Performance Evaluation 147 (2021) 102192

Contents lists available at ScienceDirect

Performance Evaluation

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

On identity-aware replication in stochastic modeling for simulation-based dependability analysis of large interconnected systems

Silvano Chiaradonna *, Felicita Di Giandomenico, Giulio Masetti

ISTI-CNR, Pisa, Italy

ARTICLE INFO

Article history:
Received 24 December 2019
Received in revised form 21 December 2020
Accepted 24 January 2021
Available online 2 February 2021

Keywords:
Stochastic models
Dependability analysis
Simulation-based evaluation
Template-based non-anonymous replication
Performance evaluation
Large interconnected systems

ABSTRACT

This paper focuses on the generation of stochastic models for dependability and performability analysis, through mechanisms for the automatic replication of template models when identity of replicas cannot be anonymous. The major objective of this work is to support the modeler in selecting the most appropriate replication mechanism, given the characteristics of the system under analysis. To this purpose, three most used solutions to identity-aware replication are considered and a formal framework to allow representing them in a consistent way is first defined. Then, a comparison of their behavior is extensively carried out, with focus on efficiency, both from a theoretical perspective and from a quantitative viewpoint. For the latter, a specific implementation of the considered replication mechanisms in the Möbius modeling environment and a case study representative of realistic interconnected infrastructures are developed.

1. Introduction

Stochastic model-based approaches are widely applied to assess system dependability attributes [1]. A variety of modeling formalisms and model solution techniques, typically automated in tools, have been developed since decades to assist the modeler's activity. Powerful software packages in this area, widely adopted in academia as well as in industry, include GreatSPN [2], Möbius [3] and SHARPE [4,5]. Unfortunately, the pace at which modern and future systems evolve in terms of largeness of components population and inherent intricacy among them, makes the model-based analysis very challenging, especially from state space explosion and models resolution time points of view.

Modularity and composition are widely recognized (e.g., [6,7]) as foundational principles to manage system complexity and largeness when applying model-based analysis. Typically, template (also called generic) models are first defined, tailored to represent the general behavior of system components at the desired level of abstraction. Then, from template models, individual instances are generated to account for the whole population of system components belonging to the respective categories. The system model is finally obtained by aggregating the specified submodels through properly defined composition rules. Therefore, most of the modeling environments and tools currently available have operators to automate the modularity and composition approach.

Replication can be addressed either in terms of anonymous identity of the system components belonging to the category represented through a template model (referred in the following as "anonymous replication"), or providing a

* Corresponding author.
E-mail addresses: silvano.chiaradonna@isti.cnr.it (S. Chiaradonna), felicita.digiandomenico@isti.cnr.it (F. Di Giandomenico), giulio.masetti@isti.cnr.it (G. Masetti).

https://doi.org/10.1016/j.peva.2021.102192
0166-5316/

specific identity to individual instances of a template model (referred in the following as "named replication"). When the former is assumed, faster model resolution approaches can be exploited than in the latter case, mainly due to the resulting symmetry and exploitation of suitable techniques, such as lumping [8]. However, the choice between anonymous or named replication is not done on the basis of opportunity for more efficient replication techniques, but on its realism given the application context under analysis. While anonymity fits well many system scenarios, there are a number of other contexts where population of similar (but not identical) components actually require an individual identity to properly analyze the impact of correct/faulty behavior of individual components on the overall system dependability, e.g. because of the position occupied by the component in the system topology. Critical infrastructures, such as in the transportation and electrical sectors, are examples of such systems, where a named replication is necessary to maintain adequate accuracy when modeling the component structure and behavior. Considering that such systems are typically large ones, with dependency relationships among constituting components, efficiency becomes of utmost importance when adopting a solution to named replication of template models.

In this paper, the focus is on discussing efficiency of techniques for named replication of stochastic template models for dependability and performability analysis. Interconnection among template instances is through state sharing, i.e. different submodels can change the value of a shared variable according to local actions [3]. A paired study focusing on action synchronization [9,10], the other alternative to manage template instances interconnections, is postponed as future work. General approaches, implementable on top of a variety of existing stochastic evaluation tools, are considered.

Aiming at supporting modelers in adopting template-based non-anonymous replication, major contributions are:

- formalization of the concept of named replication of template models, and of a context to express different approaches in a coherent format (both existing solutions and new ones, that could be devised in the future);
- formalization of three selected solutions for named replication of template models taken from the literature (referred as *SSRep*, *CSRep* and *DARep*) and discussion tailored to help a modeler to understand their characteristics in view of making the best mechanism selection;
- a simulation-based quantitative efficiency comparison of such selected approaches. To this purpose, implementations of *SSRep*, *CSRep* and *DARep* are developed in the popular Möbius tool, and applied to a representative case study. The strengths and limitations of the compared techniques in a variety of realistic system scenarios confirm the theoretical findings.

In order to address generic, realistic contexts, such as when targeted system components have non-Markovian behavior (e.g., due to deterministic time delays) and thus analytical approaches are not applicable, this paper concentrates on model replication strategies well suited to obtain simulation-based evaluations.

The paper is organized as follows. Pertinent state of the art solutions are overviewed in Section 2. Section 3 introduces the characteristics of the targeted systems. Section 4 provides the formal context for expressing replication mechanisms, in terms of formalism and necessary modeling features. Then, in Section 5 the three selected mechanisms *SSRep*, *CSRep* and *DARep* for non-anonymous modeling replication are formalized and compared in terms of efficiency on a theoretical basis. The second part of the paper concentrates on a quantitative comparison of a concrete implementation of *SSRep*, *CSRep*, and *DARep* in the Möbius modeling environment. To this purpose, a representative case study is introduced in Section 6, while the details of the implementations in Möbius are in Appendix A for the sake of readability. Then, extensive comparisons in terms of efficiency indicators of these implementations are in Section 7. Conclusions are drawn in Section 8. Finally, a table of acronyms is included in Appendix B.

2. Related work

Modeling for dependability related analysis is a long investigated topic and the corresponding literature covers decades of research activities. Even restricting to the challenging issue of addressing large systems, as imposed by modern critical application sectors and in line with the context of the presented work, the population of existing studies is such that an extensive literature review is unaffordable in a single paper. A plethora of techniques have been proposed, mainly aiming at the containment of state explosion, including truncation methods, parametric modeling, implicit representations, compositional methods. Refs. [11,12] detail examples of relevant families of such approaches. Given these premises, to position the contribution of this paper with respect to the state of the art, examined literature is restricted to solutions for replication of template-based models, with ability to preserve the identity of modeled components and adopting the state sharing approach to models interconnections.

Initial solutions mainly resorted to ad-hoc methods, as a simple way to account for instances of a template model that show identical local behavior, but different interactions with other components according to a specified topology. In the past, the interest was mainly in modeling anonymous components, that are indistinguishable both for their behavior and their interaction with other components. Anonymous replication has been automated in tool operators, such as the Rep operator in Möbius, and the observations that all the components have the same number of states, and all or none of them share a state variable, led to efficient analytical solvers, such as [8], or to enhance simulation [13].

However, as already said, there are a number of contexts where modeling adopting anonymous replication would lead to a too inaccurate analysis. Cyber-physical systems constitute a primary category where topology based modeling, leading to named replication, is required. A first direction taken in addressing named replication in state sharing interconnected

models was to define a submodel implementing the indexing of each replica to give them a specific identity, such as in [14–16] in the context of electrical power system analysis. Another direction resorted to manage the developed models through *XML* descriptors, exploiting the specific characteristics of the system under analysis, such as in [17,18] in the context of sensor networks. It needs to be remarked that the above delineated mechanisms have been adopted as ad-hoc expedients to deal with the named replication, but without any specific focus on them or desire to develop a general, automated schema, available to the modeler as a foundation building block. This aim was pursued later, and is the subject of the papers [19,20]. More precisely, in [19], the indexing mechanism originally presented in [14], has been formalized and generalized, originating *SSRep*, and a more efficient alternative, called *CSRep*. Pursuing efficiency improvement, *DARep* has been then developed in [20]; to a good extent, it appears as a well structured and generalized version of the ideas in [17]. This paper concentrates on the *SSRep*, *CSRep*, and *DARep* schemas because of their ability to be automated as general and structured solutions to be employed by modeler in a wide variety of contexts. The offered comparison study among these schemas is meant to be a valuable support to modelers in making the choice of the most appropriate solution to adopt, given the characteristics of the system under analysis.

While the above recalled solutions work at level of replication operator, other studies took the direction of developing methodologies for the automatic derivation of dependability models adopting model-driven engineering techniques, such as the recent works [21] (aiming at facilitating the selection, parameterization, and composition of predefined models from model libraries) and [22] (aiming at integrating different model generation chains). However, these solutions depart from the objective of this paper, which is the formalization and efficiency comparison of automated and generally applicable replication operators, immediately exploitable in existing and widely used modeling tools.

3. Logical representation of the addressed system category

This paper focuses on systems composed of a large number of components, in general grouped in different categories, in accordance to the nature and function they perform, and organized according to well known structures (topologies, e.g., tree, mesh, etc.). The connection among components induces a dependency among the state, and so in general the behavior, of the involved components.

Cyber-physical systems in critical sectors, such as transportation, electricity, water and oil, fit well in the addressed system category. For example, an electrical grid encompasses, among its components, a number of collection points called buses. All buses have the same aim, that is collecting and distributing electricity, so they belong to the same component category and their models are similar. The main differences among them are the position occupied in the grid topology and the number and kind of attached electrical equipments for energy production or consumption. Moreover, depending on the topology of the grid, dependencies are in place among subsets of buses connected through power lines. For example, a lightning that hits a bus can damage its equipment and, interrupting the current flow, can also damage the equipment linked to other nearby buses.

Therefore, when abstracting the components of the reference system for analysis purpose, for each category (e.g., the bus category) a *generic* component can be assumed as representative of all the *specific* system components belonging to that category (the several buses present in the grid segment under analysis). Then, each specific component, characterized by individual peculiarities, can be considered a named instance of such generic component, specifically distinguished through an index. Following this approach, the logical architecture of the given reference system can be seen as composed of:

- A large number of connected components, in general belonging to different typologies (e.g., buses, or power lines, or substations in electrical power systems). A generic component C^g is associated to each typology, which represents a subset of specific components (i.e., non-anonymous replicas or instances) denoted with C_i, where i is the index of the replica.
- One or more topologies that define the interactions among specific components (either belonging to the same generic component or spanning different generic components).

Without loss of generality, but for the sake of simplicity, in the following only one C^g describing n specific components C_0,..., C_{n-1} is considered.

The definition of C^g requires that all its parameters (such as those related to the initial state, the state changing, the random choices and times) can be defined as a function of the parameter i, with $i = 0, \ldots, n-1$.

3.1. System state, component interactions and actions

The definition of a generic component is based on the abstract notion of *state variables* and *actions* (events) as proposed in the Möbius Abstract Functional Interface (AFI) [23] for stochastic discrete event systems. This level of abstraction allows to represent generic components, dependencies among specific components and composition operators in general terms, independently from the specific formalisms in which the underlying stochastic models are expressed.

The generic component C^g includes l *template* State Variables (SVs) [20]: $V^0, V^1, \ldots, V^{l-1}$. Each V^h represents n different SVs (replicas or instances of V^h), one for each specific component C_i, in the overall system logical structure: $V_0^h, V_1^h, \ldots, V_{n-1}^h$. Thus, the entire system state is determined by assigning values to all the $l \cdot n$ SVs.

