A template-based methodology for the specification and automated composition of performability models

A Template-Based Methodology for the Specification and Automated Composition of Performability Models

Leonardo Montecchi, Paolo Lollini, and Andrea Bondavalli, *Member, IEEE*

***Abstract*—Dependability and performance analysis of modern systems is facing great challenges: their scale is growing, they are becoming massively distributed, interconnected, and evolving. Such complexity makes model-based assessment a difficult and time-consuming task. For the evaluation of large systems, reusable submodels are typically adopted as an effective way to address the complexity and to improve the maintainability of models. When using state-based models, a common approach is to define libraries of generic submodels, and then compose concrete instances by state sharing, following predefined "patterns" that depend on the class of systems being modeled. However, such composition patterns are rarely formalized, or not even documented at all. We address this problem using a model-driven approach, which combines a language to specify reusable submodels and composition patterns, and an automated composition algorithm. Clearly defining libraries of reusable submodels, together with patterns for their composition, allows complex models to be automatically assembled, based on a high-level description of the scenario to be evaluated. This paper provides a solution to this problem focusing on: i) formally defining the concept of model templates, ii) defining a specification language for model templates, iii) defining an automated instantiation and composition algorithm, and iv) applying the approach to a case study of a large-scale distributed system.**

***Index Terms*—modularity, model-based evaluation, state-based, Stochastic Activity Networks, performability, template models, composition, model-driven engineering.**

I. Introduction

Model-based evaluation [1] plays a key role in dependability [2] and performability [3] evaluation of systems. Modeling allows the system to be analyzed at different levels of abstraction, it can be used to perform sensitivity analysis, to identify problems in the design, to guide experimental activities, and to provide answers to "what-if" questions, all without actually exercising the real system. For this reason, modeling and simulation are widely used in the assessment of high-integrity systems and infrastructures, for which faults and attacks can potentially lead to catastrophic consequences.

While ad-hoc simulators are used in some domains, (e.g., see [4]), state-based formalisms like Stochastic Petri Nets (SPNs) and their extensions [5] are widely used to assess non-functional properties across different domains. Such formalisms have several key advantages: they provide a convenient graphical notation, they support different abstraction levels, they enable modular modeling via state sharing (i.e., superposition of state variables), and they are well-suited for the representation of random events (e.g., component failures). Moreover, due to their generality, such formalisms can be used in different domains, and for the analysis of different kinds of system properties.

Nowadays, the analysis of modern systems is facing great challenges: their scale is growing, they are becoming massively distributed, interconnected, and evolving. The high number of components, their interactions, and rapidly-changing system configurations represent notable challenges for model-based evaluation. A transition towards the Systems-of-Systems paradigm [6], [7] is occurring: new services emerge by the aggregation of preexisting, independent, constituent systems, whose internals may not be precisely known.

A key principle in addressing the complexity in the specification of analysis models is *modularization*. When using approaches based on SPNs, reusable submodels addressing different concerns are typically defined and then composed by state sharing, following predefined "patterns" based on the scenario to be analyzed. The reusability and maintainability of the obtained analysis model is therefore improved: submodels can be modified in isolation from the rest of the model, they can be substituted with more refined implementations, they can be rearranged based on modifications in the system configuration.

In practice, "libraries" of reusable models are defined, specific to a certain system or class of systems. However, while models in those libraries can be precisely defined using well-established formalisms (e.g., SPNs), means to specify customized patterns for their instantiation and composition are limited, and in practice such patterns are often defined only informally. As a result i) model libraries are difficult to be shared and reused, and ii) composite models for different scenarios must be assembled by hand by people who know the appropriate rules to follow. Even when rules have been properly specified, obtaining a *valid* (i.e., correctly assembled) composite model requires a lot of manual effort, involving error-prone, time-consuming, and repetitive tasks.

In this paper we address this problem by defining a methodology that supports the automated assembly of large performability models, based on well-specified libraries of reusable submodels. The approach is built around the concept of "model templates" and a specification language that we call Template Models Description Language (TMDL), used to precisely specify and instantiate model templates. The idea

L. Montecchi is with the Institute of Computing, University of Campinas, Brazil. leonardo@ic.unicamp.br

P. Lollini, and A. Bondavalli are with the Dipartimento di Matematica e Informatica, University of Firenze, Italy. {lollini,bondavalli}@unifi.it

behind the approach was initially introduced in [8]. In this paper we provide a formal definition of the approach, and we apply it to a real use case from the literature.

Summarizing, the contributions of this paper are the following: i) we formally define the concept of model templates and their composition, ii) we define a specification language for model templates, iii) we define an automated instantiation and composition algorithm, and iv) we apply the approach to a concrete case study of a large-scale distributed system.

The approach has been devised with the main objective of facilitating the selection, parametrization and composition of predefined models from model libraries. The key distinguishing aspects of our approach are the following: i) it enables the formal specification of libraries of model templates and composition patterns, with a clear separation between model specification (the "interface" modeling elements required for model composition) and model implementation (the internal, formalism-specific, structure of the model); ii) it enables the formal specification of the different scenarios that need to be analyzed, which are used to automatically assemble model templates to form the global system model, via a model-transformation algorithm; and iii) such model-transformation algorithm is defined and implemented only once, since composition patterns become part of reusable model libraries.

In the application of the approach we focus on the Stochastic Activity Networks (SANs) formalism [9], [10], for two practical reasons: i) the use case we use as reference [11] was modeled with SANs (Section VI), and ii) we can use the discrete-event simulator provided by Möbius [12] to actually analyze the composed models that are generated.

The paper is organized as follows. Related work is discussed in Section II, while an overview of the proposed approach is presented in Section III. Formal definitions are then given in Section IV. The language to concretely support our framework, TMDL, is introduced in Section V. The application to the use case is then presented in Section VI. Finally, conclusions are drawn in Section VII.

II. Related Work

Work related to this paper can be grouped according to three main topics: i) modular approaches to construct performability models, ii) variants of Petri Nets that provide a compact notation, and iii) model-driven approaches applied to performability evaluation. Each group is discussed in more detail in the following.

A. Modularity in Performability Models

The application of modularity and composition for tackling the complexity of modern systems is well-established in domains like software engineering, e.g., [13], and real-time embedded systems, e.g., [14], [15]. Our focus is on the modular construction of state-based models for performability analysis, by means of reusable elements. In this context, compositional modeling approaches were initially introduced to reduce the size of the generated state space [16] or its representation [17]. Other approaches, e.g., [18], [19], define strategies for decomposing models in submodels having specific characteristics, and exploit them to achieve a more efficient solution.

Modularization has later gained importance also for improving the *specification* of models, since it also brings a number of other practical advantages: submodels are usually simpler to be managed, they can be reused, they can be refined, and they can be modified in isolation from other parts of the model. While techniques for efficient analysis of performability models are fundamental, the growing complexity of modern systems is also posing challenges for the specification of models. In this paper we focus on this aspect, while discussions on efficient evaluation methods can be found for example in [1], [20].

Several approaches based on Petri nets and their extensions apply modularity in the construction of performability models. In such approaches, the overall model is built out of a well-defined set of submodels addressing specific aspects of the systems, which are then composed by state sharing following predefined rules based on the actual scenario to be represented. Often, submodels include parameters and structural variability, to improve their reuse. Examples of works that apply such approach are [11], [21], [22], [23], [24], [25], [26], [27], [28].

