ∧ length < capacity;
5: end if
```

The `Fire` behavior implements the end of the service of a customer in a queue. In the case of the BbsSoQueue and BbsSnoQueue, it sends the customer to the next station using the definition provided in Algorithm 6.

Algorithm 6 `Fire` - BbsSoQueue and BbsSnoQueue

```
1: for all a ∈ Arc where a.from = this do
2:    a.Push();
3: end for
4: length = length - 1;
```

The RSQueue algorithm checks if the destination queue is empty before sending the customer there. If the destination is full, it reschedules the same service. This is implemented in Algorithm 7. Note that instructions for the rescheduling are not necessary (this is automatically done in the `InitEvents` behavior).

Both algorithms use the `Push` behavior of the connecting arc to send the customer to the next station. The `Push` behavior in turn calls the `AddOccupancy`(1) behavior of the queue at the other end of the arc, which is implemented in the same way for all the queue types. The `AddOccupancy`(c) behavior adds c customer to the length of each queue.

The behaviors used to compute the performance metrics have been implemented as follows: i) `CountStateRewards` returns the number of queues in the model; ii)

Algorithm 7 `Fire` - RSQueue

```
1: for all a ∈ Arc where a.from = this do
2:   if a.HasSpace() then
3:     a.Push();
4:     length = length - 1;
5:   end if
6: end for
```

`ListImpulseRewards` returns a list of their identifiers; iii) `ComputeStateRewards` returns a vector with the length of the queues in the current state; iv) `SetStateRewards` fills the *meanLength* property, and v) `SetImpulseReward` sets the *Throughput* property.

4.2 Gordon and Newell Queueing Networks

GNQN is another variant of QN. A GNQN is a closed network in which every queue has a server with service time according to the EEF property and first come first served policy.

Table 2 defines the elements required to implement GNQNs in SIMTHESys. The main differences with respect of the previous example are that in this case a queue can have more than one possible destination and has no space constraint. In this case a probability associated with the outgoing arc (property *prob*) determines the frequency at which a given destination is chosen.

All the behaviors have exactly the same implementation as for the TFCQN case, except for `InitEvents` and `Fire`. Gordon and Newell queueing networks are characterized by the property of choosing the next station either at the end of a service or at its beginning. Given the probability p_i of choosing the i-destination for a service operating at rate λ, due to the EEF property the action of scheduling a service at rate λ first and then choosing the destination with probability p_i is equivalent to the action of scheduling one service for each possible destination i at rate $p_i \cdot \lambda$. The `InitEvents` exploits this property to implement the choice of the next station, following Algorithm 8.

Algorithm 8 `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)", q.id, 1);
5:     end for
6:   end if
7: end for
```

Note that in this case the event is scheduled at rate $a.prob \cdot q.rate$. In this case the `Fire` behavior of a Queue has a parameter that defines the destination of the service. The `Fire` behavior is thus implemented following Algorithm 9.

Element	Property	Type	Modifier	Behaviors
GNQN				InitEvents, ComputeStateRewards, CountStateRewards, SetStateRewards, ListImpulseRewards, SetImpulseRewards
Arc	from to prob	element element float	const const const	Push
Queue	length meanlength rate throughput	integer float float float	status computed const computed	IsActive, AddOccupancy, Fire

Table 2
Elements of the GNQN SIMTHESys definition

Algorithm 9 `Fire`(a)

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

4.3 Stochastic Petri Nets

Petri Nets (PN) is a formalism suitable for modeling concurrent systems. A PN is composed of four kinds of elements: the arc, the inhibitor arc, the place and the transition.

SPN ([15]) are a variant of PN that takes into account a time variable and in which enabled transitions fire according to the EEF property: thus a transition in SPN is characterized by a rate.

SPN can be defined in SIMTHESys in table 3. The complete FDL description of SPN is given and commented in [14].

In the `InitEvents` behavior the formalism checks if all transitions are enabled and schedules the firing of those enabled at the corresponding rate following Algorithm 10.