The interactions between two components are defined as the ability of one component to modify the state of the other component. They are modeled like in [20] through information sharing via one or more SVs. The interactions among C_i and C_j are defined by the $n \times n$ adjacency matrices $\mathcal{T}^h = [\mathcal{T}^h_{i,j}]$, representing the *topology of interactions* based on individual V^h. In particular, each ith row and each jth column of $\mathcal{T}^h$ represent respectively all the replicas of V^h that can be accessed by C_i and all the replicas of C^g that can access V^h_j. Formally, if C_i has read/write access to V^h_j for $j \neq i$ then $\mathcal{T}^h_{i,j} = 1$, else $\mathcal{T}^h_{i,j} = 0$. Thus, the SVs V^h can be classified into two categories:

Local state variables If V^h_i can be accessed by C_i only.

Dependency-aware state variables If V^h_j can be accessed by C_j and, depending on the topology $\mathcal{T}^h$, by other component(s) C_i, with $i \neq j$.

Fixed V^h in C^g, the *dependency degree* of C_i with respect to V^h, called $\delta^h_i \in \{1, 2, \ldots, n-1\}$, indicates how many replicas of V^h among $V^h_0, \ldots, V^h_{n-1}$ can be read/written by C_i, while $\hat{\delta}^h_j$ is the number of C_i that can access to V^h_j. Similarly, the array of indexes j of those SVs V^h_j that can be accessed by C_i is called Δ^h_i and $\Delta^h_i[k]$ is the kth element of the array, $k = 0, \ldots, \delta^h_i - 1$. Formally:

$$\delta^h_i = \sum_j \mathcal{T}^h_{i,j}, \quad \hat{\delta}^h_j = \sum_i \mathcal{T}^h_{i,j} \quad \text{and} \quad \Delta^h_i = \left(j \mid \mathcal{T}^h_{i,j} = 1\right) \in \mathbb{N}^{\delta_i}. \tag{1}$$

The state changes are defined by means of actions. In particular, the generic component C^g includes m *template* actions: $a^0, a^1, \ldots, a^{m-1}$, where each template action a^k represents n different actions, one for each replica of C^g: $a^k_0, a^k_1, \ldots, a^k_{n-1}$. Thus, the entire system is governed by $m \cdot n$ actions.

An action can be *enabled* according to conditions defined on SVs values. When an enabled action is *completed*, it can change the value of all the SVs that are accessed by C_i. Each action can take a deterministic or random (with arbitrary distribution) time period between its enabling and completion. Each a^k (parameters, enabling conditions and SV value changes at action completion) is defined as a function of the parameter i, where i is the index of C_i, such that it can naturally represent an action a^k_i. Moreover, a^k can use $\Delta^h_i[p]$ to read/write access to V^h_j such that $j = \Delta^h_i[p]$. An action in C_i can be defined *only* as a function of SVs that C_i can read/write.

3.2. Overall logical component definition

Summing up, a generic component C^g is defined by a set of template state variables SVs, a set of template actions and a set of topologies. Formally:

$$C^g = \left(V^0, \ldots, V^{l-1};\ a^0, \ldots, a^{m-1};\ \Delta^0, \ldots, \Delta^{l-1}\right). \tag{2}$$

The overall system behavior along time is represented by $l \cdot n$ SVs and $m \cdot n$ actions. Starting from C^g, n specific components are defined by those SVs that they can access and the set of actions they own, formally:

$$C_i = \left(\bigcup_h \bigcup_{j \in \Delta^h_i} V^h_j;\ a^0_i, \ldots, a^{m-1}_i\right). \tag{3}$$

4. Modeling preliminaries

Following the formal definition of the category of addressed systems, the aim of this section is to present a modeling formalism for:

1. defining a template model M^t to realize the generic component C^g,
2. automatically producing from M^t the *specific* models M_0,..., M_{n-1} to realize the specific components C_0,..., C_{n-1}, respectively, and
3. composing the *specific* models M_i to form the overall system model M^{sys}.

The model M^{sys} represents a stochastic discrete event system at level of AFI [23]. The underlying stochastic process depends on the specific formalism used to define M^t. Let M^t be a template model based on the general concepts of SVs and actions, and M_i be an instance of such template model. In order to model C^g with a single template model, M^t has to be defined as a function of the parameter i representing the index of M_i, as described in Section 3. In particular, the definition of each template action a^k must include read/write access to each instance of V^h listed in Δ^h_i. There exists a great variety of formalisms to define the model M^t based on SVs and actions, each one with its specificities, suitable to define Markovian, semi-Markov and non Markovian stochastic models. Among the most popular ones, we mention:

- Performance Evaluation Process Algebra [24], which explicitly tackles actions, while SVs are implicitly defined by composition rules among actions. PEPAk [25] is an interesting dialect that allows a direct definition of SVs and can easily implement the approaches described in this paper.

- The Stochastic Fault Trees family [26], where *node*, *failure event* and *gates* correspond respectively to SV, action and composition of failure events.
- The Stochastic Petri Nets family [27], where each *place* corresponds to a SV, and each *transition* corresponds to an action.

Therefore, although in principle any of the above mentioned formalisms (and others not mentioned as well) would be a good candidate to express template-based named replication mechanisms we are interested in this paper, it needs to be clarified that most of them would probably require extensions in order to be able to define the parametric template model M^t. This aspect of extensions is not addressed in this paper. We believe that modelers familiar with a specific formalism can easily understand its current limitations and directions for appropriate extensions. Being familiar with the SAN formalism, we state that it can be used to define M^t with minimum or no extension (depending on the considered approach), thanks to the powerfulness provided by the C++ language.

In the following, since the interest is in a general presentation of the concepts and facilities for modeling template-based named replication, a notation similar to that adopted for the system in (2) and (3) is used.

4.1. Model compositional features

As already discussed when presenting the related work, in this paper we restrict to model replication and joining operators that are state-sharing based, as made available by popular modeling frameworks. They are the *Rep* and *Join* operators [28], exploited in the definition of the named replication solutions of interest in our study. Their formalization is given in the following.

The *Join* operator combines two or more submodels into a single composed model, based on *distinguished* (or shared) SVs defined in the individual submodels to allow communication among them. Notation adopted for the *Join* operator and the resulting model are shown in the following example, where two models $M_5 = \left(A, B, C;\ a_5^0, a_5^1\right)$ and $M_7 = \left(D, E, F, G;\ a_7^0, a_7^1\right)$ are joined as follows:

$$\mathcal{J}\left(\ M_5, A, B, C, \emptyset; M_7, \emptyset, D, E, F\ \right) = \left(A, B, C, F, G;\ a_5^0, a_5^1, a_7^0, a_7^1\right). \tag{4}$$

A list of distinguished SVs is associated with each composing model (A, B, C with M_5 and D, E, F with M_7), in a manner such that particular entries are allowed to be empty ($\emptyset$) and the cardinality of each list is the same (4). In the joined model, all the not empty distinguished SVs corresponding to the same position in each list are merged to form a single SV ($B = D$ and $C = E$), i.e., the SVs are shared among the submodels. Each distinguished SV is exposed to other compositional operators, whereas undistinguished SVs are local to the submodel (G is local to M_7). In the whole resulting model defined on the right-side of (4), the occurrences (if any) of D and E in a_7^0 and a_7^1 (as defined in M_7) are replaced by the names B and C, respectively, used for the shared SVs.

The *Rep* operator applied to a template model produces a new model obtained joining n anonymous replicas (identical copies) of the template model, where the distinguished SVs are shared among all the replicas. Thus, a SV can be local to each replica or shared among all replicas (all-or-none sharing strategy). *Rep* does not assign any index to the anonymous replicas.

The *Rep* operator can be used to *automatically generate* n replicas $M_0,\ldots, M_{n-1}$ of M^t, where the index i of each replica is the value of a SV defined as local to the replica. The notation adopted is shown in the following example, where $M^t = \left(V^0, V^1;\ a^0\right)$ is replicated sharing V^0 to obtain the replicas $M_0 = \left(V_0^0, V_0^1;\ a_0^0\right), \ldots, M_{n-1} = \left(V_{n-1}^0, V_{n-1}^1;\ a_{n-1}^0\right)$ with $V_0^0 = \cdots = V_{n-1}^0$, as follows:

$$\begin{aligned}\mathcal{R}_n\left(M^t, V^0\right) &= \mathcal{J}\left(M_0, V_0^0; \ldots; M_{n-1}, V_{n-1}^0\right)\\ &= \left(V_0^0, V_0^1, V_1^1, \ldots, V_{n-1}^1; a_0^0, \ldots, a_{n-1}^0\right).\end{aligned} \tag{5}$$

Observe that:

1. *Join* takes as input a set of already defined models, thus cannot be used to *automatically generate* system component models.
2. *Join* can directly address the matrix representing $\mathcal{T}^h$ for $h = 0, \ldots, l-1$, because the modeler can manually define all M_i following (3) and share only the SVs, as defined in $\mathcal{T}^h$. However, this process is error prone and does not scale at increasing of the number n of system components.
3. Using *Rep* it is possible (although not straightforward) to define at level of template model an index to refer specific replicas.
4. Since a SV can be shared among all the replicas or none of them, *Rep* cannot directly address $\mathcal{T}^h$ when V^h is a dependency-aware state variable.

Thus, using only the *Join* operator as it is, no template model M^t can be defined to represent C^g and then produce individual components models. On the other hand, it is possible to work around the fact that *Rep* is natively an anonymous replication operator and adopt it to produce named replicas and to address the interactions among components.

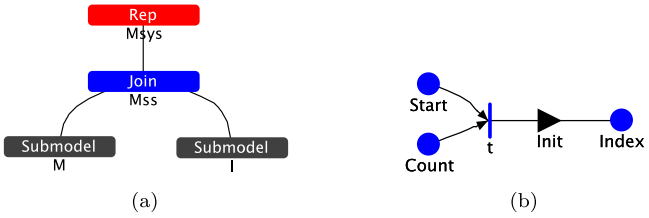

Fig. 1. The *SSRep* approach as implemented in Möbius: composed *Rep* model M^{sys} Fig. 1(a) and template SAN model *I* initializing the index of the replicas Fig. 1(b).

5. Named replication

In the following, the definition of the three approaches *SSRep* [14,15], *CSRep* [19], and *DARep* [20], already proposed to model automatically the interactions among components that require named replication, is presented.

To avoid too heavy notation, but without losing generality, only one template state variable V^h and one template action a^k are considered in M^t.

5.1. The State-Sharing Replication approach

The *SSRep* approach relies on n SVs, one for each replica of the template, for each dependency-aware SV V^h of C^g. *SSRep* has not been conceived to exploit the dependency topology when defining the variables shared among the replicas of the template. Thus all the instances of the dependency-aware SVs are shared among all the replicas of the template model, independently of the topology.

The *SSRep* approach consists of a simple architecture, as depicted in Fig. 1(a): a single template model, called M^{ss}, is replicated n times by the *Rep* operator to obtain the system model M^{sys}, where an index is assigned to each replica.