However, composition patterns and variability aspects are typically provided informally or by examples. Sometimes, those "rules" are not even written somewhere, but they are only known to the person(s) that developed the library of models for the system under analysis. The main gap in applying this approach is that, while established formalisms exist both for defining the submodels (SPNs) and for composing them (state sharing), means to define customized patterns for their instantiation and composition are limited. Consequently, reusing submodels and composition patterns, and sharing them between different teams is currently impracticable.

In this paper we address exactly this problem, by defining a methodology for specifying and using libraries of reusable submodels. The ability to precisely define generic submodels and composition patterns allows the overall performability model to be automatically assembled via model transformation from a high-level specification of the scenario of interest.

B. Compact Specification of PN-Based Models

In the literature, several variants of the Petri Nets formalism have been defined, some of them having features that provide more compact and reusable specifications. The Box Algebra [29] operates on a restricted class of Petri net models (boxes) that represent a step of computation. Boxes are composed using process algebra operators, which are mapped to specific Petri net submodels (operator boxes). Composition is performed by essentially replacing the transitions of operator boxes with the submodels used as operands, possibly synchronizing transitions based on their labels. In our approach, composition is performed by fusion of state variables, without knowledge of the internal behavior (transitions) of models. Therefore, we do not impose restrictions on their internal structure.

Colored Petri Nets (CPNs) [30] allow tokens to be distinguished, by attaching data to them. Tokens can be of different data types, called colors. Hierarchical CPNs support

modularization by means of substitution transitions, i.e., a transition is replaced by a whole subnet in a more detailed model. CPNs are able to provide very compact representations, and use concepts similar to those we propose in this paper (e.g., multiplicity).

Stochastic Reward Nets (SRNs) [31] also contain features that allow for a compact specification of SPNs, e.g., marking dependency, variable-cardinality arcs, priorities, etc. Furthermore, they embed extensions to define reward rates. We emphasize however that our objective is not to propose a new variant of Petri nets, but to automatize the composition of models exploiting existing primitives (i.e., sharing of state variables).

Stochastic Activity Networks (SANs) can also be considered a variant of SPNs [10]. In their Möbius implementation [12], they support tokens having different datatypes, including structured datatypes. The *input gate* and *output gate* primitives can be used to specify arbitrary complex functions for the enabling of transitions (called activities) and for their effects. SANs models can be composed using the Rep/Join state sharing formalism [16]. However, which state variables are composed, and how, is specified manually for each composition. In this paper we propose an approach based on Model-Driven Engineering (MDE) [32] techniques to i) define reusable composition patterns, and ii) automate their application. Such automation also reduces the possibility of introducing human mistakes in the model specification.

C. Model-Driven Approaches

Several works in the literature have applied MDE techniques for the automatic derivation of performance and/or dependability models from UML (Unified Modeling Language) or similar representations, e.g., see [33], [34], [35]. However, the purpose of such approaches is usually to provide an application-specific abstraction to users of a certain domain, in the form of a UML profile, and then automatically derive formal models defined by an expert. Composition patterns are embedded in the model transformation algorithm, which is different for every different library of submodels. Reuse across different domains, or with different libraries of submodels is not usually a concern.

More recently, the authors of [36] defined an approach to integrate different model generation chains. They assume the existence of multiple, independent, transformation chains, each one directly generating a formal model (e.g., SPNs) from a high-level model (e.g. UML). The integration of a pair of such transformation chains is performed by means of an "integration model", defined at UML level, which is then processed by a third transformation to generate a low-level integration model that connects the generated formal models.

While we share a similar long-term objective (a framework to reuse performability models), we address the problem from a different perspective. Our approach defines a method to specify and instantiate libraries of reusable submodels specified with SPNs. The language we define, TMDL (see Section V), is a sort of "intermediate model" to specify composition patterns for SPNs, as opposed to using generic transformation languages. In a certain sense, we are proposing an API (Application Programming Interface) for the composition of SPN-based models. Composition rules become part of model libraries, and only one single model transformation/composition algorithm is defined, which is the same for every library of model templates. Essentially, we use a *horizontal* intermediate model (between the design-level representation and formal models), while the work in [36] uses a *vertical* intermediate model (between different transformation chains).

Finally, it should be noted that existing modeling frameworks, e.g., Möbius [12] or CPNTools [30], provide some means for reducing the effort in the specification of complex models. For example, they both allow multiple instances of a submodel to be reused. However, instances have identical structures, and each of them still needs to be manually connected to the rest of the model. Our objective here is to propose an approach that facilitates the selection, parametrization, and composition of predefined models from model libraries, without knowledge of their internal implementation.

With a partially similar idea, the recent work in [37] defines an approach for automated "non-anonymous" replication of SANs models. That is, instead of replicating SANs models with the Rep operator [12], leading to identical replicas, they automatically generate a set of model instances, auxiliary places, and Join operators, based on predefined rules. Their objective is however to achieve efficient simulation, and do not focus on simplifying the specification or reusing models. Also, the approach in [37] is limited to replication only. In this paper, we define a generic approach to specify complex composition patterns and automate their application using MDE techniques.

III. Approach Overview

Before introducing formal definitions, here we provide an overview of the proposed approach, in terms of its requirements (Section III-A), the overall workflow (Section III-B), the underlying methodology (Section III-C), and the main concepts (Section III-D). Formal definitions of such concepts are provided next, in Section IV.

A. Requirements

Throughout the paper we use the term *formalism* to mean "a class of models that share the same primitives and notation". In MDE terms this concept is called a metamodel [38]. Our framework assumes the existence of an *instance-level formalism*, and a *template-level formalism*.

The *instance-level* formalism is a formalism having the concept of state variable [39], and it is the one that is actually used for performability evaluation (e.g., SANs). A *template-level* formalism provides a generalized representation of a set of similar instance-level models, by including variability and parameters. We do not impose the use of a specific template-level formalism. We only assume the existence of a *concretize()* function, which generates an instance-level model, taking as input a template-level model and an assignment of values to its parameters. Further details on this function are provided in Section IV-F.

Different approaches for defining a template-level formalism can be used, depending on the adopted instance-level formalism, and on the desired level of detail. A possible approach is

to use languages like the Common Variability Language (CVL, [40]) to specify parameterized variation points. This approach can in principle be applied to define a "template-level version" of any formalism.

On the other extreme, it is possible to use the same formalism both at template-level and at instance-level. In this case *concretize()* is the identity function, implying that template-level models do no include variability at all. Automated model composition based on predefined state sharing patterns would still be applicable.

B. The Workflow

The overall workflow of our approach is depicted in Figure 1. Note that the activities in the workflow are not strictly sequential. In particular, the creation of the model library (Activity 1) is performed once, and the resulting library stored for future access. Libraries are then used repeatedly to construct and to evaluate system models (Activities 2 and 3). When needed, libraries can however be updated.

Activity 1. Starting from requirements and architecture of a system (or class of systems), an expert in performability modeling develops a "Model Templates Library", i.e., a library of reusable model templates and composition rules. Such library consists of two parts:

- *Templates Specifications*, which contain a description of the available model templates, their interfaces and parameters, and information on where their implementation is stored. Templates are specified using the TMDL, which is described later (Section V).
- *Templates Implementations*, which are the internal implementation of atomic model templates specified in the library. For atomic templates, the implementation is a model in the instance-level formalism and the storage format depends on the specific tools that are adopted. The PNML (Petri Net Markup Language) [41] is an option for models based on Petri Nets. Another possibility is to store models using XMI (XML Metadata Interchange) [42], which can be used for any template-level formalism having a metamodel in the MOF (Meta-Object Facility) standard [43]. For composite templates the implementation consists in a set of composition rules, and it is specified using the TMDL.

