An Eclipse-Based Editor for SAN Templates *

Leonardo Montecchi[1][0000−0002−7603−9695], Paolo Lollini[2,3][0000−0002−2364−2538], Federico Moncini[3][0000−0003−1696−1765], and Kenneth Keefe[4][0000−0002−2690−5268]

[1] Universidade Estadual de Campinas, Campinas, SP, Brazil
leonardo@ic.unicamp.br
[2] Consorzio Interuniversitario Nazionale per l'Informatica (CINI), Firenze, Italy
[3] University of Firenze, Firenze, Italy
lollini@unifi.it, federico.moncini@stud.unifi.it
[4] University of Illinois at Urbana Champaign, Urbana, Illinois, USA
kjkeefe@illinois.edu

Abstract. Mathematical models are an effective tool for studying the properties of complex systems. Constructing such models is a challenging task that often uses repeated patterns or templates. The Template Models Description Language (TMDL) has been developed to clearly define model templates that are used to generate model instances from the template specification. This paper describes the tool support that is being developed for applying the TDML approach with Stochastic Activity Networks (SANs) models. In particular, this paper details a graphical editor for SAN templates, which assists users in creating template-level models based on SANs. From these specifications, it will be possible to generate by model-transformation the subsequent instance-level models, which can be studied by simulation or analytical tools.

Keywords: stochastic activity networks · templates · Sirius · metamodel · graphical editor.

1 Introduction

Model-based evaluation [1] has been extensively used to estimate performance and reliability metrics of computer systems. Constructing and maintaining models for large-scale, evolving systems is a challenging task. In our recent work [2], we defined an approach for reusing the specification of performability models, in particular, Stochastic Petri Net (SPN) models [3]. The approach is based on the concept of *model templates*, which use well-defined interfaces to interact with connected segments of the model. The interfaces and composition rules are specified using our novel, domain-specific language, Template Models Description Language (TMDL). A model template is essentially a parameterized

* This work has received funding from the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement No 823788. This work has received funding from the São Paulo Research Foundation (FAPESP) with grant #2019/02144-6.

abstracted version of a model in a specific formalism. From a template, concrete instances can be automatically derived by specifying values for its parameters.

In [2] we defined the overall idea of the framework, formalized its definition, and introduced the TMDL language. The framework was designed to be independent of a specific formalism, and it assumes the existence of 1) a *template-level formalism* 2) an *instance-level formalism*, and 3) a *concretize function*, to generate an instance-level model from a template-level model. Then, in [4] we completed the formalization by defining Stochastic Activity Network Templates (SAN-T), a template-level formalism based on Stochastic Activity Networks (SANs) [5]. In the same document we also defined the associated *concretize* function, thus enabling the application of the TMDL approach using SANs as instance-level formalism.

However, to be of practical use, appropriate tool support must be developed. In this paper, we introduce an Eclipse-based graphical editor for SAN-T models, the template-level models in the TMDL framework. Using SAN-T models, instance-level SAN models can be generated efficiently and accurately. The generated SAN models can be studied by modeling and simulation tools, such as Möbius [6]. The rest of the paper is organized as follows. In Section 2 we recall in more details the TMDL framework and the SAN-T formalism. In Section 3 we discuss the architecture of the overall tool, and detail the two main components of the editor. In Section 4 we show a simple example of application, and in Section 5 we conclude the paper with an overview on the planned future work.

2 Background

Before focusing on the new tool architecture, an overview of the TMDL framework and the SAN-T formalism is provided.

2.1 The TMDL Framework

Figure 1 provides an overview of the three major steps utilized by the TMDL framework. In Step #1, a *library* of reusable *model templates* is created by an

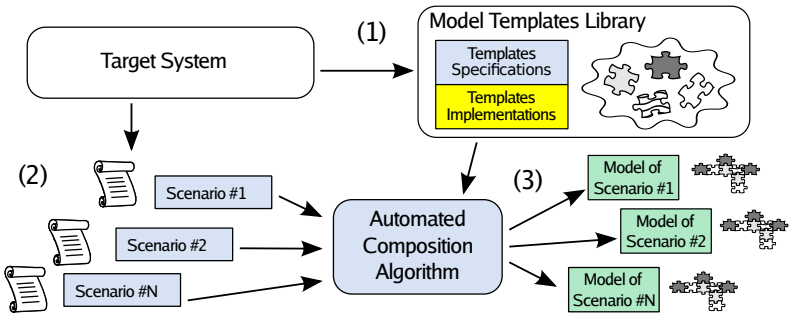

Fig. 1. Workflow of the TMDL framework for the automated composition of template performability models. Figure adapted from [2].