The `IsActive` behavior of a transition looks both for incoming arcs and inhibitor arcs, and is reported in Algorithm 11. Both inhibitor and standard arcs implement the `IsActive` behavior by checking that the marking of the incoming place (read using the `GetOccupancy` behavior of the place) is respectively less, or greater or equal

Element	Property	Type	Modifier	Behaviors
SPN	bounded	boolean	computed	InitEvents, ComputeStateRewards, CountStateRewards, SetStateRewards, ListImpulseRewards, SetImpulseRewards
Arc	weight from to	integer element element	const const const	IsActive, Push, Pull
Inhibitor Arc	weight from to	integer element element	const const const	IsActive
Place	marking meantokens	integer float	state computed	GetOccupancy, AddOccupancy
Transition	rate throughput	float float	const computed	IsActive, Fire

Table 3
Elements of the SPN SIMTHESys definition

Algorithm 10 `InitEvents`

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

to their weight. Finally, when a transition fires it updates the marking following the `Fire` behavior described in Algorithm 12.

Note that in this case the arcs implement two behaviors, i) `Push` and ii) `Pull` that respectively add and subtract as many tokens as their weight from the place connected to the other end of the arc using the `AddOccupancy` behavior. Performance indices are implemented by defining a state reward for each place (its mean number of tokens), and an impulse for each transition (its throughput).