Descriptions in textual format can also be associated to templates in the library, to facilitate users in managing and selecting them.

Activity 2. The second activity consists in defining the different system configurations that should be analyzed. The input for this activity may come from different sources, e.g., designs of the system architecture, a new system configuration detected by in-place sensors, etc. The configuration to be analyzed is specified using the TMDL as well, as a `Scenario` element. A TMDL `Scenario` defines which templates are needed and how they have to be instantiated. From a practical perspective, `Scenario` specifications can be either created manually starting from informal descriptions (e.g., provided using the natural language), or they can be automatically generated from structured models (e.g., UML models), by applying model transformation techniques. Generation of TMDL `Scenario` specifications from UML models or other high-level representations is outside the scope of this paper.

Activity 3. Starting from the Model Templates Library defined by Activity 1, and from the description of scenarios provided by Activity 2, the models for all the different system configurations are automatically instantiated, assembled, and evaluated. The generation of composed models is accomplished by means of the TMDL "Automated Composition Algorithm", which takes as input a TMDL `Scenario` specification and generates the corresponding model by generating model instances and properly assembling them based on the patterns specified in the TMDL `Library` specification.

It is important to note that the "Automated Composition Algorithm" of Figure 1 is *the same for every library of template models*, i.e., it is specified and implemented *only once*, and it is reused to automatically assemble models of different systems, possibly implemented in different state-based formalisms. This is one of the key points of our approach, and the main reason to develop the TMDL.

C. Composition Systems

From a methodological point of view, our approach defines a new composition system for the domain of state-based models.

According to [44], a *composition system* is defined by three elements: i) composition technique, ii) component model, and iii) composition language. The *component model* defines what a component is, how it can be accessed, and which are its interfaces for composition. The *composition technique* defines how components are physically connected. The *composition language* is used to specify "composition programs", i.e., specifications of which components should be selected and connected, and how, in order to obtain the intended product.

Looking at the workflow in Figure 1 under this perspective, the *component model* is given by the definition of model templates (Section IV-C); the *composition technique* is superposition of state variables on model interfaces; the *composition language* is the TMDL we define in Section V. TMDL `Scenario` specifications are our notion of composition programs. Finally, the automated composition algorithm of Figure 1 is the *composition engine*, which interprets composition programs specified in TMDL, and properly assembles instances of the selected model templates.

D. Main Concepts

We introduce here the main concepts of our approach. The basic building blocks are ***model templates***, which are the result of Activity 1. A set of model templates constitutes a *model templates library* (or *model library* for short). A model template consists of a *specification* and an *implementation*. The specification includes a set of *interfaces*, which specify how a template can interact with the other models, a set of *parameters*, and a set of *observation points*. The latter are selected state variables that can be used to "observe" the model, i.e., to define metrics. Interfaces and observation points

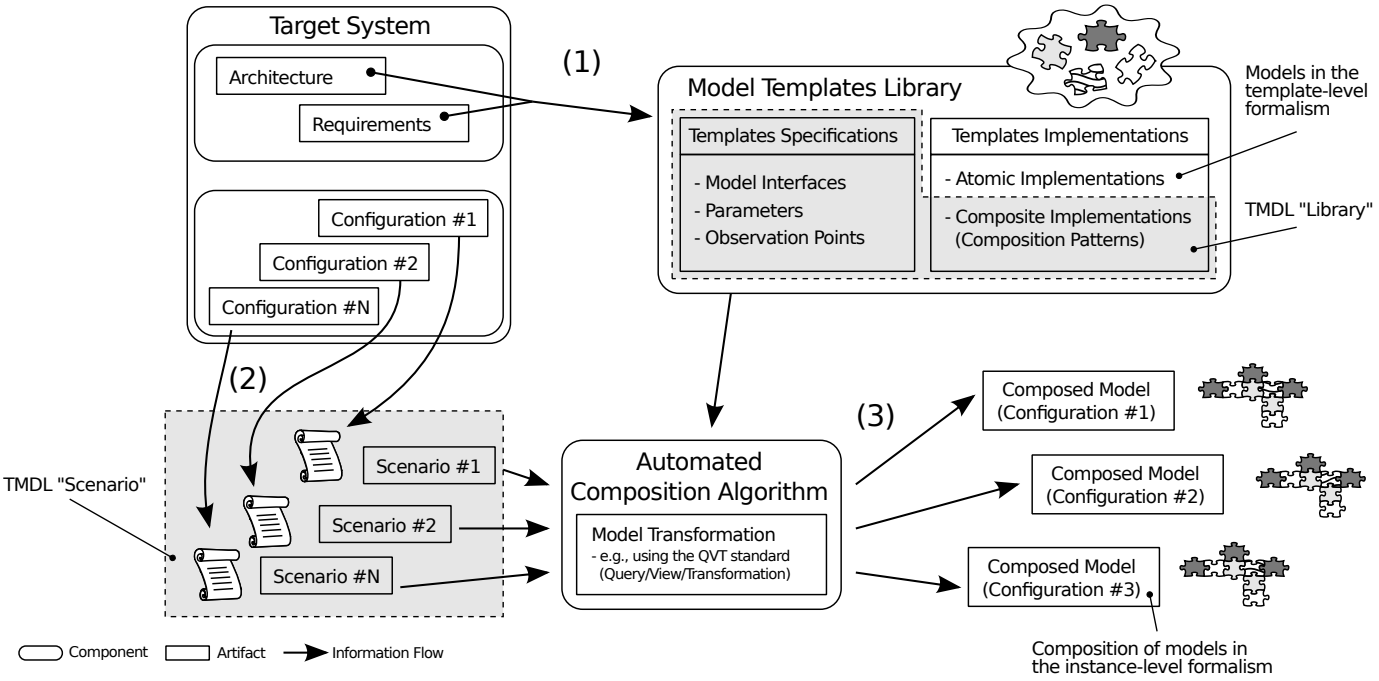

Figure 1: Our workflow for the automated generation of performability models. Elements depicted in gray are specified using the TMDL (Template Models Description Language), which is defined in Section V.

are defined in terms of *state meta-variables*, i.e., state variables with variability elements.

The implementation describes how the internal behavior of the template is realized. Based on its implementation, a model template can be either an *atomic template* or a *composite template*. For atomic templates the implementation consists in a model in the template-level formalism of choice.

Composite templates specify patterns to compose other model templates. Their implementation consists in a set of *blocks* and a set of *composition rules*. Each block is a placeholder for instances of other model templates. Composition rules define patterns to connect such instances.

A ***model variant*** is obtained from a model template by resolving all the variation points. That is, an *atomic variant* is a reference to an atomic template and an assignment for its parameters. A *composite variant*, in addition to assigning values to template parameters, also specifies which other variants are used to fill the blocks defined in the template. A selection of model variants defines a *scenario* to be analyzed, and it is the output of Activity 2.

A ***model instance*** is an individual instantiation of a model variant. Multiple instances of a model variant can be used to construct a global system model. An *atomic instance* is a concrete model in the instance-level formalism of choice (e.g., SANs), derived by applying the *concretize()* function to the atomic template implementation and the assignment of parameters in the variant. A *composite instance* is an instance of a composite variant, i.e., a collection of other model instances that get composed according to the rules defined in the composite template. Model instances are the output of Activity 3, and are automatically generated by the model composition algorithm.

