An Efficient Strategy for Model Composition in the Möbius Modeling Environment

Giulio Masetti*†, Silvano Chiaradonna†, Felicita Di Giandomenico†, Brett Feddersen‡, and Willian H. Sanders‡
* *University of Pisa,*
† *ISTI-CNR, Pisa, Italy*
{*giulio.masetti,silvano.chiaradonna,felicita.digiandomenico*}*@isti.cnr.it*
‡ *Information Trust Institute*
University of Illinois, Urbana, Illinois, USA
{*bfeddrsn, whs*}*@illinois.edu*

Abstract—**Möbius is well known as a modeling and evaluation environment for performance and dependability indicators. One of Möbius' key features is the modular and compositional approach to model definition and analysis. In particular, the modeler can define submodels using several formalisms and compose them to form the overall model of the system under analysis. The current algorithm for model composition in Möbius revealed performance issues when large systems are considered (such as in the modeling of realistic segments of energy or transportation infrastructures), due to the chosen data flow scheme. In this paper, a new algorithm for the same composition mechanism is proposed to improve efficiency. A case study is also developed to demonstrate the performance enhancements.**

1. Introduction and related work

Modularity and composability are essential features in model-based analysis due to the high level of complexity of real-world systems. Therefore, it is often preferable to decompose these systems in interacting subsystems, to model each subsystem separately and then to compose such models for obtaining the overall system model, in accordance with the system architecture.

In the context of state-space based and stochastic oriented modeling, well known tools, such as SHARPE [1] and GreatSPN [2], offer built-in capabilities to tackle modularity and composability by defining compositional operators. The most common operators involve one of the following alternative approaches: 1) obtaining the results about the overall model by solving the submodels in isolation and exchanging results as specified by the interactions among submodels described by an import graph [3]; 2) solving directly the overall model defined as a symbolic union of submodels, exploiting a compositional algebra [4].

The Möbius modeling framework [5] follows the latter approach and extends further the modularity principle adopting the object oriented programming paradigm to represent atomic models, composed models, reward structures (performance measures) and model solvers as classes. The class hierarchy describes behavioural aspects. For instance, different modeling formalisms are obtained specializing the base model class where State Variables (SVs) and actions are defined [6]. The architecture of the system model is reflected by the methods call graph. In particular, there are two kinds of models: atomic and composed.

In this paper, the focus is on state-sharing based composition, and in particular on two operators: *Join* and *Rep*. The *Join* combines individual submodels to generate a composed model, based on certain SVs of the individual submodels, called *distinguished* SVs [7], that allow communication among the individual submodels.

Join associates a m-sized array of distinguished SVs with each individual submodel, allowing particular entries of the arrays to be empty. In the joined model, all the SVs corresponding to the same position in each array are merged to form a single SV, i.e., the SVs are shared among the submodels. In addition, each distinguished SV is exposed in read/write mode to other compositional operators, whereas all the undistinguished SV are considered local to each individual submodel. Formally, *Join* is defined as:

$$\mathcal{J}\big(\mathbb{N}_1, \mathbb{V}_1^1, \ldots, \mathbb{V}_m^1; \ldots; \mathbb{N}_n, \mathbb{V}_1^n, \ldots, \mathbb{V}_m^{n-1}\big),$$

where $\mathbb{N}_1$, …, $\mathbb{N}_n$ are the submodels, $\mathbb{V}_j^i$ are the distinguished SVs defined for each submodel $\mathbb{N}_i$, and $\mathbb{V}_j^h = \mathbb{V}_j^k$, for all h and k such that $\mathbb{V}_j^h \neq \emptyset$ and $\mathbb{V}_j^k \neq \emptyset$.

The operator *Rep* is a special case of $\mathcal{J}$ that replicates a model n times, holding the distinguished SVs common to all resulting submodels. It is formally defined as:

$$\mathcal{R}_n\big(\mathbb{N}; \mathbb{V}_1, \ldots, \mathbb{V}_m\big) = \\ = \mathcal{J}\big(\mathbb{N}_1, \mathbb{V}_1^1, \ldots, \mathbb{V}_m^1; \ldots; \mathbb{N}_n, \mathbb{V}_1^n, \ldots, \mathbb{V}_m^n\big).$$

Thus, *Join* and *Rep* impose a directed tree structure over the constructor methods call graph, called *compositional tree* [7]. An example of compositional tree is in Fig. 1.