M^{ss} is structured in two parts: (i) one defining the generic component C^g as a function of the index of the replicas (represented by the SV $Index \in \mathbb{N}$), including explicitly a SV for each instance of V^h, the data structure Δ^h (representing the topology associated to V^h) and a^k, and (ii) one that initializes the indexes for each replica of M^{ss}, including the SVs $Count \in \mathbb{N}$ and $Start \in \mathbb{N}$, and the instantaneous action *tInit*, as shown in Fig. 1(b). Formally:

$$M^{ss} = \left(V_0^h, \ldots, V_{n-1}^h, Index, Count = n, Start = 1; a^k, tInit; \Delta^h\right). \tag{6}$$

The *Rep* operator generates n replicas M_i of M^{ss}:

$$M_i = \left(V_0^h, \ldots, V_{n-1}^h, Index_i, Count, Start_i; a_i^k, tInit_i; \Delta_i^h\right), \tag{7}$$

that are composed through sharing the SVs $V_0^h, \ldots, V_{n-1}^h$ and *Count*, whereas $Index_i$ and $Start_i$ are local to M_i, to generate the overall system model:

$$\begin{aligned} M^{sys} &= \mathcal{R}_n\left(M^{ss}, V_0^h, \ldots, V_{n-1}^h, Count\right) \\ &= \left(V_0^h, \ldots, V_{n-1}^h, \mathrm{Index}_0, \ldots, \mathrm{Index}_{n-1}, Count, \mathrm{Start}_0, \ldots, \mathrm{Start}_{n-1}; a_0^k, \ldots, a_{n-1}^k, \mathrm{tInit}_0, \ldots, \mathrm{tInit}_{n-1}; \Delta^h \right). \end{aligned} \tag{8}$$

Count is used to initialize the index of each replica Index_i at completion of tInit_i, through the following steps, in the order specified from left to right:

$$Count = Count - 1,\ \mathrm{Index}_i = Count,\ \mathrm{Start}_i = 0, \tag{9}$$

where tInit_i is enabled if $Start_i == 1$ and $Count > 0$. All the actions a_i^k are enabled after the initialization of all the indexes, i.e., when $Count == 0$.

Notice that each instance of V^h, being shared among all replicas of the template model, can be read/write accessed by each M_i, independently of Δ_i^h. Thus, *SSRep* assumes a complete graph of interactions among components, even if the actual number of ones in $\mathcal{T}^h$ is much less than n^2. Obviously, the definition of each action a^k is based on the actual interactions among components, as defined in $\mathcal{T}^h$, using Δ^h. For example, the enabling condition of the action a^k can be given in the template model by $V_j^h = 1 \,\forall j \in \Delta_{Index}^h$. But, when an action affecting V^h fires within M_i, all the instances of a^k, not only those listed in Δ_i^h, are checked to determine their current enabling status.

5.2. The Channel-Sharing Replication approach

The *CSRep* approach was the first to exploit the knowledge of the actual topology of dependencies among the replicas of a template model. It relies on:

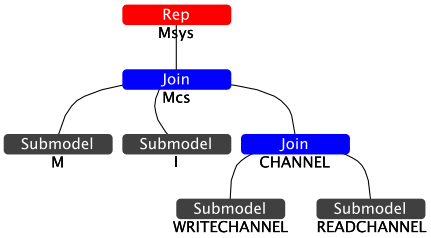

Fig. 2. The *CSRep* approach as implemented in Möbius: composed *Rep* model.

1. defining as local to each M_i all its dependency-aware SVs following $\mathcal{T}^h$,
2. using a small-sized channel composed of a small number of SVs, shared among all the M_i, to exchange (synchronize) the values of dependency-aware SVs shared among the replicas M_i, each time they are updated.

In particular, δ^h local SVs $W^0, \ldots, W^{\delta^h-1}$ are defined in the template model, with $\delta^h = \max_i \delta_i^h$, to represent the set of δ_i^h instances of V^h listed in each Δ_i^h, such that, in each model M_i, $W^p = V_j^h$ with $j = \Delta_i^h[p]$.

As for *SSRep*, also *CSRep* defines a single template model, called M^{cs}, that is replicated n times by the *Rep* operator, as depicted in Fig. 2. The template M^{cs} is structured in three parts: (1) the model of C^g as a function of the index of the replicas, represented by the SV *Index* $\in \mathbb{N}$, including the SVs W^p, the data structure Δ^h and a^k, (2) the initialization of the indexes, following the description already provided for *SSRep*, (3) the channel and the read/write channel operations, including *Channel* $\in \mathbb{N} \times \mathbb{N} \times$ *DataType*, *ToChannel* $\in \mathbb{N}$, *ReadyChannel* $\in \mathbb{N}$ and the instantaneous actions *read* and *write*, where *DataType* is the type of the data to synchronize. Formally, the architecture of *CSRep* is:

$$M^{cs} = \big(W^0, \ldots, W^{\delta^h-1}, \mathit{Index}, \mathit{Count} = n, \mathit{Start} = 1, \mathit{ToChannel} = 0, \\ \mathit{ReadyChannel} = 1, \mathit{Channel}; a^k, \mathit{tInit}, \mathit{read}, \mathit{write}; \Delta^h\big). \tag{10}$$

The SV *ToChannel*, initialized to 0, is local to each replica and it is used to trigger (*ToChannel*== 1) the writing on the channel each time a dependency-aware SV modeled by W^p is updated. The SV *ReadyChannel*, initialized to 1, is shared among all replicas and is used to lock the channel (*ReadyChannel*== 0) from writing until the data are read by all the destination replicas.

The extended SV *Channel* = (*index*, *ndest*, *data*) is shared among all replicas M_i and is defined by 3 fields (SVs). The SV *index* $\in \mathbb{N}$ represents the index j of the updated dependency-aware SV V_j^h modeled by W^p. The SV *ndest* $\in \mathbb{N}$ is the current number of destination M_i that have not yet read the data (initialized with the number of M_i to synchronize). The SV *data* $\in$ DataType contains the value of V_j^h to send to the M_i. The field *Channel.index* is used to identify whether M_i is a destination depending on $\mathcal{T}^h$. The field *ndest* is used to unlock the channel when all the destination M_i have read the SV *data* (Channel.ndest == 0).

The action a^k extends the corresponding action of C_i with the assignment ToChannel = 1, each time a local state variable W^p is updated.

The *Rep* operator generates n replicas M_i of M^{cs}:

$$M_i = \big(W_i^0, \ldots, W_i^{\delta^h-1}, \mathit{Index}_i, \mathit{Count}, \mathit{Start}_i, \mathit{ToChannel}_i, \mathit{ReadyChannel}, \mathit{Channel}; a_i^k, \mathit{tInit}_i, \mathit{read}_i, \mathit{write}_i; \Delta_i^h\big), \tag{11}$$

that are composed sharing the SVs *Count*, *ReadyChannel* and *Channel*, to generate the overall system model, where *none* of W^p is shared among replicas:

$$\begin{aligned} M^{sys} &= \mathcal{R}_n\big(M^{cs}, \mathit{Count}, \mathit{ReadyChannel}, \mathit{Channel}\big) \\ &= \big(W_0^0, \ldots, W_0^{\delta^h-1}, \ldots, W_{n-1}^0, \ldots, W_{n-1}^{\delta^h-1}, \mathit{Index}_0, \ldots, \mathit{Index}_{n-1}, \mathit{Count}, \\ &\quad \mathit{Start}_0, \ldots, \mathit{Start}_{n-1}, \mathit{ReadyChannel}, \mathit{Channel}, \mathit{ToChannel}_0, \ldots, \mathit{ToChannel}_{n-1}; \\ &\quad a_0^k, \ldots, a_{n-1}^k, \mathit{tInit}_0, \ldots, \mathit{tInit}_{n-1}, \mathit{read}_0, \ldots, \mathit{read}_{n-1}, \mathit{write}_0, \ldots, \mathit{write}_{n-1}; \Delta^h\big). \end{aligned} \tag{12}$$

The actions *write* and *read* are enabled respectively by the sender replica M_s to write the data into the channel, and by all the destination replicas M_i to read the data from the channel. In particular, *write* is enabled if:

$$\text{ToChannel} == 1 \text{ and ReadyChannel} == 1, \tag{13}$$

i.e., when, respectively, there are new data to send and the channel is ready to accept the data. At completion of *write*, the channel is updated and locked, performing the following assignments in the order specified from left to right:

$$\text{Channel.index} = \text{j}, \text{Channel.ndest} = \hat{\delta}^h_{\text{Index}}, \text{Channel.data} = W^p, \text{ToChannel} = 0, \text{ReadyChannel} = 0, \tag{14}$$

where $W^p = V_j^h$ is the value that has to be transmitted through the channel.

The action *read* is only enabled when the channel has been updated with new data to send and the current replica is the destination of the data of the channel (and it is not the sender of the data), i.e.:

$$\text{ReadyChannel} == 0 \text{ and } \exists q | \text{Channel.index} == \Delta_i^h[q]. \tag{15}$$

At completion of *read*, the following assignments are performed in order:

$$W^q = \text{Channel.data}, \text{Channel.ndest} = \text{Channel.ndest} - 1,$$
$$\text{if Channel.ndest} == 0 \text{ then ReadyChannel} = 1, \tag{16}$$

i.e., the local W^q and the state of the channel are updated. Then, if the replica is the last destination of the data, the channel is unlocked.

Thus, whenever an action a_s^k of M_s changes the value of W_s^q, equal to V_j^h with $j = \Delta_s^h[q]$, then the action $write_s$ immediately performs the assignments in (14). Then all the $read_i$ for which $j \in \Delta_i^h$ perform the assignments in (16), among them $W_i^q = Channel.data$.

When more than one replica of one, or more than one, dependency-aware SV are updated at the same time, a sequence of consecutive transmissions using the same channel can be easily modeled. Alternatively, the channel can be easily extended by adding new fields.

5.3. The Dependency-Aware Replication approach

The *DARep* approach takes advantage of the topology of dependencies among system components, by sharing subsets of the instances of each dependency-aware SV among only those replicas M_i that need to access them.

Differently from the *SSRep* and *CSRep* approaches, *DARep* proposes a new compositional operator in order to: (1) generate automatically the instances of the dependency-aware SV associated to each M_i, (2) share different subsets of the instances of the dependency-aware SV among different subsets of M_i, following $\mathcal{T}^h$. Thus, it merges the advantages of the *Join* and *Rep* operators.

In more details, the *DARep* approach is based on:

1. two functions *Index()* and *Deps()* that extend the template model to represent, respectively, the index of the replicas of the template model and the instances of the dependency-aware SVs occurring in each replica of the template model, following the actual system topology,
2. a new state-sharing template-based compositional operator that, automatically: (i) generates the indexed replicas of the template model supporting *Index()* and *Deps()*, and (ii) generates the overall system model, by composing through the *Join* operator the indexed replicas of the template.

The *DARep* approach defines a single template model, called M^{darep}, that is replicated n times by the newly defined $\mathcal{D}$ operator. M^{darep} is defined as follows:

$$M^{\text{darep}} = \left(V^h; a^k; \text{Index}(), \text{Deps}()\right). \tag{17}$$

Index() and *Deps()* are two functions used in the definition of the actions of the template and supported by the $\mathcal{D}$ operator. Notice that Δ^h is moved from the template model (where it is replaced by the function *Deps()*) to the $\mathcal{D}$ definition.

The $\mathcal{D}$ operator first generates n indexed replicas M_i of M^{darep} and in particular for each M_i generates the list of δ_i^h instances of each V^h shared among different M_i, following the topology $\mathcal{T}^h$ associated to each V^h:

$$M_i = \left(\{V_j^h | j \in \Delta_i^h\}, a_i^k; Index_i(), Deps_i()\right), \tag{18}$$

where

$$\text{Index}_i() = i, \forall i = 0, \ldots, n-1,$$
$$Deps_i(h, s) = V_j^h | j = \Delta_i^h(s), \text{ for all } h, s \text{ and } i, \tag{19}$$

i.e., $Deps(h, s)$ is used in M^{darep} to access to the sth instance of V^h automatically generated by $\mathcal{D}$ in the Index()th replica of M^{darep}.