IV. The Templates Framework: Formal Definitions

We now take a step forward and provide formal definitions for the concepts introduced in the previous section. In Section IV-A we introduce basic definitions that will be used in the rest of the paper, while in Section IV-B we define the concept of model interfaces. Section IV-C, Section IV-D, and Section IV-E, formalize the concepts of *templates*, *variants*, and *instances*, respectively.

A. Preliminaries

1) Basic Definitions: We adopt the definitions of *sort*, *operator*, *term* and *assignment* from the ISO/IEC 15909 standard [45], which apply to a wide range of formalisms based on Petri Nets. However, instead of places, we use the more general concept of *state variable*, as in [39]. The definitions of *term* and *assignment* will be used extensively in the rest of the paper.

A *state variable* is the basic unit of decomposition of system state. The set of possible values of a state variable is defined by its associated *sort* (i.e., type).

A *many-sorted signature* is a pair (S, O), where S is a set of sorts and O is a set of *operators*, together with their arity. Arity is a function from the set of operators to $S^* \times S$, where S^* is the set of finite sequences over S, including the empty string ε. An operator is thus denoted as $o_{(\sigma,s)}$, where $\sigma \in S^*$ are the input sorts, and $s \in S$ is the output sort. Constants are operators with empty input sorts, and are denoted as $o_{(\varepsilon,s)}$ or simply o_s.

We denote with Δ a set of parameters; a parameter in Δ of sort $s \in S$ is denoted by δ_s. $\Delta_s \subseteq \Delta$ is the set of parameters of sort s.

Terms of sort $s \in S$ may be built from a signature (S, O) and a set of parameters Δ. The set of terms of sort s is denoted by $\mathtt{TERM}(O \cup \Delta)_s$, defined inductively as [45]:

- for all $o_{(\varepsilon,s)} \in O$, $o_{(\varepsilon,s)} \in \mathtt{TERM}(O \cup \Delta)_s$;
- $\Delta_s \subseteq \mathtt{TERM}(O \cup \Delta)_s$; and
- for $s_1, \ldots, s_n \in S$, if $e_1 \in \mathtt{TERM}(O \cup \Delta)_{s_1}, \ldots, e_n \in \mathtt{TERM}(O \cup \Delta)_{s_n}$ are terms and $o_{(s_1 \ldots s_n, s)} \in O$, is an operator, then $o_{(s_1 \ldots s_n, s)}(e_1, \ldots, e_n) \in \mathtt{TERM}(O \cup \Delta)_s$.

A *many-sorted algebra* H provides an interpretation of a signature (S, O). For every sort $s \in S$ there is a corresponding set H_s, and for every operator $o_{s_1 \ldots s_n, s} \in O$ there is a corresponding function $o_H \colon H_{s_1} \times \ldots \times H_{s_n} \to H_s$. A many-sorted algebra is thus a pair $H = (S_H, O_H)$, where $S_H = \{H_s | s \in S\}, \forall s \in S, H_s \neq \emptyset$, and $O_H = \{o_H | o_{(\sigma,s)} \in O\}$ is the set of corresponding functions.

Given a many-sorted algebra H, and many-sorted parameters in Δ, an *assignment* for H and Δ is a family of functions ξ, comprising an assignment function for each sort $s \in S$, $\xi_s \colon \Delta_s \to H_s$. The function may be extended to terms by defining a family of functions Val_ξ comprising, for each sort $s \in S$ the function $Val_{s,\xi} \colon \mathtt{TERM}(O \cup \Delta)_s \to H_s$ [45].

To support the subsequent definitions, we require the existence of at least the "integer", "real", "set of integers", and "set of reals" sorts. Formally, we assume a signature (S, O), such that $\{\mathtt{Int}, \mathtt{Real}, \mathtt{Set\{Int\}}, \mathtt{Set\{Real\}}\} \subseteq S$, and O contains the common operators applicable on such sorts. The corresponding many-sorted algebra is (S_H, O_H), with $\{\mathbb{N}, \mathbb{R}, \mathcal{P}(\mathbb{N}), \mathcal{P}(\mathbb{R})\} \subseteq S_H$, and O_H containing the set of functions corresponding to operators in O.

2) Indices, Multiplicity, Labels: Template-level models contain variable elements, which serve as a placeholder for a set of concrete elements that will be derived upon the instantiation of the template. Multiple concrete elements can be originated from the same variable element. We use indices to distinguish them. Such indices are not required to form a progressive sequence.

Figure 2 shows a simple example adapted from the SANs models in [24]. The right part of the figure shows two models in the instance-level formalism (SANs), representing two different classes of users in a mobile network. *UserX* may make requests for services 1, 6, or 7, by adding a token in places *Req1*, *Req6*, or *Req7*, with a certain probability. Similarly, *UserAmbulance* may make requests for services 3 or 7, by adding a token in places *Req3* or *Req7*, with a certain probability. This behavior can be abstracted to a generic *User* template model, shown in the left part of the figure. Such model contains a generic "place template" *Req*. Based on the value assigned to parameters of the template-level model, different models in the instance-level formalism are generated by *concretize()*, resulting in a different model structure and different indices.

The number of concrete elements to which a template element gets mapped, and their indices, is called its *multiplicity*, denoted as k. The multiplicity is specified as a set of values in $\mathbb{N}$ (i.e., $k \subseteq \mathbb{N}$), which indicate the indices assigned to concrete elements. In the previous example, the multiplicity assigned to the *Req* place in the generation of *UserX* would be $k = \{1, 6, 7\}$. In general, the multiplicity

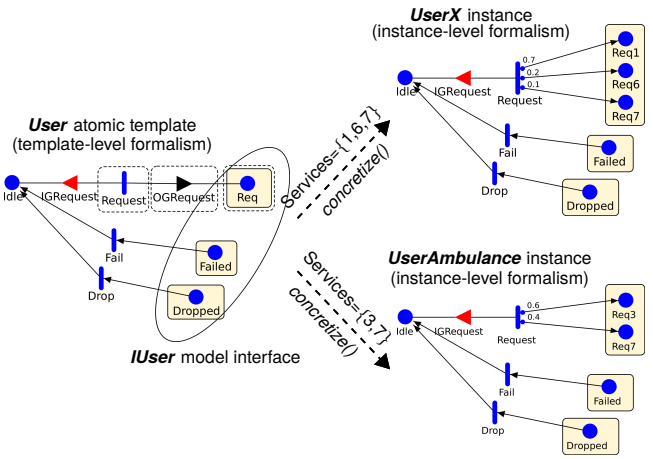

Figure 2: Example of two different instances originating from the same *User* template, using SAN as the instance-level formalism. The *IUser* model interface is formed by the state meta-variables *Req*, *Failed*, and *Dropped*.

may depend from the parameters of the template, and is thus more accurately defined as a term of sort "set of integers", i.e., $k \in \mathtt{TERM}(O \cup \Delta)_{\mathtt{Set\{Int\}}}$.

In our framework, variability (and thus multiplicity) appears at different levels, i.e., elements of an atomic template, instances of atomic templates, block of composite templates, instances of composite templates. For this reason, state variables in a model may be assigned more than one index, identifying the variable across different dimensions. To generalize this aspect, each index is coupled with a label, which identifies the dimension that is captured by the index.

Formally, a *labeled state variable* is a triple (v_s, L^ξ, f), where v_s is the state variable (of sort s), L^ξ is the set of labels, and $f \colon L^\xi \to \mathbb{N}$ is a function that associates a numeric index with each label. Labeled state variables belong to instance-level models; the concrete values of L^ξ and f are a result of the instantiation process, discussed later (Section IV-F). In the rest of the paper, we denote with L^* the set of all the possible labels.