Möbius offers a set of model solvers, all having in common the execution of two phases: initialization and analysis. Here the focus is mainly on the initialization phase because the current version follows an algorithm that presents inefficiencies.

The aim of this paper is to investigate the causes of inefficiencies during solver initialization and propose a new algorithm to overcome them.

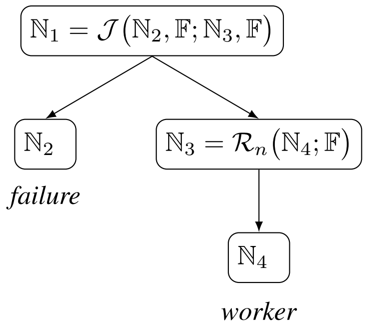
$\mathbb{N}_1 = \mathcal{J}(\mathbb{N}_2, \mathbb{F}; \mathbb{N}_3, \mathbb{F})$
$\mathbb{N}_2$
$\mathbb{N}_3 = \mathcal{R}_n(\mathbb{N}_4; \mathbb{F})$
failure
$\mathbb{N}_4$
worker

Figure 1. Example of compositional tree: nodes are model classes, an edge from $\mathbb{N}_h$ to $\mathbb{N}_k$ indicates that within $\mathbb{N}_h$ the constructor method of $\mathbb{N}_k$ is called. Leaf nodes are atomic models, the root is the overall system model.

The rest of the paper is structured as follows. In Section 2, the current implementation is discussed. In Section 3, our proposal for a new implementation is introduced. Section 4 briefly presents the case study adopted in Section 5 to compare current and proposed composition algorithm in terms of performance. Conclusions and future work are presented in Section 6.

2. Current algorithms for *Join* and *Rep*

As already mentioned, in Möbius: designing a model means defining a hierarchy of classes that describe atomic and composed models; constructing a model means instantiating an object whose class is the class of the root in the compositional tree. This object is then passed to the selected solver. An example of model design is the compositional tree depicted in Fig. 1 and listed as

$$\text{system} = \mathcal{J}\big(\textit{failure}, \mathbb{F}; \mathcal{R}_n\big(\textit{worker}; \mathbb{F}\big), \mathbb{F}\big).$$

It is the reliability model of a system comprising n identical workers that perform computations in parallel. The atomic model *worker* represents the failure of a worker. If a *worker* fails, then the distinguishable SV called $\mathbb{F}$ is increased by one. The atomic model *failure* models the system failure mechanism, based on the value of $\mathbb{F}$. The compositional tree has 4 nodes: two atomic model classes, $\mathbb{N}_2$ for failure and $\mathbb{N}_4$ for worker; a composed model $\mathbb{N}_3 = \mathcal{R}_n\big(\mathbb{N}_4, \mathbb{F}\big)$, where $\mathbb{F}$ is shared among all worker replicas; and the root model class $\mathbb{N}_1 = \mathcal{J}\big(\mathbb{N}_2, \mathbb{F}; \mathbb{N}_3, \mathbb{F}\big)$ for system, where $\mathbb{F}$ is shared among $\mathbb{N}_2$ and $\mathbb{N}_3$. The constructor of class $\mathbb{N}_1$ calls the constructors of classes $\mathbb{N}_2$, $\mathbb{N}_3$ and base SV class; the constructor of $\mathbb{N}_3$ calls the constructors of classes $\mathbb{N}_4$ and base SV class; the constructors of $\mathbb{N}_2$ and $\mathbb{N}_4$ call the constructor of class base SV class. The sharing can take place among objects of base SV class. An object N_1 of class $\mathbb{N}_1$ is created and passed to the solver for performing the analysis.

In Möbius 2.5 (the current version), model classes are not aware of the compositional tree between the root and them. In particular, each node constructor does not receive information from its parent about already created SVs, thus multiple copies of the same distinguished SV are created.

Calling $A.B$ the object B of class $\mathbb{B}$ inside the object A of class $\mathbb{A}$, the constructor method of *Join* performs the following steps:

$\mathcal{J}_{\text{Current}}$ 1) creates an object for each distinguished SV, in the example $N_1.F$,

$\mathcal{J}_{\text{Current}}$ 2) creates an object for each of its children, in the example $N_1.N_2$ and $N_1.N_3$,

$\mathcal{J}_{\text{Current}}$ 3) shares its distinguished SVs with the corresponding SVs of its children objects, in the example $N_1.F$ is shared with $N_1.N_2.F$ and $N_1.N_3.F$.