Then $\mathcal{D}$ composes the generated submodels M_i, as depicted in Fig. 3, sharing all the instances of V^h having the same name in different M_i, to generate the overall system model:

$$\begin{aligned} M^{\text{sys}} &= \mathcal{D}\left(M^{\text{darep}}, V^h, \Delta^h\right) \\ &= \mathcal{J}\left(M_0, W_0^0, \ldots, W_0^{n-1}; \ldots, M_{n-1}, W_{n-1}^0, \ldots, W_{n-1}^{n-1}\right) \\ &= \left(V_0^h, \ldots, V_{n-1}^h; a_0^k, \ldots, a_{n-1}^k\right), \end{aligned} \tag{20}$$

where

$$W_i^j = \begin{cases} V_j^h & \text{if } j \in \Delta_i^h, \\ \emptyset & \text{otherwise.} \end{cases} \tag{21}$$

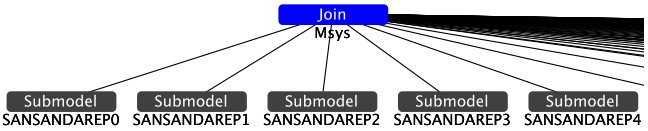

Fig. 3. The *DARep* approach as implemented in Möbius: the left part of the composed *Join* model automatically generated from the template M^{darep} for the case study described in Section 6 with $n = 100$ and $\delta = 74$.

5.4. Considerations on the presented named replication approaches

From the descriptions above, the following features are required in order to allow named replication of a given template model:

1. data structures to manage the parameters of the template model M^t, as a function of a generic replica index of the template, in particular the topology associated to each dependency-aware SV, that is the array Δ^h;
2. an indexing mechanism for M_i, used to access the parameters of each replica of the template and to define the actions as a function of the index;
3. for each replica M_i of the template, constant-time read/write access to the replicas of each dependency-aware SV V^h shared among the other replicas of the template as defined in Δ_i^h;
4. automatic generation of M_i from the template M^t, where interactions among M_i occur through dependency-aware SVs, differently modeled in each of the three approaches;
5. automatic generation of the final composed model joining the specific models M_i and addressing the interactions among M_i through dependency-aware SVs.

As already discussed in Section 4, feature 1 depends on the formalism adopted to define the template model. In this paper, the SAN formalism is adopted to define the template models for all the approaches.

With regard to the feature 2, *SSRep* and *CSRep* use the same indexing mechanism, that relies on a SV "Index" local to each replica M_i, i.e., the index of each replica is part of the state of the replica. Moreover, *SSRep* and *CSRep* are based directly on the *Rep* operator, that supports above features 4 and 5.

DARep is based on a new operator that explicitly supports 2 and 4 and on the *Join* operator that supports 5.

For all the approaches, feature 3 is obtained defining the replicas of the dependency-aware SVs with arrays (for *SSRep* and *CSRep* at definition of the template, while for *DARep* automatically when the replicas M_i are generated).

The major implications derived from how these features are used by the different approaches are discussed in the following. They are the basis for a theoretical comparison of the replication mechanisms, and mainly relate to:

- The ability to account for the actual dependency topology, so to avoid unnecessary interactions among instances. Both *CSRep* and *DARep* take into account the interactions topology, while *SSRep* always relies on a complete interactions graph. In fact, *SSRep* is able to account for the actual dependency topology at run time only after the start of the simulation, when the indexes of the replicas are defined. In addition to the potential overhead incurred by *SSRep* because of this extended interactions graph, there is also the fact that erroneous accesses to SVs (i.e., accesses that do not follow the actual system topology) cannot be automatically avoided. Therefore, from this perspective, *SSRep* has in principle disadvantages with respect to the other two approaches;
- The ability to take into account the actual dependency topology in building the actions connectivity lists, used by the simulator to trace the enabling status of other actions after an action fires. Being based on the *Join* operator, *DARep* is able to share specific SVs among specific submodels. Thus *DARep* supports the creation of action connectivity lists [29] following the interactions topology $\mathcal{T}^h$. Action connectivity lists can allow for a significant reduction in the simulation time in most models [29]. *Join* provides statically (at compilation time) the simulator with the information necessary to determine which actions within a submodel may affect other actions within other submodels based on $\mathcal{T}^h$. On the contrary, *Rep* limits the connectivity lists optimization, such that each action that affects a shared SV is connected to all the replicas of actions linked to (affected by) the SV. In fact, being the distinguished SVs shared among all the replicas, *Rep* cannot statically identify which specific actions of a replica are affected by actions within other replicas as defined by the interactions topology. Thus, *SSRep* works considering a complete interactions graph also at connectivity lists generation. Also *CSRep*, although it exploits the actual topology at level of operational structures, suffers from the same limitations as *SSRep* for what concerns the connectivity lists, being not able to follow the interactions topology for all the activities connected to dependency-aware SVs and for $read_i$, respectively. This can lead to a significant simulation overhead in *SSRep* and *CSRep* that increases with the number of replicas and dependency-aware SVs. Therefore, from this perspective, the *SSRep* and *CSRep* approaches have in principle disadvantages with respect to *DARep*;
- Incurred overhead due to complex structures and coordination activities. From the point of view of model construction, *SSRep* is the heaviest, since it needs more complex structures to build all the local SVs and make them shared among all the replicas. Instead, from the coordination activities point of view, *CSRep* pays the highest price, since

each change of W_s^q, equal to V_j^h, produces 1 new event for $write_j$ and 1 new event for each $read_i$, for which $j \in \Delta_i^h$. This implies a time overhead due to the synchronization of the dependency-aware SVs each time they are updated.

- Modeling simplicity. From this perspective, the template model M^{darep} is simpler than those defined for SSRep and CSRep. In fact, once the *Index()* and *Deps()* functions are made available in a certain modeling environment, they can be used by modelers working in that environment without requiring knowledge of the details about assignment of replica indexes and dependency-aware SVs. This simplification is expected to be reflected in an easier and less error-prone modeling activity.
- Compilation versus runtime overhead. A time consuming aspect is related to the automatic management of the replicas of the template model and the dependency topology, to correctly model the access to SVs by the different replicas. In both *SSRep* and *CSRep*, this is accomplished at runtime, while *DARep* has a compilation time management. In principle, the different behavior shown by the approaches does not imply a systematic advantage of one over the others in terms of efficiency, since the dimension and dependency degree of the system under analysis have a high impact and, depending on the specific values, could favor one behavior or another.

5.5. From the theoretical definition to a concrete implementation

Although the theoretical observations discussed in Section 5.4 are useful to derive trends and to point out expectations from the approaches definition, the final selection of the best solution requires knowledge of the modeling context where the approach is going to be implemented. In fact, a theoretical advantage shown by an approach over another could result significantly lessened due to inefficiency of the implementation environment, and vice versa.

To provide evidences to this claim, implementation of *SSRep*, *CSRep* and *DARep* in the Möbius environment [3] and their application to a case study are addressed. The aim is to conduct a quantitative comparison analysis in a concrete context. The case study is introduced in Section 6, and the feasibility of the proposed approaches in the Möbius framework, through their application to the case study in a variety of scenarios, is provided in Appendix A.

6. Case study

Although defined for illustration purposes only, the selected case study is effective in demonstrating features, benefits and limitations of the approaches described in Section 5, since it fully represents the logical architecture of targeted systems described in Section 3. Moreover, it can be considered as a basis to be easily extended and adapted to represent a great variety of real contexts.

In a cloud computing context, two different services A and B can be requested from clients. The infrastructure is composed of n virtual machines, called here nodes, and each can host: (1) one server for A, or (2) one server for B, or (3) two servers, one for A and one for B. On each node i, the demand (service requests) of service $S \in \{A, B\}$, at any time instant t, is $D_{S,i}(t) \geq 0$. At any time instant t, node i has maximum capacity $C_{S,i}$ to satisfy service S requests, where $D_{S,i}(t) \leq C_{S,i}$. The matrix $\mathcal{T}^S$ represents the topology of interactions among servers providing service S on different nodes. Δ_i^S is the list of nodes from which the server of service S on node i depends on.

Demand $D_{S,i}(t)$ can increase or decrease due to:

- Random interaction with clients: the demand of service S follows a Markovian stochastic process so that, after an exponentially distributed period of time, the demand increases or decreases, depending on the demand trend.
- Interaction with neighbors: whenever $D_{S,i}(t) > C_{S,i}$ or the server reboots or recovers, the server of service S on node i tries to dispatch the exceeding demand respectively $D_{S,i}(t) - C_{S,i}$ or $D_{S,i}(t)$ (for reboot or recovery) to other working servers of the same service on nodes in Δ_i^S.

The exceeding demand of service S on node i is divided into δ_i^S equal parts $E_{S,i}(t) = (D_{S,i}(t) - C_{S,i})/\delta_i^S$. For each working neighbor $j \in \Delta_i^S$, $E_{S,i}(t)$ is sent to node j, if $E_{S,i}(t) \leq C_{S,j} - D_{S,j}(t)$, decreasing $D_{S,i}(t)$ and incrementing $D_{S,j}(t)$ by $E_{S,i}(t)$. Otherwise, if the server on node j cannot satisfy all the demand $E_{S,i}(t)$, i.e., $E_{S,i}(t) > C_{S,j} - D_{S,j}(t)$, then only a part of $E_{S,i}(t)$ equal to $C_{S,j} - D_{S,j}(t)$ is sent to j, decreasing $D_{S,i}(t)$ by $C_{S,j} - D_{S,j}(t)$ and increasing $D_{S,j}(t)$ to the maximum capacity $C_{S,j}$. Of course, if the node j is down or it does not host a server for service S, then $E_{S,i}(t)$ is not sent to j. Thus, it is not guaranteed that all $D_{S,i}(t) - C_{S,i} > 0$ can be redispatched.

To resemble more realistic application scenarios, if the total time $T_{S,i}^{overc}(t)$ in which $D_{S,i}(t) \geq C_{S,i}$ overcomes a pre-set threshold $T_{S,i}^{threshold}$, for at least one service S, then the state of node i is switched from working to down and a reboot is performed. The reboot takes a deterministic amount of time and has success with probability c. In case it fails, or after a given number of reboots n^{rcv}, a recovery process is launched on node i. The recovery operation can take considerably longer time with respect to reboot, but guarantees a complete reset of node i. During reboot and recovery, all demand $D_{S,i}(t)$, for each service S, is not satisfied. In addition to overload conditions, after an exponentially distributed period of time, with rate depending on the value of the current demand, each node also experiences a failure, which requires a recovery action.

This case study is more complex than the one exercised in [19,20], since: (i) the number of shared variables, kept to 1 in the previous publications, becomes 3, and (ii) 2 read/write access topologies instead of 1 are introduced. This leads

to more realistic and challenging system scenarios from the replication point of view. The SAN models for this case study are described in Appendix A.5.

To compare the replication approaches, the above described system has been evaluated in terms of expected: unsatisfied demand on node i for the server providing service S, unsatisfied demand on node i and global unsatisfied demand. Since the goal is to compare the performance of the three replication approaches, the obtained results for these dependability measures are not shown in Section 7.