B. Model Interface

A *model interface* defines in which ways a model template can interact with the other models, and with the rest of the framework. A model interface can be *realized* by different model templates having different implementations. Two template models that realize exactly the same model interfaces can be interchanged. Clearly, depending on the internal implementation, the resulting behavior might be different. The selection of the most appropriate template should be based on the scenario under analysis.

A model interface specifies a set of state variables that a model must have. As model templates include variability, model interfaces are specified by means of state variables having variability information, that is, multiplicity and labels. A model interface is thus a set of state "meta-variables" V.

Formally, a *state meta-variable* is a tuple (v_s, Δ^v, L, k), where v is the state variable (of sort s), Δ^v is a set of param-

eters, $L \subseteq L^*$ is a set of labels, and $k \in \texttt{TERM}(O \cup \Delta^v)_{\texttt{Set\{Int\}}}$ is the parametric multiplicity. A multiplicity $k = \emptyset$ indicates that any multiplicity is admitted for that variable. The instantiation process generates a set of labeled state variables (in the instance-level model) from each state meta-variable, and assigns labels and indices to them. We denote as $V^{\xi}(v)$ the set of labeled state variables generated from a state meta-variable v under an assignment ξ.

Following this definition, the *IUser* interface of Figure 2 contains three state meta-variables, and it is specified as:

$$\begin{aligned} &\texttt{IUser} = \{v_1, v_2, v_3\}, \\ &v_1 = (\texttt{Req}_{\texttt{Int}}, \{s_{\texttt{Set\{Int\}}}\}, \{\texttt{srv}\}, s_{\texttt{Set\{Int\}}}), \\ &v_2 = (\texttt{Failed}_{\texttt{Int}}, \emptyset, \emptyset, \{1\}), \\ &v_3 = (\texttt{Dropped}_{\texttt{Int}}, \emptyset, \emptyset, \{1\}. \end{aligned} \tag{1}$$

Note in particular that: i) the multiplicity k for the *Req* place is defined by the s parameter of sort "set of integers" ($s_{\texttt{Set\{Int\}}}$), ii) the multiplicity of *Failed* and *Dropped* is constant, and iii) the *Req* state meta-variable is labeled with the `srv` label.

C. Templates

A *model template* is defined as:

$$MT = ((\mathcal{I}, \Delta, O, L_T), \Psi),$$
$$\Psi = \begin{cases} \mathcal{M} & \text{for } \textit{atomic} \text{ templates,} \\ (\mathcal{B}, \mathcal{R}) & \text{for } \textit{composite} \text{ templates.} \end{cases}$$

The quadruple $(\mathcal{I}, \Delta, O, L_T)$ constitutes the *specification* of the template. $\mathcal{I}$ is the set of model interfaces that are realized by the template, Δ is the set of parameters, O is the set of *observation points*, and $L_T \subseteq L^*$ is the set of template-specific labels. Note that $\forall V \in \mathcal{I},\ \forall (v, \Delta^v, L, k) \in V,\ \Delta^v \subseteq \Delta$ must hold, i.e., the parameters of the template must include all the parameters of its interfaces. Observation points are specific state meta-variables that the model shall have, with the purpose to define metrics for observation. A state meta-variable may act at the same time as an interface variable and as an observation point.

Ψ is the *implementation* of the template, which can be either atomic or composite. For an *atomic* template, $\Psi = \mathcal{M}$, where $\mathcal{M}$ is a model in the template-level formalism of choice; in our examples we use a formalism derived from SANs. For completeness, its formal definition can be found in a technical report [46]. The implementation of the template must be compatible with its specification, i.e., the implementation should contain all the state meta-variables that are declared in the specification (interface variables and observation points).

The implementation of a *composite* template is given by $\Psi = (\mathcal{B}, \mathcal{R})$, where $\mathcal{B}$ is a set of *blocks* and $\mathcal{R}$ is a set of *composition rules*. Blocks are slots to be filled by instances of other templates, which will then be composed together according to the rules in $\mathcal{R}$. Blocks define which kinds of templates are required to perform the composition, and their roles.

A block $b \in \mathcal{B}$ is a triple $(\mathcal{I}_b, L_b, k_b)$, where: $\mathcal{I}_b$ is the set of required model interfaces; $L_b \subseteq L^*$ is a set of block-specific labels; and $k_b \in \texttt{TERM}(O \cup \Delta)_{\texttt{Set\{Int\}}}$ is the block multiplicity. $k_b = \emptyset$ indicates an optional block with unspecified multiplicity. Block-specific labels allow instances originating from different blocks to be distinguished, i.e., they identify different roles in the composition. The multiplicity defines how many instances of the block should be generated, possibly based on template parameters.

Each composition rule $r \in \mathcal{R}$ defines how a specific set of labeled state variables belonging to different block instances will be connected together. Three composition rules are possible in our framework: *all*, *match*, and *forward*. Clearly, only state variables of the same sort can be connected. Besides specifying how variables are connected, they also specify how indices and labels are altered, thus defining labeled state variables for the resulting composite model instance.

We denote with $V^{\mathcal{B}}$ the set of all the interface variables that are required by all the blocks in $\mathcal{B}$, i.e., $V^{\mathcal{B}} = \{v \in V \mid V \in \mathcal{I}, (\mathcal{I}, L, k) \in \mathcal{B}\}$. The subset of $V^{\mathcal{B}}$ formed by its elements having sort s is denoted as $V_s^{\mathcal{B}}$. Then, for each $s \in \mathcal{S}$:

- An ***all*** rule is a pair (ω, W), with $W \subseteq V_s^{\mathcal{B}}$. The rule specifies that all the labeled state variables generated from interface variables in W should be composed together. Such composition forms a new labeled state variable $(\omega, \emptyset, \emptyset)$.
- A ***match*** rule is a triple (ω, W, L), with $W \subseteq V_s^{\mathcal{B}}$, and $L \subseteq L^*$ a set of labels. The rule specifies that labeled state variables generated from interface variables in W should be composed together based on their labels and associated indices. That is, instances having the same indices for labels in L are composed together: a labeled state variable (a, L_a, f_a) is composed with any (b, L_b, f_b) such that $f_a(l) = f_b(l)\ \forall l \in L$. Each of such compositions forms a new labeled state variable $(\omega, L_a \cup L_b, f')$.
- A ***forward*** rule is a pair (ω, W), with $W \subseteq V_s^{\mathcal{B}}$, specifying that all the interface variable instances generated from interface variables in W will simply become state variables of the composite instance, available to be exposed as interfaces.

Note that also for composite templates the implementation $(\mathcal{B}, \mathcal{R})$ must comply with the specification $(\mathcal{I}, \Delta, O, L_T)$. In this case it means that i) for each meta-variable of interfaces specified in $\mathcal{I}$, there is at least one rule in $\mathcal{R}$ such that $v = \omega$, and ii) for each observation point in O there is an interface variable required by one of the blocks $\mathcal{B}$.

D. Variants

A model variant is derived from a template by assigning concrete values to its parameters. Formally, a model variant is defined as:

$$MV = (MT, \xi, \gamma),$$
$$\gamma = \begin{cases} \emptyset & \text{for } \textit{atomic} \text{ variants,} \\ \gamma \colon \mathcal{B} \to \mathcal{P}(\mathcal{V}) & \text{for } \textit{composite} \text{ variants.} \end{cases}$$

where MT is a model template, and ξ is an assignment. For composite variants (i.e., those derived from composite templates), $\gamma \colon \mathcal{B} \to \mathcal{P}(\mathcal{V})$, with $\mathcal{V}$ the set of all model variants, is a function that assigns a set of variants to each block of the referenced template. Note that the variants selected by γ must realize all the model interfaces required by the block, i.e.,

$\forall(\mathcal{I}_b, L_b, k_b) \in \mathcal{B},\ \forall((\mathcal{I}, \Delta, O, L_T), \Psi) \in \gamma(b),\ \mathcal{I}_b \subseteq \mathcal{I}$. For variants of atomic templates, γ is the empty function.