expert. In Step #2, the different system configurations that should be analyzed are defined in terms of "scenarios." Scenarios are composed of *model variants*, that is, a selection of model templates with their parameter value assignment. In Step #3, the *model instances* are automatically created and assembled, thus generating the complete system model for each scenario. What makes the model templates reusable is that they have well-defined *interfaces* and *parameters*. Interfaces specify how they can be connected to other templates, while parameters make it possible to derive different concrete models from the same template.

A model template has a *specification* and an *implementation*. The specification of a template is provided with TMDL. The implementation of an atomic template is given using a *template-level formalism*, that is, a modeling formalism that defines partially specified models in the concrete formalism of choice. By "partially specified", we mean that some aspects of the structure and behavior of the model are controlled by parameters, e.g., the number of cases of an activity. Conversely, the *instance-level formalism* is the modeling formalism actually used for the analysis (e.g., SANs). The models generated in Step #3 conform to the instance-level formalism.

In [4] we provided the definition of a template-level formalism based on SANs, that we call *Stochastic Activity Network Templates* (SAN-T).

2.2 Stochastic Activity Network Templates

Stochastic Activity Networks (SANs) are formal models that represent stochastic behavior, in general, of a complex system [5]. A SAN is defined as a tuple: $SAN = (P, A, I, O, \gamma, \tau, \iota, o, \mu_0, C, F, G)$, where P is a finite set of *places*; A is a finite set of *activities*; I is a finite set of *input gates*; and O is a finite set of *output gates*. The function $\gamma\colon A \to \mathbb{N}^+$ specifies the number of *cases* for each activity, that is, the number of possible choices upon execution of that activity. τ specifies the type of each activity; ι maps input gates to activities and o maps output gates to cases of activities. Places can hold *tokens*; the number of tokens in each place gives the state of the network, called its *marking*.

The behavior of a SAN is determined by input gates and output gates. An *input gate* contains a predicate on the marking of connected places (input predicate), and an input function that alters the marking. An input gate holds in a certain marking if its input predicate holds. An *output gate* contains only the output function, which alters the marking of the connected places. Input arcs and output arcs are special cases of input and output gates that add/remove one token to the connected place. When an activity is enabled it can *fire* (instantaneous activities have priority); when an activity fires, one of its cases is selected. The new marking is obtained by computing the functions of all the input and output gates connected to the activity. The stochastic behavior is given by three functions that are associated to each activity a: function $C_a \in C$ specifies the probability distribution of its cases; $F_a \in F$ specifies the probability distribution of the firing delay; and $G_a \in G$ describes its reactivation markings [5].

In [4] we introduced the new SAN-T formalism, as a template-level formalism [2] based on SANs. The idea is to leave some elements of the SAN model

unspecified, and to make them depend on parameter values. This is different from what is done for example in the Möbius tool [6], where global variables can be used to set initial marking values and distribution parameters. In SAN-T models, parameters can also affect some aspects of the model structure, like the number of cases of a transition or the number of places in the model.

Formally, a *Stochastic Activity Network Template* (SAN-T) is also a tuple:

$$SAN\text{-}T = (\Delta, \tilde{P}, \tilde{A}, \tilde{I}, \tilde{O}, \tilde{\gamma}, \tilde{\tau}, \tilde{\iota}, \tilde{o}, \tilde{\mu}_0, \tilde{C}, \tilde{F}, \tilde{G}), \tag{1}$$

where Δ is a set of parameters, and elements marked with a tilde accent, $\tilde{\cdot}$, are modified versions of SANs elements, reformulated to take parameters into account. The main differences are summarized in the following.