In this way the compositional tree is traversed recursively from root to leaves. The application of $\mathcal{J}_{\text{Current}}$ to the particular case of a *Join* generated by the *Rep* operator produces:

$\mathcal{R}_{\text{Current}}$ 1) creates an object for each distinguished SV, in the example $N_3.F$,

$\mathcal{R}_{\text{Current}}$ 2) creates n objects (replicas) of the class specified by its children, in the example $N_3.N_4^1$, ..., $N_3.N_4^n$,

$\mathcal{R}_{\text{Current}}$ 3) shares its distinguished SVs with the corresponding SVs of its children objects, in the example $N_3.F$ is shared with $N_3.N_4^1.F$, ..., $N_3.N_4^n.F$.

For both *Join* and *Rep*, multiple copies of the same distinguished SV are created in 1) and 2). Then, in 3), all these copies have to be scanned and reduced to a unique object. A detailed description of the sharing mechanism is out of this paper's scope. The complete characterization is in [7]; here it is sufficient to notice that it involves not only the distinguished SVs, but also the actions that are affected by, or can have impact on, the distinguished SVs. For the *Join* operator each sharing has a time complexity of $\mathcal{O}(n^2)$, where n is the number of replicas.

Models with a complex compositional tree, i.e., one or more *Rep* operators with many replicas, do not pose performance issues as long as only a few distinguished SVs are involved. However, when system components have complex interactions, as it increasingly occurs in modern systems, a large number of distinguished SVs is needed. Experiments have shown that performance of the solver initialization phase quickly degrades if the number of distinguished SVs in the overall model is $\mathcal{O}(n)$ or higher, due to the too expensive cost of the sharing mechanism.

The aim of Section 3 is to present a new algorithm to drastically reduce the computational cost per share.

3. New algorithm proposal

The modeler follows an ordering when designing the model: first, the atomic models are defined independently from the others, then atomic models are composed to form the overall system model. So the design ordering is from leaves to root. Unfortunately, the objects construction in the current version of Möbius follows the same ordering and, as discussed in Section 2, this can lead to long startup times.

The key idea is to revert the design ordering during construction, i.e., calling constructor methods from root to leaves: 1) introduce a data structure that keeps track of the distinguished SVs already created in the compositional tree; 2) redefine all the constructors, for both atomic and composed submodels, to search in the data structure for the presence of a SV before creating it. The data structure of choice is an unordered map $\mathcal{M}$ whose keys are SV names and values are SV object references. In the example shown in Fig. 1, the map has only one pair (key, value), namely $\mathcal{M} = \{(\mathbb{F}, N_1.F)\}$.

In our proposal, labeled *New*, the empty map $\mathcal{M} = \{\}$ is created by the solver and passed to the constructor of the compositional tree root, that can be a *Join* or a *Rep*. Notice that *Rep* is a special case of *Join*, thus the new algorithm equally impacts on both *Join* and *Rep*.

Each node constructor, either atomic or composed model, performs the following steps:

New 1) receives $\mathcal{M}$,
New 2) checks if distinguished SVs exist within $\mathcal{M}$:
- if yes, then it copies the object references instead of creating new ones,
- else, it creates the new objects and inserts the object references into $\mathcal{M}$,

New 3) calls the constructor of children nodes,
New 4) if the node is a composed model, then removes distinguished SV references from $\mathcal{M}$.

Notice that *New* 4) is needed because two leaves of the compositional tree can have SVs with the same name that are local to different models. The sharing in our new algorithm is no more performed after node objects creation, but is part of the objects creation itself.

The time complexity of each sharing is bounded by the worst complexity among searching, inserting and removing couples from the map $\mathcal{M}$. In particular, the map $\mathcal{M}$ has been implemented using the unordered map of the C++ standard library [8], where searching, inserting and deleting are preformed in $\mathcal{O}(1)$. Thus, compared with the $\mathcal{O}(n^2)$ shown by the *Current* algorithm, the *New* algorithm performs significantly better. Experiments in Section 5quantitatively assess the improvements on the case study.