7. Numerical comparison of the approaches

A comparison of the performance results of the Möbius implementations of *SSRep*, *CSRep* and *DARep* when applied to the case study described in Section 6 has been conducted. The characteristics of the system under analysis (in terms of the size and dependency degree), as well as the accuracy requested to the analysis output, have been varied to explore a variety of scenarios. The terminating Möbius simulator [3] has been used to evaluate the dependability related measures described in Section 6. As a form of validation of the developed models, for some sample models, it has been verified that the number of stable states obtained with all the approaches is the same and it is equal to the theoretical prediction. In addition, the computed measures resulted in the same values for all the approaches. Different reward structures [30,31] over different set of markings have a different impact on simulation times. Thus, to improve accuracy and fairness of the comparison, a high number of reward variables (around 40) has been considered in the study. However, since here the analysis focuses on the comparison of the performance of the approaches, details on these measures and the obtained results are out of the scope of this paper.

Each execution of the terminating Möbius simulator is defined for a specific setting of all the parameters of the considered models. Each execution of the terminating simulator starts initializing the data structures, then runs k batches (independent model simulations in Möbius terminology) with $k \geq 1$.

7.1. Efficiency metrics

For the comparison, the following performance measures have been considered, relative to one execution of the Möbius simulator that runs k batches ($k \geq 1$):

- $\tau(k)$: Total amount of *CPU* time, in seconds, used by the Möbius simulator. It includes the initialization time and the time necessary to run k batches.
- τ_{init} or $\tau(0)$: The amount of *CPU* time, in seconds, used by the Möbius simulator to initialize its data structures. This is the *CPU* time used by the simulator to output the string "SIMULATOR::Preparing to run()". The definition of τ_{init} as a function of $\tau(k)$ is: $\tau_{init} = \tau(1) - (\tau(2) - \tau(1)) = 2\tau(1) - \tau(2)$, where $\tau(2) - \tau(1)$ is the total amount of *CPU*, in seconds, used by the Möbius simulator to run a batch.
- $\Delta\tau(k)$: Difference between the total CPU time to run k batches and the initialization time:

$$\Delta\tau(k) = \tau(k) - \tau_{init}.$$

- ${}^{D}_{S}\mathcal{C}\tau(k)$: Compared simulation performance (dimensionless ratio) between *DARep* and *SSRep*, defined as follows:

$${}^{D}_{S}\mathcal{C}\tau(k) = \frac{\tau^{\text{DARep}}(k)}{\tau^{\text{SSRep}}(k)}. \tag{22}$$

- ${}^{C}_{S}\mathcal{C}\tau(k)$: Compared simulation performance (dimensionless ratio) between *CSRep* and *SSRep*, defined as follows:

$${}^{C}_{S}\mathcal{C}\tau(k) = \frac{\tau^{\text{CSRep}}(k)}{\tau^{\text{SSRep}}(k)}. \tag{23}$$

- ${}^{D}_{C}\mathcal{C}\tau(k)$: Compared simulation performance (dimensionless ratio) between *DARep* and *CSRep*, defined as follows:

$${}^{D}_{C}\mathcal{C}\tau(k) = \frac{\tau^{\text{DARep}}(k)}{\tau^{\text{CSRep}}(k)}. \tag{24}$$

The considered *CPU* time includes both user and system *CPU* times.

7.2. Simulation settings

The following topologies $\mathcal{T}^A$ and $\mathcal{T}^B$, having the same structure but involving different indexes, are chosen for the case study. The first three quarts of the virtual machines host a server for service A, i.e., $i = 0, \ldots, n_A - 1$ with $n_A = \lceil 0.75 \cdot n \rceil$, the last three quarts host a server for service B, i.e., $i = n_B - 1, \ldots, n - 1$ with $n_B = \lceil 0.25 \cdot n \rceil$ (notice that virtual machines in the intersection host servers for both service A and service B). The degree of interaction δ is the same for all i, i.e., $\delta = \delta_i^A = \delta_i^B$, where δ_i^h is defined in (1). The demand re-dispatching follows a cyclic graph, i.e., $\Delta_i^S = \{i, (i+1) \bmod n_S, \ldots, (i+\delta) \bmod n_S\}$.

To exercise the approaches in a variety of relevant contexts, scenarios generated as combinations of the following values for n, δ and k, have been considered:

Table 1
Initialization time τ_{init} (seconds) for the *SSRep* approach.

		δ			
		1	7	74	148
n	10^1	0.019	0.013		
	10^2	3.556	3.806	3.623	
	10^3	29637.500	29723.400	29623.600	29673.200

Table 2
Initialization time τ_{init} (seconds) for the *CSRep* approach.

		δ			
		1	7	74	148
n	10^1	0.015	0.018		
	10^2	0.118	0.142	0.438	
	10^3	4.789	5.592	12.251	22.452

Table 3
Initialization time τ_{init} (seconds) for the *DARep* approach.

		δ			
		1	7	74	148
n	10^1	0.015	0.012		
	10^2	0.021	0.033	1.746	
	10^3	0.309	0.746	23.414	137.610

Table 4
Batch time $\Delta\tau(1000)$ (seconds) for the *SSRep* approach.

		δ			
		1	7	74	148
n	10^1	0.412	0.437		
	10^2	48.873	48.476	50.306	
	10^3	5382.400	5216.000	5373.300	5455.600

- number of batches k varying from 1 to 1000. The value of k has impact on the precision of the obtained results, and 1000 has been selected to assure convergence with confidence interval width lower than 10^{-5},
- number n of replicas ranging from 10 to 1000,
- dependency degree δ varying from 1 (minimum connectivity) to 148.

Simulations were sequentially performed on Intel(R) Core(TM) i7-5960X with fixed 3.50 GHz CPU, 20M cache and 32GB RAM, and an up to date GNU/Linux Operating System. Each $\tau(k)$ has been evaluated 10 times and the arithmetic mean is taken as final result. It is important to notice that, for the *SSRep* and *CSRep* approaches, models compilation times are negligible, being constituted by a few single atomic SANs and one composed model, while for the *DARep* approach component models compilation time can be relevant. In particular, if $n \leq 100$ then atomic SANs compilation times and composed model compilation time can take few minutes, while, for $n = 1000$ and $\delta = 148$, composed model compilation time can grow up to about 10 hours. However, it is important to notice that recompilation is required *only* if the structure of the template model or the topology of interdependencies is changed. Thus, changing all the other model parameters, such as failure rates or service request rates, as well as changing the measures under evaluation, does not have an impact on compilation time. In addition, investigations are currently in progress to understand how to promote parallel models compilation to improve on compilation time.

7.3. Comparison results

First, the results relative to the τ_{init} indicator for the three approaches are presented in Table 1, Tables 2 and 3, respectively.

From the inspection of Table 1 two important conclusions can be drawn: for *SSRep*, the initialization phase time is almost independent from the dependency degree δ, but strongly correlated with the system size n. This confirms the already predicted behavior, as discussed when presenting this approach in Section 5.1. Instead, Tables 2 and 3 show that both *CSRep* and *DARep* initialization times are almost linear in n and differently correlated with δ, with τ_{init} of *DARep* growing faster than *CSRep* at increasing δ. Again, these results are in line with what already predicted in Sections 5.2 and 5.3, by observations regarding the behavior of these approaches. It is important to notice that the presence of two read/write access topologies, $\mathcal{T}^A$ and $\mathcal{T}^B$, drastically impacts the *SSRep* initialization phase performance, well beyond the predicted $\mathcal{O}(n^2)$ (see Section 5.1 and Appendix A.2). Also, there are insights that the numbers presented in Table 1 suffer from implementation issues concerning the model construction operated by *Rep* in Möbius [32].

The results obtained for the next performance indicator under analysis, the batch time $\Delta\tau(k)$, are presented for each approach in Table 4, Table 5 and Table 6, respectively. In detail, Table 4 shows the batch time $\Delta\tau(1000)$ for *SSRep*.

Table 5
Batch time $\Delta\tau(1000)$ (seconds) for the *CSRep* approach.

		δ			
		1	7	74	148
n	10^1	0.838	2.580		
	10^2	74.272	272.091	2455.352	
	10^3	9273.481	24185.708	216642.749	*NaN*

Table 6
Batch time $\Delta\tau(1000)$ (seconds) for the *DARep* approach.

		δ			
		1	7	74	148
n	10^1	0.164	0.204		
	10^2	6.786	8.067	17.513	
	10^3	1185.831	1292.934	1536.186	2482.280

Table 7
Compared performance ${}^{D}_{S}\mathcal{C}\tau(k)$ (dimensionless ratio) between the *DARep* and *SSRep*.

$\delta = 10$		k				
		1	10	100	1000	10000
n	10^2	0.0168	0.0378	0.104	0.156	0.195
	10^3	0.0001	0.0005	0.004	0.042	0.223

Table 8
Compared performance ${}^{C}_{S}\mathcal{C}\tau(k)$ (dimensionless ratio) between the *CSRep* and *SSRep*.

$\delta = 10$		k				
		1	10	100	1000	10000
n	10^2	0.139	0.975	4.141	7.195	7.905
	10^3	0.002	0.016	0.118	1.052	6.696

Table 9
Compared performance ${}^{D}_{C}\mathcal{C}\tau(k)$ (dimensionless ratio) between the *DARep* and *CSRep*.

$\delta = 10$		k				
		1	10	100	1000	10000
n	10^2	0.121	0.039	0.025	0.022	0.025
	10^3	0.043	0.030	0.038	0.040	0.033

The evaluation of batches during the simulation, as can be inferred from the considerations in Section 5.1, has a time complexity of $\mathcal{O}(n^2)$ and it is almost independent from δ.

For the two approaches *CSRep* and *DARep*, the results are shown in Tables 5 and 6, respectively. Not surprisingly, since they have been defined with the purpose to exploit the real dependency topology among system components, their batches execution times depend on both n and δ. In particular, it can be observed how *CSRep* batches time explodes at increasing of δ. In fact, $\Delta\tau(1000)$ for $n = 1000$ and $\delta = 148$ has not been reported because the total execution time exceeds a fixed upper limit.

Comparing *SSRep* and *CSRep* values, an interesting phenomenon can be highlighted. Summing up τ_{init} and $\Delta\tau(1000)$ for the case $n = 1000$, if $\delta \leq 7$ then $\tau_{init} + \Delta\tau(1000)$ for *SSRep* is greater than $\tau_{init} + \Delta\tau(1000)$ for *CSRep*, whereas for higher values of δ it is clear that $\tau_{init} + \Delta\tau(1000)$ for *CSRep* is greater than $\tau_{init} + \Delta\tau(1000)$ for *SSRep*. Thus, from the point of view of the modeler, if the target is to optimize the total simulation time, it is the value of δ that determines which approach is more convenient to adopt between *SSRep* and *CSRep*. If system components are loosely interconnected, i.e., δ is small, then *CSRep* performs better, otherwise *SSRep* outperforms *CSRep*.

Of course, also *DARep* fully exploits the topologies of read/write accesses, thus outperforming *SSRep*, and, having no time overhead due to the special SVs management (the channel), is always better than *CSRep*. Thus, without considering compilation times, *DARep* is the best choice for complex systems whatever the value of δ is.

The final part of the comparison is carried out in terms of the $\mathcal{C}\tau(k)$ indicator, for different values of k. Varying the number of batches k has an impact on the accuracy of the simulation results, so it is a parameter to be cautiously selected, in accordance with the criticality of the system under analysis and of the purpose of the analysis itself. Tables 7, and show the results of ${}^{D}_{S}\mathcal{C}\tau(k)$, ${}^{C}_{S}\mathcal{C}\tau(k)$ and ${}^{D}_{C}\mathcal{C}\tau(k)$, respectively, keeping δ fixed at 10. From Table 7, it can be immediately concluded that the *DARep* approach is always better than *SSRep*. From Table 8, a similar trend already observed between *CSRep* and *SSRep* at varying δ is shown also at varying k. In fact, at increasing k, *CSRep* initially outperforms *SSRep* (also depending on the value of n), but then it is the reverse. Actually, the assumed value for the dependency degree ($\delta = 10$) in this evaluation is already rather high for having *CSRep* really competitive with respect to *SSRep*. Last, Table 9 shows values of *DARep* always much better than those of *CSRep* (between 30 and 40 times better for the highest values of k and n).