The set Δ is the set of *parameters* of the template, which may have a type. We denote with $\texttt{TERM}_t$ the set of all the possible *terms* of type t, that is, all the possible combinations of parameters and operators that are of type t. For example $\texttt{TERM}_{\texttt{Int}}$ is the set of all terms of integer type. $\tilde{P}$ is a finite set of *place templates*. A place template is a pair (τ, k), where τ is the name of the place, and $k \in \texttt{TERM}_{\texttt{Set\{Int\}}}$ is its multiplicity. When values are assigned to parameters and the instance-level model is derived, the place template is expanded to a set of SAN places. The concept of marking has also been extended. The idea is to anticipate that the place template will be mapped to a set of places, and thus allow the marking for each of them to be specified using an index. Given a set of place templates $\tilde{S} \subseteq \tilde{P}$, a *marking template* of $\tilde{S}$ is a mapping $\tilde{\mu}\colon \tilde{S} \times \mathbb{N} \to \mathbb{N}$. For example, $\tilde{\mu}(\tilde{p}, 2) = 10$ means that the place generated from $\tilde{p}$ having index 2 contains 10 tokens.

All the other elements of a SAN have been adapted to depend on parameters. In particular, cases of *activity templates* also depend on parameters; function $\tilde{\gamma}\colon \tilde{A} \to \texttt{TERM}_{\texttt{Int}}$ specifies the number of cases for each activity template. An *output gate template* is connected directly to an activity template, as opposed to normal output gates that are connected to activity cases. When a regular SAN is generated from the template, the output gate template will be expanded to multiple concrete output gates.

3 Tool Architecture

3.1 Overview

An overview of the tool that will implement the TMDL framework is presented in Figure 2, showing its main components and their dependencies. Colors indicate the role of each component with respect to the workflow of Figure 1. The whole tool is based on the ecosystem provided by the Eclipse Modeling Framework (EMF)[5], and it is composed of a *formalism-independent* layer and a *formalism-specific* layer. The project is available as a public GitHub repository [7].

The *formalism-independent* layer implements the part of the framework that is not tied to a specific instance-level formalism. This is basically the TMDL

[5] `https://www.eclipse.org/modeling/emf/`

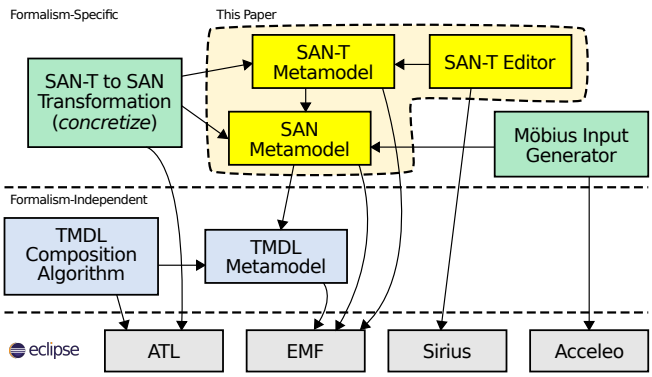

Fig. 2. Main components of the TMDL Framework for SANs, and their dependencies. An arrow from A to B means that component A uses component B. Dependencies between Eclipse components are not shown.

metamodel and the associated composition algorithm introduced in [2]. The *formalism-specific* layer includes the components that support a specific instance-level formalism, in this case SANs. The core of this layer are the metamodels of the instance-level and template-level formalisms. As shown in Figure 2, they depend on the TMDL metamodel; this dependence is limited to the root elements (i.e., the *SAN* and *SANT* metaclasses), which need to extend specific classes to connect to the framework.

The focus of this paper are the components in the dashed region (in yellow), which includes the two metamodels and the graphical editor for SAN-T models. The other components are based on these two metamodels: the *concretize* function that transforms SAN-T models into SAN models will be realized as a model-to-model transformation; instead, a model-to-text transformation will be developed to generate the concrete input for the Möbius analysis tool, which uses an XML-based format.

3.2 SAN and SAN-T Metamodels

To the best of our knowledge there was no EMF-based metamodel that supports all the concepts of the SAN formalism, and we thus developed one as part of this project. We based on the formal definition of SANs [5], but also on their practical implementation provided by Möbius [6]. In fact, Möbius includes some variations to the original definition; for example, it supports extended places, which may hold values of other datatypes, instead of natural numbers only. The complete SAN metamodel contains 57 metaclasses and 4 packages: *Core*, *Types*, *Expressions*, and *Distributions*.