E. Instances

A model instance is derived by combining i) the implementation given in the model template, and ii) the assignment of parameters given in the model variant. To build a model for a given scenario, many instances derived from the different variants are typically needed. The definition of a composite template is recursive: it is built connecting together other model templates, which in turn can be either atomic or composite. Instances are thus organized in a tree, where internal nodes are instances of composite templates, and leaves are instances of atomic templates. The root of the tree is the model instance representing the *complete model of the system*. We call such model instance the *model root*.

Formally, a *model instance* is defined as:

$$MI = (k^\xi, V^\xi, O^\xi, \Psi^\xi),$$
$$\Psi^\xi = \begin{cases} \mathcal{M}^\xi & \text{for } \textit{atomic} \text{ instances,} \\ (I^\xi, C^\xi) & \text{for } \textit{composite} \text{ instances.} \end{cases}$$

where $k^\xi \in \mathbb{N}$ is the index of the instance, V^ξ is the set of labeled state variables derived from interface variables of the template, O^ξ is the set of labeled state variables derived from observation points of the template, and Ψ^ξ is the implementation of the instance.

For an *atomic instance*, $\Psi^\xi = \mathcal{M}^\xi$ is a concrete model in the instance-level formalism (SANs in our case), derived through the *concretize()* function from the implementation of the template $\mathcal{M}$, and the assignment ξ in the variant.

For a *composite instance*, $\Psi^\xi = (I^\xi, C^\xi)$, where I^ξ is the set of model instances that have been generated from the blocks of the template, and C^ξ is a set of connections (i.e., superposition of state variables) of such model instances, as in [30]. The implementation of composite instances is generated by the automated composition algorithm, described in the next section. In both cases, V^ξ, O^ξ, and k^ξ are also generated by the automated composition algorithm. That is, model instances are fully generated by the composition algorithm.

F. Instantiation and Composition Algorithm

The instantiation and composition algorithm is constituted of three procedures: *instantiateVariant*, *concretize*, and *connectStateVariables*, described in the following.

1) instantiateVariant: This is the main procedure of the composition algorithm, and it is executed on the model variant corresponding to the scenario to be analyzed. The algorithm is recursive, and consists of two phases. In the first phase, the tree of model templates is traversed top-down, all the model instances are generated, and they are temporarily added to a list. This step defines the set I^ξ of each composite instance. Indexes are assigned by evaluating the terms in the multiplicity specification with current parameter values.

In the second phase, instances are progressively retrieved from the list, and connected together according to the composition rules in their template. The pseudo-code of this procedure is listed in the following.

```
instantiateVariant((MT, ξ, γ), k) {
  /* allInstances variable is a set of model instances */
  allInstances as List = {};
  /* Phase 1: Generate Instances */
  if(MT is atomic) {
    /* for atomic templates call the formalism−specific */
    /* instantiation procedure */
    inst = concretize(MT, ξ);
    inst−>setIndex(k);
    allInstances = allInstances−>append(inst);
  }else{
    /* For composite templates, instantiate all the blocks */
    /* Note: MT = ((I, Δ, L_T), (B, R)) */
    for( b ∈ B ) {
      /* For each block − Note: b = (I_b, L_b, k_b) */
      for(v ∈ γ(B) {
        /* For each variant assigned to the block */
        for(j ∈ Val_ξ(k_b)) {
          /* For each term in the multiplicity */
          /* generate an instance with that index */
          inst = instantiateVariant(v, j);
          allInstances = allInstances−>append(inst);
        }
      }
    }
  }
  /* Phase 2: Connect state variables according to the rules */
  for( i ∈ allInstances ) {
    connectStateVariables(i);
  }
  return allInstances;
}
```

2) concretize: As introduced in Section III-A, we assume the existence of a *concretize()* function, which generates a model in the instance-level formalism, based on a template-level formalism and an assignment of parameters. That is, it takes as input the pair $(\mathcal{M}, \xi)$ and returns $\mathcal{M}^\xi$. The actual implementation of this function depends on the selected template-level and instance-level formalisms. An interested reader may find a definition of this function for a template-level formalism based on SANs in the technical report in [46].

3) connectStateVariables: This procedure assigns labels and indices to interface variables of the generated instances, and executes connection rules specified in composite templates. The procedure takes as input a model instance $MI_i = (i, V^\xi, O^\xi, \Psi^\xi)$.

For atomic instances, the procedure generates labeled state variables from meta-variables, assigning them initial indices and labels. That is, it populates sets V^ξ and O^ξ. Given a state meta-variable (v, Δ^v, L, k), and the assignment ξ, the set of labeled state variables derived from it is:

$$\left\{(v_j, L^\xi, f) \mid j \in Val_\xi(k)\right\},$$

with $L^\xi = L \cup L_T$, and L_T the set of labels associated with the template. Being j the index associated to the state variable, the function f is defined as:

$$f = \begin{cases} j, & \forall \tau \in L, \\ i, & \forall \tau \in (L_T \setminus L). \end{cases}$$

That is, for each state meta-variable, the indices of generated labeled state variables is given by the set of integers resulting from evaluating the multiplicity k. Each generated state variable receives the labels from its meta-variable, L, which gets

assigned its index, j, as well as those of the template itself, L_T, which get assigned the index of the model instance, i.

For composite instances, i.e., $\Psi = (I^\xi, C^\xi)$ the procedure executes the composition rules in the corresponding template $(\mathcal{I}, \Delta, O, L_T, \Psi = (\mathcal{B}, \mathcal{R}))$, consequently generating labeled state variables for the composite instance. Before actually executing the rules, labeled state variables of children instances in I^ξ are updated, based on properties of the originating blocks. Each labeled state variable (v, L^ξ, f), belonging to an instance filling block $(\mathcal{I}_b, L_b, k_b)$, is updated to (v, L', f'), where $L' = L^\xi \cup L_b$, and, denoting with j the index of the model instance:

$$f'(\alpha) = \begin{cases} j & \text{if } \alpha \in L_b, \\ f(\alpha) & \text{otherwise.} \end{cases}$$

Then, composition rules are applied. Their application defines the sets C^ξ and V^ξ for the composite instance. This step is described in the following, denoting with $\psi(x)$ the state meta-variable from which the labeled state variable x has originated, and with i the index of the composite instance being processed.