Algorithm 11 `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 12 `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
```

5 Solution Engines

SIMTHESys provides two solution engines for the EEF presented in Section 3, that solve models using discrete event simulation and steady state analysis of the underlying Markov chain (the engine requires this to be finite). Both engines take a *snapshot* of the state of the model by storing all the properties with the *status* modifier, and then back track to it.Also, they implement the `Schedule` behavior by storing all the scheduled events into a list. Note that as SIMTHESys aims at supporting multiformalism development, currently the solving engines have not been optimized. They are presented here to show that it is possible to solve models written in a large variety of formalisms without having to re-design new solvers by hand. The comparison with other known approaches to modeling and evaluation of systems is outside the scope of this paper.

5.1 Stochastic Simulation

Due to the EEF property, it is sufficient for the simulator to reschedule all the events after each firing. The simulator repeats the analysis for a fixed number N_{runs} of runs. Each run is executed until a global time T_{max} is reached; statistics are collected only after a transient time of fixed length T_{trans}. The parameters N_{runs}, T_{max} and T_{trans} are constants defined by the modeler. A snapshot of the initial state is taken, and after each run has finished, the snapshot is used to start a new simulation from the same initial state. The execution of each run calls the `InitEvents` behavior to find all the enabled events, and then draws an exponentially distributed sample for each of them. The event with the shortest sample is executed, and time is advanced accordingly until T_{max} is reached. At the end of all the simulation runs, statistics are collected and returned to the model using the `SetStateRewards` and `SetImpulsReward` behaviors.

5.2 Numerical Analysis

The numerical solution solver generates the CTMC (Continuous Time Markov Chain) that describes the stochastic process equivalent to the model. Starting from the initial state, the solver calls the `InitEvents` behavior to compute all the enabled events. The solver then builds a transition graph, executing each enabled event. Then, it checks if the snapshot of the obtained state has already been encountered (if not, it adds a new state) and backtracks to the initial event. The process is repeated until all the states have been visited. The use of snapshots is twofold: to reset the properties to a previously encountered state, and to backtrack to the current state whenever an enabled event is considered. Event firing rates are used to label the arcs of the transition graph. Each time a new state is found, the solver computes the associated reward vector: for state rewards it accounts for the values returned by the `ComputeStateRewards` behavior, while for impulse rewards, it considers the reward value multiplied by the rate of the corresponding enabled event. The generator matrix C of the underlaying Markov chain is next computed from the transition graph. Finally, the solver computes the steady state solution vector π of the Markov chain by computing $\pi C = 0$, and normalizing the solution such that all the components of π sums up to 1. Performance indices are computed by multiplying π times the reward vectors, and stored back into the model using the `SetStateRewards` and `SetImpulsReward` behaviors.

5.3 Example

In Figure 1 a closed TFCQN with three servers is taken as example [13].

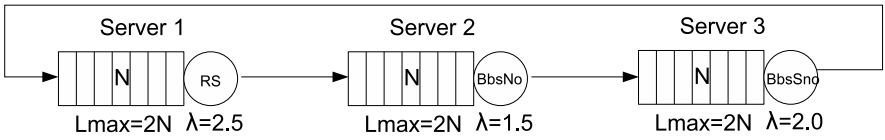

Fig. 1. Example TFCQN, with N initial customers per queue

The capacity Lmax and the rate λ for each queueing station are specified together with the initial number of customers N. The model has been solved with both analytical and simulative engines for different values of N. The results for the simulative and analytical engines are sketched respectively in Figure 2 and 3. Confidence intervals for the simulation are omitted for the sake of brevity. For $N = 20$ the model has 1801 states. For $N = 100$, as the number of states grows proportionally to N^3, the java implementation of the numerical solution is not able to compute the results (though the optimization of the engines is not in the scope of this paper, a C++ implementation has provided promising results). As expected, all queueing stations have the same throughput except for RS, that re-issues the requests in case of block. The slowest queueing station is saturated while RS serves all the customers that did not manage to have access to it. The third server has a stable number of customers, tending to the mean length of the same queueing station if isolated in

[13] More complex examples can be found on SIMTHESys web site at www.dem.unina2.it/simthesys

an open network with infinite capacity $(\rho/(1-\rho))$, where $\rho = \lambda/\mu$, $\lambda = 1.5$ and $\mu = 2$.

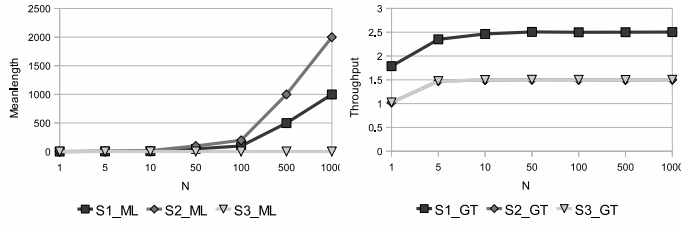

Fig. 2. Results for Simulation

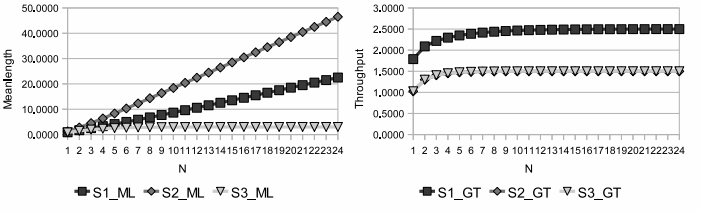