The SAN-T metamodel is organized in 4 packages: *Core*, *Places*, *Cases*, and *Gates*, and it also reuses some elements from the SAN metamodel. Figure 3 depicts the *Core* package, which defines the main elements of a SAN-T model. Figure 4 depicts the *Places* package, which defines elements used in place templates to specify their multiplicity and initial marking. A *PlaceTemplate* contains

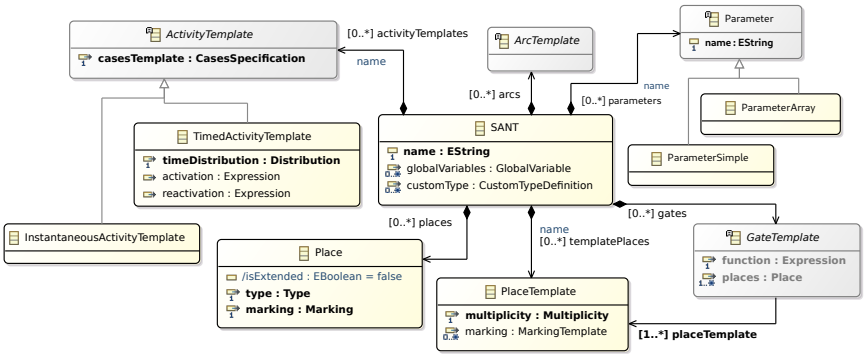

Fig. 3. SAN-T metamodel — *Core*.

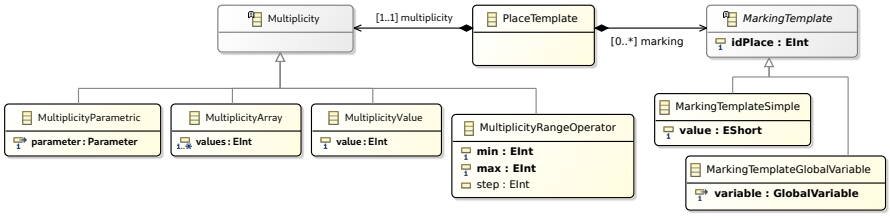

Fig. 4. SAN-T metamodel — *Places*.

a *MarkingTemplate* and a *Multiplicity*. The metamodel contains classes to specify the multiplicity in different ways: as a constant value (*MultiplicityValue*), as a parameter (*MultiplicityParametric*), as an array (*MultiplicityArray*), and as a range of values (*MultiplicityRangeOperator*). The complete SAN-T metamodel contains 22 metaclasses, in addition to those reused from the SAN metamodel.

3.3 Graphical Editor

Based on the metamodels introduced above, we used the Sirius tool to create a graphical editor for SAN-T models. Sirius is an Eclipse project that permits to easily create customized graphical editors. It adopts a declarative approach, in which the developer specifies which element of a certain metamodel should be represented and how, and the framework takes care of the actual realization of the feature and of its integration within Eclipse. Thanks to these facilities, our editor supports most of the features expected from a graphical editor: a tool palette, a property page, synchronization between model and its graphical representation, copy and paste, rearrangement of connections, etc. (Figure 5).

As a positive side effect of the fact that the SAN-T metamodel reuses elements of the SAN metamodel, our editor can also be used to specify regular (i.e., instance-level) SAN models. However, these models will be stored as XMI files, and thus they cannot be directly used as input to Möbius at this time.

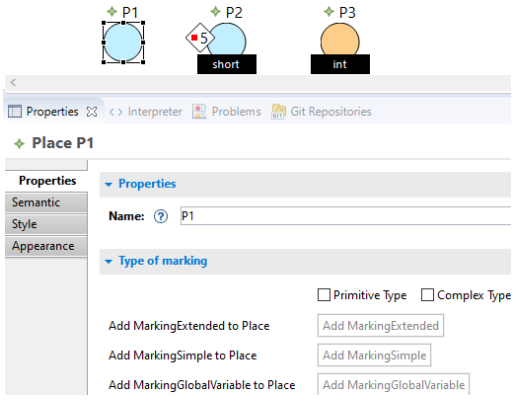

Fig. 5. Editing properties of a place using our editor.