- When applying an ***all*** rule (ω, W), all labeled state variables x such that $\psi(x) \in W$ are connected together. The result, which is added to V^ξ, is a single labeled state variable (ω, L', f'), with $L' = L_T$ and $f'(\alpha) = i \quad \forall \alpha \in L'$.
- When applying a ***match*** rule (ω, W, L^ω), any labeled state variable $x = (\omega^x, L_x, f_x)$ is composed with any other labeled state variable $y = (\omega^y, L_y, f_y)$, such that $\psi(x), \psi(y) \in V$ and $f_x(\alpha) = f_y(\alpha), \ \forall \alpha \in L^\omega$. Each composition results in a labeled state variable (ω, L', f') being added to V^ξ, with $L' = L_x \cup L_y \cup L_T$, and:

$$f'(\alpha) = \begin{cases} f_x(\alpha) \equiv f_y(\alpha) & \text{if } \alpha \in L_\omega \cap L_x \cap L_y, \\ f_x(\alpha) & \text{if } \alpha \in L_x \setminus L_y, \\ f_y(\alpha) & \text{if } \alpha \in L_y \setminus L_x, \\ i & \text{otherwise.} \end{cases}$$

  Note that when applying a match rule, each labeled state variable may be composed with one or more other labeled state variables. If no matching variable exists, the outcome of a match rule is exactly the same as the forward rule, defined in the following.
- When applying a ***forward*** rule (ω, W), any labeled state variable $x = (\omega^x, L_x, f_x)$ such that $\psi(x) \in W$ simply becomes a labeled state variable $(\omega^x, L', f') \in V^\xi$ of the composite instance, with $L' = L_x \cup L_T$, and:

$$f'(\alpha) = \begin{cases} i & \text{if } \alpha \in L_T, \\ f(\alpha) & \text{otherwise.} \end{cases}$$

The final step of this procedure consists in managing the observation points of composite instances, that is, defining the O^ξ sets. This task is performed simply by looking at the meta-variables in O in the corresponding template: all the labeled state variables generated from them are added to O^ξ, i.e., $O^\xi = \{v \in V(\omega) \mid \omega \in O\}$.

G. Definition of Metrics

The main purpose for constructing this kind of models is to understand the behavior of a complex system, and evaluate probabilistic metrics. Such metrics are typically defined as *reward variables* [39], [47].

Normally, a reward variable is defined based on a function that maps each state of the model and each transition of the model to a number in $\mathbb{R}$. One of the main objectives of our framework is to encapsulate the internal implementation of templates, and improve reusability of models. For this reason, we restrict reward variables to be defined based on *observation points* only, that is, all the labeled state variables in set O^ξ of any model instance. Such observation points are part of the specification of the template. The model library would then include a textual description of the purpose and meaning of each observation point. Examples of reward variables definition based on observation points are provided in the case study in Section VI.

It should be noted that this approach implies that all the reward variables must be based on *rate rewards* only, since all the observation points are state variables. From a practical point of view, the absence of *impulse rewards* (defined on the firing of transitions) is not a limitation, as an equivalent rate reward can be defined by adding a state variable that tracks the number of transition firings and exposing it as an observation point.

V. Template Models Description Language

We have now introduced all the concepts that enable our automated model composition approach. In this section we define a Domain-Specific Language (DSL) [48] that can be used to concretely specify and use libraries of model templates. We call this language the Template Models Description Language (TMDL).

The metamodel of the TMDL is shown in Figure 3, using the Ecore notation. We use the support for generic types provided by Ecore [49] to implement the *sort* concept; that is, some entities of the metamodel accept a type parameter `S`, indicating their sort. We denote an element of sort `S` with `<S>`.

A `TMDLSpecification` may be either a `Library` (of model templates), or a `Scenario`. Some elements are common to both kinds of specifications. A TMDL `Parameter` has a *name* and type `S`. An `Assignment` references a `Parameter<S>`, and assigns it a concrete *value* of type `S`. The `Set` entity is a representation of the set concept; accordingly, it contains *items* of type `S`. A `Term<S>` is an abstract element representing a generic term $t \in \texttt{TERM}(O \cup \Delta)_s$. Each `Term<S>` can be evaluated with respect to a set of `Assignment` elements, returning a value of type `S`.

There can be different kinds of *terms*, each one representing an operator in O. In Figure 3 we included those that specify: i) literal values, ii) parametric values, and iii) set of values. `TermLiteral<S>` is a term representing a literal value of sort `S`. `TermParameter<S>` is a reference to a `Parameter<S>`. `TermSet<S>` is an abstract element extending `Term<Set<S>>`. A `TermSetOfTerms<S>` contains a set of elements of sort `S`, thus being able to return a

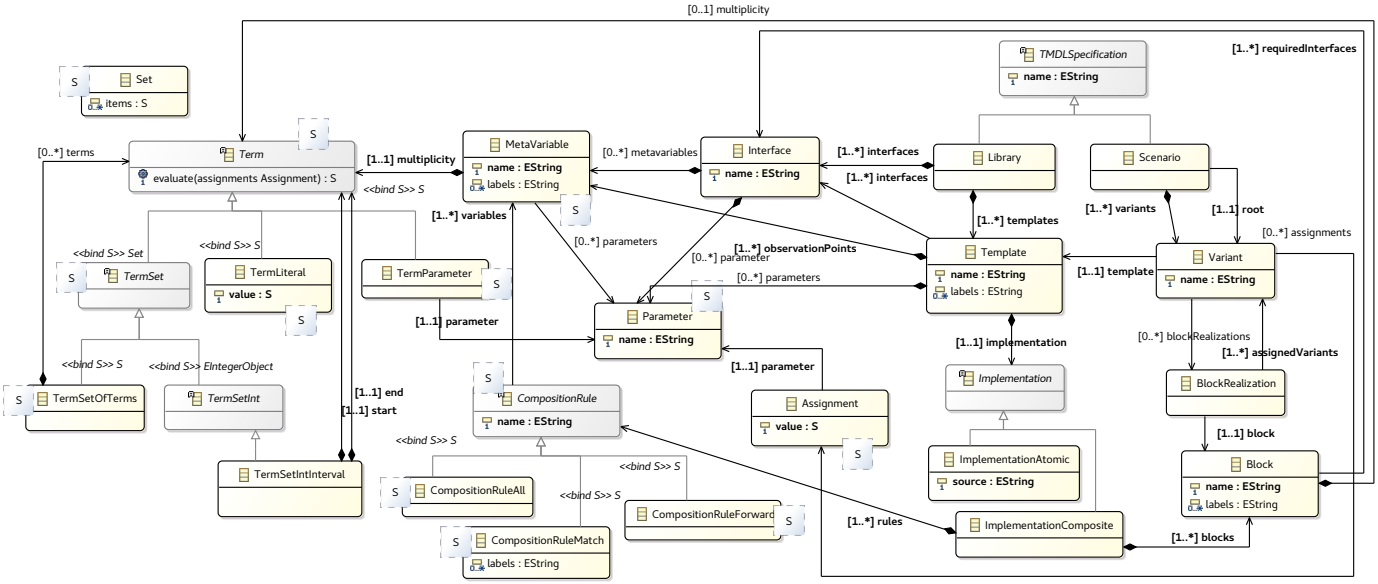

Figure 3: TMDL metamodel.

value of sort `Set<S>` when evaluated. `TermSetInt` extends `TermSet<EIntegerObject>`[1], meaning that it is a `Term` of sort "set of integers", i.e., $t \in \texttt{TERM}(O \cup \Delta)_{\texttt{Set\{Int\}}}$. Finally, a `TermSetIntInterval` represents the set of integers $\{n \in \mathbb{N} \mid \textit{start} \leq n \leq \textit{end}\}$, where *start* and *end* are terms of integer type, i.e. `Term<EIntegerObject>` elements.

The hierarchy can then be extended with new operators as needed, thus extending the expressiveness of the framework. For example, a `SumInt` operator could be added, to specify a term of sort `Int` as the arithmetic sum of other terms of sort `Int`. This would allow the modeler, for example, to define a multiplicity as the sum of the values assigned to various parameters.

A. TMDL Library

A TMDL `Library` is created in Activity 1 of the workflow in Figure 1. A `Library` contains a set of `Interface` and `Template` elements.