7.4. Final discussion

Some final remarks on the conducted comparison study are drawn in the following. The performance of all the three approaches is affected by the size of the system and by the number of simulation batches: not surprisingly, the obtained values degrade at increasing of both n and k. With respect to the other considered parameter, namely the topology of interaction that is a key aspect of complex systems, the behavior among the three replication solutions is diversified. The *SSRep* approach is insensitive to δ, which results in high values of the initialization time τ_{init} and in quite high values of the batch simulation time: that grow higher than 8 h and about an hour and a half, respectively, when n is 1000 and k is 1000. *CSRep* exploits not only the topology of interactions but also the fine grained topology of read/write accesses with the channel mechanism. The benefits on the initialization time τ_{init} are great (just 22 s for the highest considered values for δ and n), but correspondingly the overhead required to manage the channel grows significantly (up to around 60 h when δ is 74 and n is 1000). Therefore, *CSRep* outperforms *SSRep* when the dependency degree is low (which is however not uncommon in relevant application sectors, such as in power grid systems, as already discussed in [19]). Applying a totally different idea as done by the *DARep* approach, which relies on an external *Perl* program in its implementation in Möbius, the performance is greatly improved: *DARep* is always the best in all the tables showing the evaluation results, whichever be the values of δ, n and k. However, it might suffer from long compilation times, although it has to be reminded that a recompilation is needed only when there are changes in the structure of the template model and/or in the interdependency topology.

From this study, we believe useful insights are provided to modelers facing model-based reliability/dependability evaluation of large, interconnected systems. On the basis of the characteristics of the system under analysis, the choice of the most appropriate modeling replication approach can be performed, so to make the assessment more efficient and affordable at the requested degree of accuracy.

The gained knowledge is also useful to reflect on the opportunity of performing modifications to the Möbius implementation that could bring benefits to (all or some of) the considered replication approaches. For example, as already observed in Section 7.3, the current implementation of the *Rep* operator in Möbius shows inefficiencies [32], with negative impact especially on those approaches that make heavy use of *Rep*. This is an aspect on which the Möbius developers are planning to improve in a next release of the tool, but from this study they could find inspiration also for other enhancements.

Finally, this study is a helpful guideline to reproduce a similar investigation in a modeling environment different from Möbius, but still amenable to realize implementations of *SSRep*, *CSRep*, and *DARep*. The expectation is that the major theoretical observations presented in Section 5.4 will be equally confirmed. Of course, it is likely that the results of performance indicators will differ; e.g., the values combination of system size and dependency degree which lead *SSRep* to outperform *CSRep* could be different, as well as the extent of the variation in efficiency among the three approaches.

8. Conclusion

This paper addressed template-based non-anonymous replication in systems composed of large populations of interconnected components, when simulation-based solvers are used. Since the demand for non-anonymous replication comes from a variety of key sectors for our society and economy, it is very important to understand the efficiency of the available solutions when copying with real-size systems, so to make the most appropriate choice.

Specifically, the study focused on the major solutions currently available, which are the State-Sharing Replication (*SSRep*), Channel-Sharing Replication (*CSRep*) and Dependency-Aware Replication (*DARep*) approaches. These are general mechanisms, applicable to any modeling and evaluation environment that supports automatic replication and composition of submodels based on sharing state variables. First, a rigorous formalization framework has been proposed, which embeds the concepts at the basis of modeling non-anonymous replication, suitable to express in a coherent format the adopted replication approaches. Then, a comparison among the approaches has been performed to better understand their peculiarities and expected performance. Both theoretical reasoning, based on the definition of the approaches, and a quantitative evaluation based on an implementation of the three solutions in the Möbius modeling environment have been carried out. Indicators representative of the execution and initialization time of the Möbius terminating simulator have been computed on a generic client–server application.

Several scenarios, characterized by different values of the number of replicas, the dependency degree and the number of simulation batches, have been analyzed.

From the simulation results, strengths and weaknesses of the compared approaches are highlighted. The *SSRep* and *CSRep* approaches have peculiar trends: when the system size is large but the dependency degree is small, *CSRep* performs much better, but the situation reverses at increasing the dependency degree. *DARep* shows the best performance, although its compilation time needs to be accounted for, as discussed in Section 7.

This study can inspire work in several directions.

Implementing *SSRep*, *CSRep* and *DARep* in a modeling environment different from Möbius would bring further feedback on the implications of the underlying technological choices provided by the adopted tool. Although Möbius is a widely used framework for dependability and performance analysis, it is not the only one; thus, extending the proposed replication mechanisms to other stochastic model-based frameworks would enlarge the population of modelers who would take advantage of them. One example can be GreatSPN [2,33].

A more long term objective would be to define and making native in the adopted evaluation tool the auxiliary functions *Index()* and *Deps()* and the operator $\mathcal{D}$. This is expected to improve the efficiency with respect to solutions built on top of the tool operators/features, as done so far. Compared to *DARep*, such a smart replicator operator would avoid resorting to the many files now needed, which require increasing (and possibly prohibitive) compilation time at increasing the number of replicas and their dependency degree.

Finally, the knowledge gained from the detailed inspection of the considered replication approaches and the related comparison study could trigger the definition of new solutions, in the attempt to further mitigate weak behaviors. The formal context has the ingredients to express other formulations of replication techniques based on composition and state-sharing features

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Named replication in Möbius

In this Appendix, the feasibility of the proposed approaches is illustrated, resorting to the SAN formalism and the Möbius framework. Using the case study in Section 6 for $n = 100$ and $\delta = \delta_i^S = 74$, the general steps each approach consists of are first detailed. Then, the models for the case study are described.

A.1. Overview of the Möbius modeling framework

The Möbius modeling framework and its supporting tool Möbius [3] are briefly recalled in the following. Our models are defined using the SAN formalism [28], a stochastic extension of Petri nets based on the following primitives: plain and extended places (blue and orange circles) represent SVs, timed and instantaneous activities (hollow and solid vertical bars) with linked input and output gates (triangles pointing left or right) represent actions. Extended places represent complex data types (like int, float, double, structures and arrays). Input gates control when an activity is enabled. The delay between enabling and completing of timed activities is a generally distributed random variable, whereas enabling and completing of instantaneous activities take place at the same time. SVs changes occur when an activity is completed, as defined by the input and output gates. The SAN primitives are defined by C++ statements, supporting external C++ data structures and the linking to external C++ libraries.

In Möbius, the *Join* and *Rep* state-sharing compositional operators [28] are supported at level of AFI [3,23] as already described in Section 4. The auxiliary functions *Index()* and *Deps()*, and the operator $\mathcal{D}$ are implemented through a Perl program [20,34] which manipulates the *XML* files describing the models defined in Möbius.

A.2. State-Sharing Replication implementation

The *SSRep* approach can be implemented in the Möbius framework by defining the composed model M^{sys} as depicted in Fig. 1(a), where:

- The *Rep* operator automatically constructs the overall system model M^{sys}, composed of the indexed replicas M_i of the template M^{ss}, as in (8).
- The template M^{ss} is defined by the *Join* operator as in (6) composing the atomic SAN M, which represents C^g, and the atomic SAN I, which initializes the place *Index* shared among the M and I but local to M^{ss}.

The action *tInit* of (6) and the steps in (9) are implemented by the activity t and the output gate *Init*, as shown in Fig. 1(b). When t completed, one token is removed from the places *Start* (local to M^{ss}) and *Count* (shared among replicas), and the C++ code of *Init* is executed: *Index*−>*Mark*() = *Count*−>*Mark*().

The SVs $V_0^h, \ldots, V_{n-1}^h$ listed in (8), are automatically implemented by an n-sized array-type extended place W defined in the model M and shared among all M_i, such that the ith entry of W, obtained with W−>*Index*(i)−>*Mark*(), corresponds to V_i^h. In M shown in Fig. A.4, there are two n-sized array-type extended places *Demand_A* and *Up*, representing the dependency-aware SVs.

Each action b^k in (6) is implemented by an activity and the linked gates. The place *Index* can be used to define parameters of activities and gates. For example, one of the 5 actions in Fig. A.4 is implemented by the activity *TUpdD_A*, with rate defined as a function *LambdaD*$_A$(*Index*−>*Mark*()) of the place *Index*, and the gates *is_A*, *Go*, *IncrD_A* and *DecrD_A*.

The topology $\mathcal{T}$ is modeled by C++ constant data structures defined at compilation time, e.g., Δ^h is a C++ array of n different-sized arrays of short. Thus, for example, in M shown in Fig. A.4, the entries of the place *Up* that are accessed by a replica of M are obtained by scanning the array Δ^h[*Index*−>*Mark*()], i.e., by *Up*−>*Index*(Δ^h[i][k])−>*Mark*() for $k = 0, \ldots, \delta_i^h - 1$, with i = *Index*−>*Mark*().

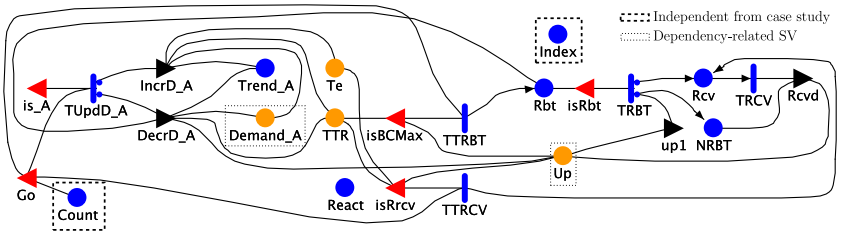

Fig. A.4. The *SSRep* approach: template SAN model M defining the generic component for the case study described in Section 6.

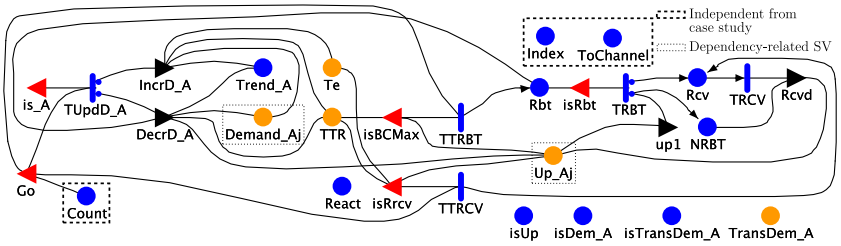

Fig. A.5. The *CSRep* approach: template SAN model M defining the generic component for the case study described in Section 6.

A.3. Channel-Sharing Replication implementation

The *CSRep* approach can be implemented in the Möbius framework by defining the composed model M^{sys} as depicted in Fig. 2, where:

- Like for the *SSRep* approach, the *Rep* operator automatically generates the overall model M^{sys}, composed of the replicas M_i of M^{cs}, as in (12).
- The template M^{cs} is defined by the *Join* operator as in (10) composing three models: M, which represents C^g, I, which initializes the place *Index* (local to M^{cs}), and the composed model *CHANNEL*, which models the channel and the related read and write operations.
- The model *CHANNEL* is defined by the *Join* operator composing the atomic SAN models *WRITECHANNEL* and *READCHANNEL* that are used, respectively, by the sender replica to write the data into the channel and by all the destination replicas to read the data from the channel.