4 Application Example

In this section we provide a simple example of application of our editor. We use the editor to specify the same example that was introduced in [4]; more specifically, in Figure 3(c) of that document. The context is that of a mobile network, in which different services and different classes of users are available. Different users have similar behavior, but may request different network services. The variability is both in the amount of services they have access, and also on which ones. Each service is identified by a number.

Figure 6 shows the *User* SAN-T model, specified with our editor. It includes three normal places (*Idle*, *Failed*, and *Dropped*), and a place template (*Req*). It also has two normal instantaneous activities (*Fail* and *Drop*), and one timed activity template (*Request*). The part of the model that remains unspecified (i.e., the "template" part) is the one determining which services are requested by the user, and with which probability. This aspect of the model is determined by parameter S and by parameter P. It should be noted that the connections involving template elements are highlighted with a different color in the editor (green), to facilitate their identification.

The semantic of the model is the following. The user is initially in idle state (place *Idle* contains one token). After a certain amount of time, given by the distribution associated with activity *Request*, the user requests a network service.

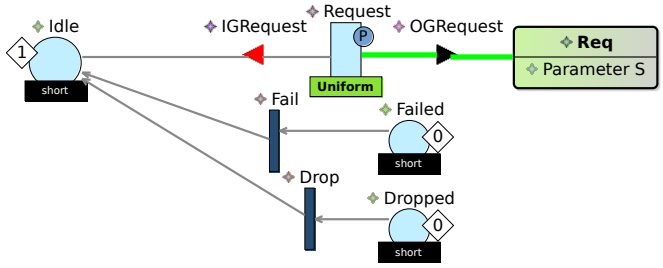

Fig. 6. Example of a SAN-T model, specified with our editor. *Request*, *OGRequest*, and *Req* are template-level entities, and thus highlighted in green.

The identifiers of the services he or she may request are given by the value(s) of parameter S, which determines how many places named *ReqX* will appear in the generated instance-level SAN model (*MultiplicityParametric* element). The number of cases of the activity *Request* and their probabilities are given by the P parameter (*CaseSpecificationProbabilityArray*). The output gate template *OGRequest* determines that if case k is selected, then a token should be added to place Req_k. When a token is added to place *Failed* or *Dropped*, the corresponding activity fires, and the user goes back to the idle state.

5 Concluding Remarks

In this paper we have introduced a graphical editor for SAN-T models, developed on top of the modeling facilities provided by Eclipse. The editor is part of a bigger project that aims to provide concrete tool support for applying the TMDL approach with SANs. As future work, we are working on completing the tool support for an end-to-end application of the TMDL approach, implementing the components highlighted in green in Figure 2. It should be noted that most of them will rely on the SAN and SAN-T metamodels that we developed for realizing this editor.

In particular, as the next immediate steps we plan to implement the model-to-model transformation from SAN-Ts to SANs, and on the model-to-text transformation to generate the input for the Möbius framework. These two components will finally enable the evaluation of performability metrics based on SAN-T models specified with our editor.

References

1. D. M. Nicol, W. H. Sanders, and K. S. Trivedi, "Model-based evaluation: from dependability to security," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 48–65, 2004.
2. L. Montecchi, P. Lollini, and A. Bondavalli, "A template-based methodology for the specification and automated composition of performability models," *IEEE Transactions on Reliability*, vol. 69, pp. 293–309, 2020.
3. G. Ciardo, R. German, and C. Lindemann, "A characterization of the stochastic process underlying a stochastic petri net," *IEEE Transactions on Software Engineering*, vol. 20, no. 7, pp. 506–515, 1994.
4. L. Montecchi, P. Lollini, and A. Bondavalli, "A Formal Definition of Stochastic Activity Networks Templates," arXiv:2006.09291, June 2020, https://arxiv.org/abs/2006.09291.
5. W. Sanders and J. Meyer, "Stochastic activity networks: formal definitions and concepts," in *Lectures on formal methods and performance analysis*, ser. LNCS. Springer, 2002, vol. 2090, pp. 315–343.
6. G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster, "The Möbius modeling tool," in *Proc. of the 9th Int. Workshop on Petri Nets and Perf. Models*, Sept. 2001, pp. 241–250.
7. "TMDL Framework," https://github.com/montex/TMDL-Framework, accessed: Wednesday 8$^{\text{th}}$ July, 2020.