4. Case study

Similarly to the logical structure of the system presented in [9], consider n working stations, called *worker*$_1$, ..., *worker*$_n$, dedicated to performing the same task in parallel. At every time instant, each station can be either working or failed and its failure rate is a function of the workload assigned to it. The failure of a station implies a reconfiguration of the workload assigned to the other stations to continue accomplishing the tasks of the failed station. Just before failing, a station redirects its tasks to one or more of the other stations it is connected with. Connections, and then neighbouring relations, follow a predefined oriented graph. The system model is obtained by non-anonymous replication, that means:

- a generic worker is modeled and replicated n times,
- when the system model is constructed, replicas are indistinguishable,
- a sophisticated mechanism promotes each replica to become self-aware of its index and then interact with other replicas according to the connections graph.

Detailed logical structure and the complete model, implemented using the Stochastic Activity Network (SAN) formalism, where SVs correspond to places in Petri Net dialects, has been published in [9]. Here, in order to simplify the notation, the model compositional tree is described as

$$\mathit{system} = \mathcal{R}_n\big(\mathit{worker}, \mathbb{S}_1, \ldots, \mathbb{S}_n\big),$$

where $\mathbb{S}_j$ are distinguished SVs that can be read and written by *worker*$_i$ if the connections graph contains the edge (i, j). The model is designed to tackle each possible connection graph, thus all the $\mathbb{S}_j$ are shared among all the *worker* replicas. The compositional tree has 2 nodes: one atomic model class $\mathbb{N}_2$ for *worker*, and one composed model class $\mathbb{N}_1$ for the root. In the current version of Möbius the following steps are performed:

Current 1) for all $j = 1, \ldots, n$ creates $N_1.S_j$,
Current 2) for all $i = 1, \ldots, n$ creates $N_1.N_2^i$, that in turn, for all $j = 1, \ldots, n$, creates $N_1.N_2^i.S_j$,
Current 3) for all $i, j = 1, \ldots, n$ shares $N_1.S_j$ with $N_1.N_2^i.S_j$.

The time complexity of Current 1) and Current 2) is $\mathcal{O}(n^2)$, but the complexity of Current 3) is $\mathcal{O}(n^4)$ because there are n distinguished SVs, shared among all the n replicas (therefore, n^2 shares, each of complexity $\mathcal{O}(n^2)$).

Instead, our new algorithm performs the following steps:

New 1) receives $\mathcal{M} = \{\}$,
New 2) for all $j = 1, \ldots, n$ creates $N_1.S_j$ and inserts $N_1.S_j$ inside $\mathcal{M}$, i.e., $\mathcal{M} = \mathcal{M} \cup \{(S_j, N_1.S_j)\}$,
New 3) for all $i = 1, \ldots, n$ creates $N_1.N_2^i$, that in turn, for all $j = 1, \ldots, n$, searches S_j in $\mathcal{M}$ and copies $N_1.S_j$ into $N_1.N_2^i.S_j$,
New 4) removes nothing.

The time complexity of *New* 3) is $\mathcal{O}(n^2)$.

5. Numerical results

The adopted case study has been modeled to assess the Mean Time to Failure of *worker*$_i$ for $i = 1, \ldots, n$, through the Möbius simulator [10] (running 1000 batches). To compare the two algorithms, the CPU time they need for the initialization phase is computed, as reported in Table 1. To compare performance, the CPU time of initialization are shown in Table 1. All the experiments have been performed on a computer with an Intel i7-4710MQ CPU running at 2.50 GHz and with 16 GB of RAM clocked at 1333 MHz. Within the Möbius modeling environment, models are designed as a hierarchy of C++ classes. Thus the theoretical prediction of the complexity for the simulator initialization phase is $\mathcal{O}(n^4)$ for the current version of Möbius and $\mathcal{O}(n^2)$ for our new algorithm. The predicted time complexities are confirmed by experimental results, as reported in Table 1 and clearly visible in Figure 2. In addition, notice that the sharing in the current version of Möbius produces a complex set of

TABLE 1. CPU TIMES FOR INITIALIZATION AS n INCREASES.

n	$t_{\text{init\_Current}}$ (s)	$t_{\text{init\_New}}$ (s)
100	0.22	$4.3 \cdot 10^{-2}$
200	1.78	$9.2 \cdot 10^{-2}$
300	6.99	0.20
400	18.90	0.36
500	41.69	0.59
600	82.56	0.91
700	151.07	1.38
800	254.02	1.99
900	414.83	2.69
1000	631.35	3.60