The δ^h local SVs necessary to represent Δ_i^h, as listed in (12), are automatically implemented by an δ^h-sized array-type extended place W defined in M and local to each M_i. Therefore, the rth entry of W on M_i, obtained with $W{-}{>}Index(r){-}{>}Mark()$, corresponds to V_j^h with $j = \Delta_i^h[r]$, as defined in (10).

For example, in Fig. A.5, *Demand_Aj* and *Up_Aj* are two local δ^h-sized array-type extended places representing the dependency-aware SVs. Moreover M also includes, as defined in (10), the place *ToChannel*, which is local to M^{cs} and shared with the model *CHANNEL*. At completion of the activity *TUpdD_A* when the output gate *DecrD_A* updates one of the entries of *Demand_Aj*, then it also sets the place $ToChannel{-}{>}Mark() = 1$, triggering the activation of the model *WRITECHANNEL*.

The composed model *CHANNEL* is used to send the values of the dependency-aware SVs of a replica of M to other replicas. Each time a dependency-aware SV is updated, the following steps are performed:

1. the model *WRITECHANNEL* writes the new values (the data) in the channel and locks the channel (the channel is busy),
2. the model *READCHANNEL* of each destination replica updates the SVs of the replica with the data received from the channel,
3. the model *READCHANNEL* of the last destination replica that received the data unlocks the channel, which can be used to transmit new data.

Fig. A.6(a) and A.6(b) depict respectively *WRITECHANNEL* and *READCHANNEL*, when two dependency-aware SVs *Demand_Aj* and *Up_Aj* are considered for the case study described in Section 6. The place *Channel* implements an extended version of the channel given in (11). Thus, the values of both *Demand_Aj* and *Up_Aj* can be synchronized at the same time.

The activity *tw* and the output gate *wChannel* implement the action *write* in (10), in particular the enabling condition defined in (13) and the SV changes in (14), which update and lock the channel. The activity *tupd* with the ouptut gates *Read* and *Upd* implement the action *read* in (10), in particular in *Read* the enabling condition defined in (15) and in *Upd* the SV changes defined in (16).

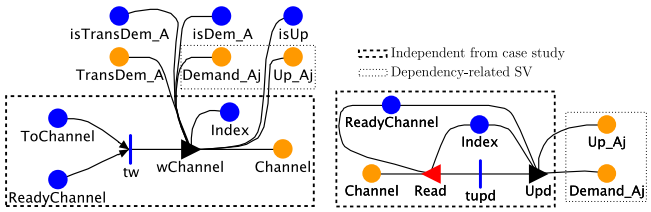

Fig. A.6. The *CSRep* approach: SAN models *WRITECHANNEL* Fig. A.6(a) and *READCHANNEL* Fig. A.6(b) used respectively to write and to read the shared data into the channel.

A.4. Dependency-Aware Replication implementation

The *DARep* approach can be implemented in the Möbius framework through the following steps, listed in order of execution:

1. Manual or automatic definition of the dependency topology associated to each dependency-aware SV, as in (1), and of the parameters of the system.
2. Automatic generation, based on the dependency topology and on the parameters of the system, of the C++ user defined library of Möbius.
3. Manual definition of the template atomic SAN model M^{darep} that represents the generic component, as in (17).
4. Automatic generation of the atomic models M_i, as in (18) and (20).
5. Automatic generation of the list $\mathcal{J}^{shared}$ of the replicas of the dependency-aware SVs shared among M_i, as in (20) and (21).
6. Automatic definition, using the outputs of the steps 4 and 5, of the composed model obtained through the operator *Join*, as in (20).

In [20], where *DARep* was initially proposed, the above automatic steps 2, 4, 5 and 6 were implemented through a XQuery script and run on the XQilla tool. However, since efficiency is of utmost relevance given the high numbers of files involved, in this paper a new *Perl* script, denoted by $\mathcal{D}^{perl}$, is proposed to implement such steps. We verified that the new script is faster than the XQilla one by around one order of magnitude.

Using the results of step 1, step 2 implements Δ_i^h, for each i, h with a different C++ constant array of short and the other system parameters with C++ data structures, like arrays, records and plain types. These C++ data structures, statically defined at compilation time, are automatically generated and included in the user defined library supported by Möbius, and then they can be used in each model defined in the tool. Steps 4, 5 and 6 implement the function $\mathcal{D}$, as in (20). Each model generated at steps 4 and 6 is defined in an *XML* file, automatically generated with *XML::LibXML*, a Perl Binding for libxml2, using the dependency topology described with an *XML* input file defined at step 1.

Each time the template SAN model undergoes updates at step 1 or 3, implying changes either in the dependency topology or in the number of replicas n, steps 4, 5 and 6 must be repeated, to update the resulting *XML* files and C++ files. Consequently the overall model must be compiled again.

Fig. 3 depicts the left part (the overall picture is omitted for the sake of space) of the composed model M^{sys} automatically generated at step 6, as in (20), for the case study described in Section 6, for $n = 100$ and $\delta = 74$. The SAN models *SANSANDARep*0, ..., *SANSANDARep*99 (their names are obtained merging the name SAN of the template, the string *SANDARep* and the index of the replica) are the models automatically generated at step 4, corresponding to the replicas M_i, $i = 1, \ldots, n-1$, as in (18) and (20).

At step 4, a place is automatically generated in each M_i for each entry of the array Δ_i^h associated to V^h. The name of each place V^h*DRSVDARep*$_j$ is obtained merging the name V^h, the string *DRSVDARep* and the index j of the replica V_j^h that the place represents. Thus, these places model only the replicas of each dependency-aware SV that are accessed by M_i, as defined in Δ^h. These places are also shared among the replicas M_i by the *Join* operator, as defined in the list $\mathcal{J}^{shared}$ automatically generated at step 5.

Fig. A.7 shows the SAN model *SANSANDARep0* generated for the 0th replica M_0. The bottom of the figure is omitted for the sake of space. Thus, only *Demand_ADRSVDARep0* and *Up_ADRSVDARep0* are shown of the 75 replicas of the SVs *Demand_A* and *Up_A* that are accessed by the model *SANSANDARep0*.

The template model M^{darep} defined at step 3 is an atomic SAN model, where either plain or extended places are used to represent dependency-aware SVs.

The functions *Index()* and *Deps()* are implemented by two C++ functions, that can be only used in the SAN template model, as follows:

SANSANDARep ::M^{darep}:: *Index*(),

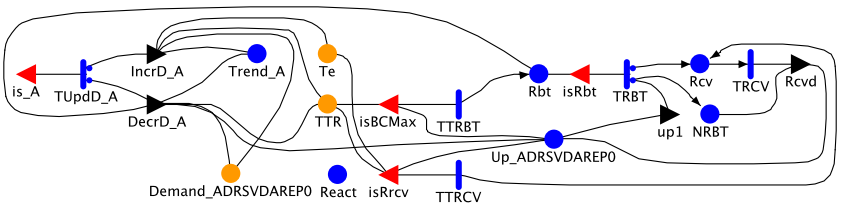

Fig. A.7. The *DARep* approach: upper part of the SAN model *SANSANDAREP0* corresponding to the 0th replica M_0 generated from the template M^{darep} defining the generic component for the case study described in Section 6 with $n = 100$ and $\delta = 74$.

$$SANSANDARep::M^{\text{darep}}::V^h->Deps(s), \tag{A.1}$$

where the strings *SANDARep* and M^{darep} are C++ namespaces introduced to avoid names conflicting with Möbius C++ code.

The C++ statement $SANSANDARep::M^{\text{darep}}::Index()$ is replaced in each atomic SAN generated at step 4 by the actual index of the modeled replica.

The C++ statement $SANSANDARep::M^{\text{darep}}::V^h->Deps(s)$ refers to the place $\Delta^h(s)$, as defined in (19). Without the index s, this statement can be used to pass to each user defined C++ function the list of references to all the places of Δ^h for the current replica of M^{darep}, to access the values of these places.

In each atomic SAN model M_i generated at step 4, the statement $SANSANDARep::M^{\text{darep}}::V^h->Deps(s)$ is replaced by

$$SANDARep::M^{\text{darep}}::rep(s).V^h(), \tag{A.2}$$

where the C++ object *rep* calls the actual method that returns the reference to the place $V^hDRSVDARep_j|j = \Delta_i^h(s)$, automatically defined in the generated SAN M_i. Moreover, in each SAN primitive where this statement is used (e.g., in the enabling condition of an input gate), all the actual names of the automatically generated places modeling Δ_i^h are included in the primitive through a call to a dummy empty function having all these names as arguments. This is needed because the links among SAN primitives (e.g., the activities) and places in the Möbius tool are statically defined when the C++ code describing the model is generated, based on the names of the places. Thus, for example, an enabling condition defined by a statement that accesses a place by reference, like the statement in (A.2) generated by *DARep*, is checked at each update of the place only if the enabling condition includes also the name of the place.

The C++ code (definition of classes and initialization of objects and constants) used to implement the method called by *rep()* in (A.2) and that relies on the dependency topology, is automatically generated and included in each submodel at step 4. In particular, the code to initialize the object resulting from *rep()* with the array of the pointers to the places modeling the array Δ_i^h, is included in the field "Custom Initialization" of each SAN; thus, the C++ data structures are set before the model simulation starts.

Finally, notice that the Perl-based implementation of *DARep* introduces a time overhead at generation time of the atomic and composed models (steps 4, 5 and 6) and at compilation time, due to the number n of the atomic models and to the size of the composed model. In particular, for very large values of n and δ_i, the time required for the generation and compilation of the composed model could have a relevant impact on the efficiency of the model evaluation.

A.5. SAN models representing the case study

For each approach *SSRep*, *CSRep* and *DARep*, the SAN model representing the generic component is composed of two parts, one for the service *A* and one for *B*. For the sake of brevity, only the part modeling the service *A* is described. The part modeling the service *B* is obtained duplicating all the primitives ending with "_A" (and the related arcs) and replacing the occurrence of "_A" with "_B" at the end of each string.

As shown in Figs. A.4, A.5 and A.7, the template SAN models defined for all the considered replication approaches have a similar structure (excluding the SAN models used for the channel in the *CSRep* approach). They mainly differ in the definition of the index and of the dependency-aware SVs, and in the channel used for the *CSRep* approach. In particular, the demand $D_{S,i}(t)$ and the state (working or down) associated to the generic node i are modeled in the template SAN by the following dependency-aware SVs:

- the extended places *Demand_A* (n-sized array of double) and *Up* (n-sized array of short), that are shared among all the replicas, for *SSRep*,
- the extended places *Demand_Aj* (δ^h-sized array of double) and *Up_Aj* (δ^h-sized array of short), that are local to all the replicas, for *CSRep*,
- the double-type extended place *Demand_A* and the place *Up_A*, that replace V^h in (A.1), for *DARep*.

The double-type local extended place *TTR* models the value $T_{S,i}^{threshold} - T_{S,i}^{overc}(t)$, used to set the deterministic time to reboot, represented by the activity *TTRBT*, when the demand overcomes a pre-set threshold. The double-type local

extended place *Te* models the instant of time when the demand overcomes a pre-set threshold, used to evaluate the new value for *TTR* when the demand returns below the threshold.