Fig. 3. Results for CTMC

6 Conclusions and future work

In this paper three example formalisms have been implemented by a new approach to multiformalism modeling. The SIMTHESys approach is based on the use of the concept of behavior to define formalisms and their semantics independently of the solver used. The three examples exploit the same solution engines and demonstrate the advantages of this approach. This research is the first detailed description of how a formalism and its solver can be implemented in SIMTHESys. Research in this area aims to better understand the possibilities opened up by the behavior mechanism. This could be achieved by implementing more formalisms linked to different families, solution engines and enhancing the usability of the behaviors by designing a dedicated scripting language. Other interesting perspectives are given by further investigation into multiformalism models and their application to different real world problems.

References

[1] Bause, F., Buchholz, P., Kemper, P., 1998. 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, pp. 356–359.

[2] Ciardo, G., Jones, III, R. L., Miner, A. S., Siminiceanu, R. I., June 2006. Logic and stochastic modeling with SMART. Perform. Eval. 63, 578–608.

[3] Clark, G., Courtney, T., Daly, D., Deavours, D., Derisavi, S., Doyle, J. M., Sanders, W. H., Webster, P., 2001. 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, pp. 241–.

[4] Courtney, T., Derisavi, S., Gaonkar, S., Griffith, M., Lam, V., McQuinn, M., Rozier, E., Sanders, W. H., 2005. The Mobius modeling environment: Recent extensions - 2005. Quantitative Evaluation of Systems, International Conference on 0, 259–260.

[5] Courtney, T., Gaonkar, S., Keefe, K., Rozier, E., Sanders, W. H., 2009. Mobius 2.3: An extensible tool for dependability, security, and performance evaluation of large and complex system models. In: DSN. IEEE, pp. 353–358.

[6] Deavours, D. D., Clark, G., Courtney, T., Daly, D., Derisavi, S., Doyle, J. M., Sanders, W. H., Webster, P. G., 2002 The Mobius framework and its implementation.
URL citeseer.ist.psu.edu/525756.html

[7] S. Derisavi, P. Kemper, W. H. Sanders, and T. Courtney, 2003 The Mobius State-level Abstract Functional Interface. Performance Evaluation, vol. 54, no. 2, October 2003, pp. 105-128.

[8] Franceschinis, F., Gribaudo, M., Iacono, M., Mazzocca, N., Vittorini, V., Aug. 2002. Towards an object based multi-formalism multi-solution modeling approach. In: Proc. of the Second International Workshop on Modelling of Objects, Components, and Agents (MOCA'02), Aarhus, Denmark, August 26-27, 2002 / Daniel Moldt (Ed.). Technical Report DAIMI PB-561, pp. 47–66.

[9] Franceschinis, G., Gribaudo, M., Iacono, M., Marrone, S., Moscato, F., Vittorini, V., 2009. Interfaces and binding in component based development of formal models. In: Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools. VALUETOOLS '09. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, pp. 44:1–44:10.

[10] Gordon, W., Newell, G., 1967. Closed queueing systems with exponential servers. Op Res 15 (2), 254–265.

[11] Gribaudo, G., Iacono, M., Mazzocca, M., Vittorini, 2003. The OsMoSys/DrawNET Xe! languages system: A novel infrastructure for multi-formalism object-oriented modelling. In: ESS 2003: 15th European Simulation Symposium And Exhibition.

[12] Gribaudo, M., Raiteri, D. C., Franceschinis, G., 2005. DrawNET, a customizable multi-formalism, multi-solution tool for the quantitative evaluation of systems. In: QEST. pp. 257–258.

[13] Gribaudo, M., Sereno, M., 1997. Gspn semantics for queueing networks with blocking. In: Proceedings of the 6th International Workshop on Petri Nets and Performance Models. IEEE Computer Society, Washington, DC, USA, pp. 26–.

[14] Iacono, M., Gribaudo, M., 2010. Element based semantic and behavior based composition in multi formalism performance models. In: MASCOTS. pp. 413-416. IEEE (2010).

[15] Kartson, D., Balbo, G., Donatelli, S., Franceschinis, G., Conte, G., 1994. Modelling with Generalized Stochastic Petri Nets. John Wiley & Sons, Inc., New York, NY, USA.

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

[17] Sahner, R. A., Trivedi, K. S., Puliafito, A., 1996. Performance and Reliability Analysis of Computer Systems; An Example-based Approach Using the SHARPE Software Package. Kluwer Academic Publisher.

[18] Sanders, W. H., Courtney, T., Deavours, D., Daly, D., Derisavi, S., Lam, V., 2007, Multi-formalism and multi-solution-method modeling frameworks: The Mobius approach.
URL citeseer.ist.psu.edu/685964.html

[19] Trivedi, K. S., 2002. 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, p. 544.

[20] Vittorini, V., Franceschinis, G., Gribaudo, M., Iacono, M., Mazzocca, N., April 2002. DrawNET++: Model objects to support performance analysis and simulation of complex systems. In: Proc. of the 12th Int. Conference on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation (TOOLS 2002). London, UK.

[21] Vittorini, V., Iacono, M., Mazzocca, N., Franceschinis, G., 2004. The OsMoSys approach to multi-formalism modeling of systems. Software and System Modeling 3 (1), 68–81.