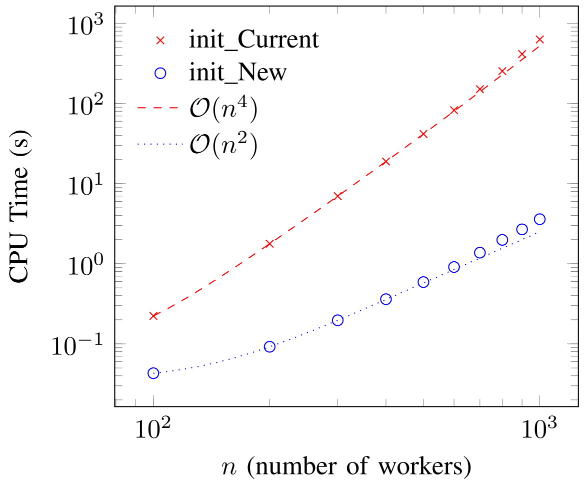

Figure 2. Comparison of time complexity for the solver initialization phase in the current version of Möbius and for our new algorithm as the number n of workers increases.

data structures that have an impact on batches performance, whereas in the new algorithm no additional data structure is needed. Thus, also batches performance is improved, as depicted in Table 2.

TABLE 2. CPU TIMES FOR 1000 BATCHES AS n INCREASES.

n	$t_{\text{batches\_Current}}$ (s)	$t_{\text{batches\_New}}$ (s)
100	9.99	8.78
200	43.47	35.83
300	121.34	84.15
400	246.35	155.98
500	505.52	277.18
600	783.10	436.57
700	1142.66	735.40
800	1561.30	959.69
900	2183.85	1292.19
1000	2796.18	1676.77

6. Conclusions and future work

A new algorithm for Möbius *Join* and *Rep* operators has been presented to enhance performance. The analysis of time complexity, confirmed by experiments conducted on a case study requesting non-anonymous replication, has shown a great improvement of solvers initialization phase and also during the model solution.

Research extensions are foreseen in two main directions: 1) investigation on adopting the same approach in other modeling tools, developed in accordance with the object-oriented paradigm and offering composition by state sharing. Given the generality of the idea at the basis of our solution, we believe it is possible; 2) study of a new native operator in Möbius to further boost performance with respect to the $\mathcal{O}(n^2)$, which is the best achievable with the proposed technique. Methods to improves performance in simulation-based modeling have been proposed in [9], [11] for non-anonymous replication, but not as a native Möbius operator, which would be instead a promising direction to explore.

References

[1] R. A. Sahner, K. S. Trivedi, and A. Puliafito, *Performance and Reliability Analysis of Computer Systems: An Example-based Approach Using the SHARPE Software Package*. Norwell, MA, USA: Kluwer Academic Publishers, 1996.

[2] E. G. Amparore, G. Balbo, M. Beccuti, S. Donatelli, and G. Franceschinis, *30 Years of GreatSPN*. Cham: Springer International Publishing, 2016, pp. 227–254.

[3] G. Ciardo and K. S. Trivedi, "A decomposition approach for stochastic reward net models," *Perform. Eval.*, vol. 18, no. 1, pp. 37–59, 1993.

[4] S. Bernardi, S. Donatelli, and A. Horvath, "Compositionality in the GreatSPN tool and its application to the modelling of industrial applications," in *University of Aarhus (Denmark*, 2000, pp. 127–146.

[5] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster, "The Möbius framework and its implementation," *IEEE Trans. on Softw. Eng.*, vol. 28, no. 10, pp. 956–969, 2002.

[6] S. Derisavi, P. Kemper, W. H. Sanders, and T. Courtney, "The Möbius state-level abstract functional interface," *Perform. Eval.*, vol. 54, no. 2, pp. 105–128, Oct. 2003.

[7] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for stochastic activity networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 1, pp. 25–36, 1991.

[8] N. M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, 2nd ed. Addison-Wesley Professional, 2012.

[9] G. Masetti, S. Chiaradonna, and F. Di Giandomenico, "Model-based simulation in Mobius: an efficient approach targeting loosely interconnected components," in *Computer Performance Engineering: 13th European Workshop (EPEW)*, Berlin, Germany, 2017.

[10] A. L. Williamson, "Discrete event simulation in the Möbius modeling framework," U. of Illinois at Urbana-Champaign, Tech. Rep., 1998.

[11] S. Chiaradonna, F. Di Giandomenico, and 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) - IEEE*, Toulouse, France, Oct. 2017, pp. 46–55.