The activity *TUpdD_A* and the associated cases model respectively the random time to update the demand $D_{S,i}(t)$ and the probability $\mathbb{P}_{\text{incr}}$ (upper case 0), depending on the value of the local place *Trend_A* (if it is 1 the demand trend is increasing). The output gate *DecrD_A* updates the demand each time it decreases and also updates the value of *TTR* to $TTR{->}Mark() - BaseModelClass::LastActionTime + Te{->}Mark()$ (the current value of *TTR* minus the current time plus the time when *TTRBT* has been enabled), if the demand returned below the threshold for both services. The activity *TTRCV*, enabled when the service *A* on the node is working, models the random time to failure of the node. The rate of this activity is marking dependent, being a function of the demand $D_{S,i}(t)$. At completion of *TTRCV*, when a failure occurs, 1 token is added to the place *Rcv* to enable the activity *TRCV* that models the duration of the recovery. The activity *TRBT* models the duration of the reboot. At completion of *TRBT* the reboot ends successfully with probability *c* associated to the lower case 2, when one token is added to the short-type place *NRBT* counting the number of successfully reboots, otherwise the reboot is considered failed, and 1 token is added to the place *Rcv* to enable the recovery action. The recovery is also enabled, selecting case 1, when at completion of *TRBT* the value of *NRBT* is greater than n^{rcv}. The place *React* is used to reactivate the activity *TTRCV* each time the demand $D_{S,i}(t)$ is updated. Finally, with the *CSRep* approach, the places *isUp*, *isDem_A*, *isTransDem_A* and *TransDem_A*, shared among the SAN models shown in Figs. A.5 and A.6(a), are used to set the information that has to be transferred with the channel (for example, *isUp* is equal to 1, if the place *Up_Aj* has been updated).

Appendix B. Acronyms and symbols

AFI Abstract Functional Interface

SAN Stochastic Activity Network

SV State Variable

a_i^k Specific component action, defined in Section 3.1

a^k Generic component action, defined in Section 3.1

${}^{C}_{S}\mathcal{C}\tau(k)$ Compared simulation performance between *CSRep* and *SSRep* with *k* batches, defined in (23)

${}^{D}_{C}\mathcal{C}\tau(k)$ Compared simulation performance between *DARep* and *CSRep* with *k* batches, defined in (24)

${}^{D}_{S}\mathcal{C}\tau(k)$ Compared simulation performance between *DARep* and *SSRep* with *k* batches, defined in (22)

C^g Generic component of the system, defined in (2)

C_i *i*th specific component of the system, defined in (3)

CSRep Channel-Sharing Replication

DARep Dependency-Aware Replication

$\mathcal{D}$ *DARep* operator

δ^h Maximum dependency degree for V^h, defined in Section 5.2

$\hat{\delta}_i^h$ Number of C_i that can access to V_j^h, defined in (1)

δ_i^h Dependency degree of component *i* for V_i^h, defined in (1)

Δ_i^h List of components *j* from which the component *i* can read/write V_j^h, defined in (1)

$\Delta\tau(k)$ Amount of *CPU* time to run *k* batches of the Möbius simulator

Deps() Auxiliary function used in *DARep*

Index() Auxiliary function used in *DARep*

Join *Join* operator, example in (4)

$\mathcal{J}^{shared}$ List of shared SVs used in *DARep*

M^{cs} Template model for the *CSRep* approach, defined in (10)

M^{darep} Template model for the *DARep* approach, defined in (17)

M_i ith specific model representing C_i

M^{ss} Template model for the *SSRep* approach, defined in (6)

M^{sys} Overall system model

M^t Template model to represent a generic component

n Number of system components

n_S Number of virtual machines hosting a server for service S

Rep *Replication* operator, example in (5)

SSRep State-Sharing Replication

τ_{init} Amount of *CPU* time to initialize the Möbius simulator

$\tau(k)$ $\tau_{init} + \Delta\tau(k)$

$\mathcal{T}^h$ Adjacency matrix representing the topology of interactions based on V^h, defined in Section 3.1

V^h Generic template SV, defined in Section 3.1

V_i^h ith instance of the template SV V^h, defined in Section 3.1

References

[1] D.M. Nicol, W.H. Sanders, K.S. Trivedi, Model-based evaluation: from dependability to security, IEEE Trans. Dependable Secure Comput. 1 (1) (2004) 48–65.
[2] S. Baarir, M. Beccuti, D. Cerotti, M.D. Pierro, S. Donatelli, G. Franceschinis, The GreatSPN tool: recent enhancements, ACM SIGMETRICS Perform. Eval. Rev. 36 (4) (2009) 4–9, Spec. Issue on Tools for computer performance modeling and reliability analysis.
[3] D.D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J.M. Doyle, W.H. Sanders, P.G. Webster, The Möbius framework and its implementation, IEEE Trans. Softw. Eng. 28 (10) (2002) 956–969.
[4] C. Hirel, R. Sahner, X. Zang, K. Trivedi, Reliability and performability modeling using SHARPE 2000, in: B.R. Haverkort, H.C. Bohnenkamp, C.U. Smith (Eds.), Computer Performance Evaluation. Modelling Techniques and Tools, TOOLS 2000, in: LNCS, vol. 1786, Springer, Berlin, Heidelberg, 2000, pp. 345–349.
[5] K.S. Trivedi, R. Sahner, SHARPE at the age of twenty two, SIGMETRICS Perform. Eval. Rev. 36 (4) (2009) 52–57.
[6] G. Ciardo, K.S. Trivedi, A decomposition approach for stochastic Petri net models, in: The 4th Int. Workshop on Petri Nets and Performance Models, PNPM 1991, Melbourne, Victoria, Australia, 1991, pp. 74–83.
[7] P. Lollini, A. Bondavalli, F. Di Giandomenico, A decomposition-based modeling framework for complex systems, IEEE Trans. Reliab. 58 (1) (2009) 20–33.
[8] S. Derisavi, P. Kemper, W.H. Sanders, Symbolic state-space exploration and numerical analysis of state-sharing composed models, Linear Algebra Appl. 386 (2004) 137–166, Special Issue on the Conf. on the Numerical Solution of Markov Chains 2003.
[9] P. Buchholz, P. Kemper, Kronecker based matrix representations for large Markov models, in: C. Baier, B. Haverkort, H. Hermanns, J. Katoen, M. Siegle (Eds.), Validation of Stochastic Systems, in: LNCS, vol. 2925, Springer, Berlin, Heidelberg, 2004, pp. 256–295.
[10] L. Brenner, P. Fernandes, A. Sales, T. Webber, A framework to decompose GSPN models, in: G. Ciardo, P. Darondeau (Eds.), Applications and Theory of Petri Nets 2005, in: LNCS, vol. 3536, Springer Verlag, 2005, pp. 128–147.
[11] K.S. Trivedi, A. Bobbio, Reliability and Availability Engineering: Modeling, Analysis, and Applications, Cambridge University Press, 2017.
[12] G. Ciardo, R. Marmorstein, R. Siminiceanu, The saturation algorithm for symbolic state-space exploration, 8 (4), Int. J. Softw. Tools Technol. Transfer (2006) 4–25.
[13] W.H. Sanders, R.S. Freire, Efficient simulation of hierarchical stochastic activity network models, Discrete Event Dyn. Syst. 3 (2) (1993) 271–300.
[14] S. Chiaradonna, P. Lollini, F. Di Giandomenico, On a modeling framework for the analysis of interdependencies in electric power systems, in: 37th Annu. IEEE/IFIP Int. Conf. on Dependable Syst. and Netw., DSN 2007, Edinburgh, UK, 2007, pp. 185–195.
[15] F. Flammini, Critical Infrastructure Security: Assessment, Prevention, Detection, Response, in: Information & Communication Technologies, WIT Press, 2012.
[16] S. Chiaradonna, F.D. Giandomenico, P. Lollini, Definition, implementation and application of a model-based framework for analyzing interdependencies in electric power systems, Int. J. Crit. Infrastruct. Prot. 4 (1) (2011) 24–40.
[17] C. Di Martino, M. Cinque, D. Cotroneo, Automated generation of performance and dependability models for the assessment of wireless sensor networks, IEEE Trans. Comput. 61 (6) (2012) 870–884.
[18] E. Battista, V. Casola, S. Marrone, N. Mazzocca, R. Nardone, V. Vittorini, An integrated lifetime and network quality model of large WSNs, in: 2013 IEEE International Workshop on Measurements and Networking, M&N, 2013, pp. 132–137.
[19] G. Masetti, S. Chiaradonna, F. Di Giandomenico, Model-based simulation in Möbius: an efficient approach targeting loosely interconnected components, in: Computer Performance Engineering: 13th European Workshop, EPEW, Berlin, Germany, 2017, pp. 184–198.
[20] S. Chiaradonna, F. Di Giandomenico, G. Masetti, A stochastic modeling approach for an efficient dependability evaluation of large systems with non-anonymous interconnected components, in: The 28th Int. Symp. on Softw. Reliab. Eng., ISSRE 2017, Toulouse, France, 2017, pp. 46–55.
[21] L. Montecchi, P. Lollini, A. Bondavalli, A template-based methodology for the specification and automated composition of performability models, IEEE Trans. Reliab. (2019) 1–17.

[22] S. Bernardi, S. Marrone, J. Merseguer, R. Nardone, V. Vittorini, Towards a model-driven engineering approach for the assessment of non-functional properties using multi-formalism, Softw. Syst. Model. 18 (3) (2019) 2241–2264.
[23] S. Derisavi, P. Kemper, W.H. Sanders, T. Courtney, The Möbius state-level abstract functional interface, in: T. Field, P.G. Harrison, J. Bradley, U. Harder (Eds.), Computer Performance Evaluation: Modelling Techniques and Tools, Springer, Berlin, Heidelberg, 2002, pp. 31–50.
[24] J. Hillston, A Compositional Approach to Performance Modelling, Cambridge University Press, New York, NY, USA, 1996.
[25] G. Clark, W.H. Sanders, Implementing a stochastic process algebra within the Möbius modeling framework, in: L. de Alfaro, S. Gilmore (Eds.), Process Algebra and Probabilistic Methods. Performance Modelling and Verification, Springer, Berlin, Heidelberg, 2001, pp. 200–215.
[26] E. Ruijters, M. Stoelinga, Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools, Comput. Sci. Rev. 15–16 (2015) 29–62.
[27] M.A. Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, Modelling with Generalized Stochastic Petri Nets, first ed., John Wiley & Sons, Inc., New York, NY, USA, 1994.
[28] W. Sanders, J.F. Meyer, Reduced base model construction methods for Stochastic Activity Networks, IEEE J. Sel. Areas Commun. 9 (1) (1991) 25–36.
[29] A.L. Williamson, Discrete Event Simulation in the Möbius Modeling Framework (Master's thesis), University of Illinois at Urbana-Champaign, 1998.
[30] V.V. Lam, P. Buchholz, W.H. Sanders, A component-level path-based simulation approach for efficient analysis of large Markov models, in: M.E. Kuhl, N.M. Steiger, F.B. Armstrong, J.A. Joines (Eds.), 37th Conf. on Winter Simulation, Orlando, Florida, 2005, pp. 584–590.
[31] K.S. Trivedi, Probability and Statistics with Reliability, Queueing and Computer Science Applications, second ed., John Wiley & Sons, New York, 2002.
[32] G. Masetti, S. Chiaradonna, F. Di Giandomenico, B. Feddersen, W.H. Sanders, An efficient strategy for model composition in the Möbius modeling environment, in: 2018 14th European Dependable Computing Conference, EDCC 2018, 2018, pp. 116–119.
[33] M. Beccuti, G. Franceschinis, Efficient simulation of stochastic well-formed nets through symmetry exploitation, in: 2012 Winter Simulation Conference, WSC, Berlin, Germany, 2012, pp. 1–13.
[34] L. Wall, T. Christiansen, J. Orwant, in: M. Loukides (Ed.), Programming Perl, third ed., O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.