A `MetaVariable` has a *name* (v), a sort (`S`), a set of *labels* (L), and a *mutliplicity* (k), which is specified as a `Term` of sort `Set<EIntegerObject>`. In TMDL, parameters are owned by `Interface` and `Template` elements. As such, each `MetaVariable` only contains references to the parameters it uses.

An `Interface` contains a set of `MetaVariable` elements (V) and a set of `Parameters`; this is the union of all the parameters of all meta-variables contained in the interface, i.e., $\{\delta \in \Delta^v \mid (v, s, \Delta^v, L, k) \in V\}$.

Each template references a set of `Interface` elements ($\mathcal{I}$), it has a set of `Parameter` elements (Δ), a set of `MetaVariable` elements that define observation points (O), and a set of *labels* (L_T), which collectively represent the specification of the template. A template also includes an `Implementation` (Ψ), which may be either atomic or composite. An `ImplementationAtomic` consists of a *source* attribute that indicates where the implementation of that template can be found ($\mathcal{M}$). An `ImplementationComposite` consists of a set of `Block` elements ($\mathcal{B}$) and a set of `CompositionRule` elements ($\mathcal{R}$).

A `Block` has a *name* and a set of *labels* (L_b), and it references a set of `Interface` elements ($\mathcal{I}_b$). As for meta-variables, the multiplicity of the block (k_b) is specified as a `Term` of sort `Set<EIntegerObject>`.

A `CompositionRule` has a *name* (ω) and it references a set of `MetaVariable` elements (W). In accordance with the definitions in Section IV, three kinds of composition rules exist: `CompositionRuleAll`, `CompositionRuleMatch` and `CompositionRuleForward`. For rules of kind `CompositionRuleMatch` a set of labels on which the composition is restricted can be specified (L). If it is not specified we assume $L = L^*$, i.e., composition is performed taking into account all the labels.

B. TMDL Scenario

A TMDL `Scenario` defines a concrete scenario to be evaluated, and it is produced during Activity 2 of the workflow in Figure 1.

A `Scenario` consists of a set of model variants (`Variant` elements). One of them is marked as the *model root*, which identifies the entry point of the automated composition algorithm. Each `Variant` contains a reference to a `Template` (MT) and a set of `Assignment` elements (ξ). A composite variant also contains a set of `BlockRealization` elements (γ): each of them associates a `Block` b with the set of `Variant` elements that fill that block, i.e., it defines $\gamma(b)$.

VI. Application Example

In this section we demonstrate the application of our approach to a concrete case study from the literature. The

[1] `EIntegerObject` is the Ecore type for integer objects, and it is mapped to `java.lang.Integer`.

case study is based on the work in [11], in which a large-scale distributed application was analyzed, focusing on how different configuration and arrangement of system components would reflect on performability metrics.

A. The World Opera

The modeled system is the one envisioned by "The World Opera" (WO) consortium, which aims at conducting distributed, real-time, live opera performances across the world. Participating artists from different real-world stages are mapped to virtual-world stages, which are projected as video and shown to the audience at the local opera house, as well as to audiences at geographically distributed (remote) opera houses.

The infrastructure enabling such an application includes a high number of specialized hardware and software components, whose slight malfunction could severely affect the performance. Fault-tolerant architectural solutions are therefore necessary to ensure the correct execution of a WO performance. To design such solutions it is essential to understand the interactions between components, and the potential effects of their failures on the overall quality as perceived by users.

The typical setup for a World Opera performance consists of 3 to 7 real-world stages with different artists and possibly a different set of technical components (microphones, projectors, etc.). Components like cameras and microphones generate multimedia streams of different kinds: video, audio, and sensor (e.g., to track the movement of an artist on the stage). Streams are then processed in different ways, and transmitted to and from the remote stages. Finally, streams are rendered, i.e., decoded, synchronized, and reproduced to the audience.

The number and kinds of components in a certain stage, and their interconnections, depend on the artists present in the stage, and the role of the stage in the overall performance. For example, some stages may only contain audience, while others may contain only a specific set of artists.

In this context, model-based evaluation is needed to evaluate the impact of component failures and compare different architectural solutions. Metrics for this kind of systems are based on the reliability and availability of individual application streams (i.e., video and audio streams) during the show. Given an application stream j (e.g., audio or video stream), and being T the duration of the show, we are interested in the following metrics: i) $R^j(T)$, the reliability of the stream until the end of the show, and ii) $A^j(0,T)$, the fraction of time the stream is available during the show (interval-of-time availability).

This scenario is well-suited for showing the benefits of our approach. The complexity is given not only by components having variability in their structure, but also by the complex interconnections that exist between them, and the need to modify them in order to assess different configurations.

B. Model Templates

The model in [11] considers components and streams as the basic elements of a WO performance, both having different possible working states (e.g., working/failed for components, good/missing/delayed for streams). The state of a stream in a certain point of the architecture depends on the state of all the components that have processed it so far (including components that captured it). The state of different streams as they are reproduced to the audience determines the quality as perceived by the users and it is therefore the target of evaluation.

In [11], a template-based approach was "empirically" used, that is, without formally defining such templates, variability aspects, or composition rules. Four atomic templates were identified: *Component*, *StreamAcquiring*, *StreamProcessing*, and *StreamMixing*. *Component* represents a single functional component of the WO architecture, *StreamAcquiring* models the process of capturing a stream, and *StreamProcessing* models the processing of the stream by some component. *StreamMixing* represents the act of mixing two or more streams in a single one.

Instances of those templates are connected in a way that reflects the path that multimedia streams follow across the stage components. By changing the way in which such templates are arranged it is possible to assess different scenarios in an efficient way. For example, it may be required to evaluate how the target metrics would change if a certain stream is processed by a given workstation instead of another one, or if different combinations of components are used to reproduce multimedia streams.

All the model templates for the WO system interact by means of three model interfaces, each comprising a single meta-variable (v_s, Δ^v, L, k):

$$\begin{aligned}
&\texttt{IComponent} = \{(\texttt{FailedState}_{\texttt{Int}}, \emptyset, \emptyset, \{1\})\}\,;\\
&\texttt{IStreamInput} = \{(\texttt{StreamStateIn}_{\texttt{Int}}, \emptyset, \emptyset, \emptyset)\}\,;\\
&\texttt{IStreamOutput} = \{(\texttt{StreamStateOut}_{\texttt{Int}}, \emptyset, \emptyset, \emptyset)\}\,;
\end{aligned} \tag{2}$$

The *IComponent* interface provides a view of the current working state of a component, through the *FailedState* meta-variable. The *IStreamInput* interface models the reception of input streams, with *StreamStateIn* encoding the current stream state. Similarly, the *IStreamOutput* interface models the production of stream as output, with the current stream state encoded in the *StreamStateOut* meta-variable. The multiplicity of *StreamStateIn* and *StreamStateOut* is not specified, meaning that the interface may refer to any number of streams.

The *Component* model template realizes the *IComponent* model interface, and adds specific parameters:

$$\begin{aligned}
&\texttt{Component} = ((\mathcal{I}, \Delta, O, L_T), \Psi),\\
&\mathcal{I} = \{\texttt{IComponent}\}, \quad \Delta = \{\lambda_{\texttt{Real}}, N_{\texttt{Int}}, c_{\texttt{Real}}, t_{\texttt{Real}}\},\\
&O = \{\texttt{FailedState}\}, \quad L_T = \emptyset, \quad \Psi = [\text{SAN template}],
\end{aligned} \tag{3}$$

where λ is the failure rate of the component, N is the number of spares, c and t are the coverage and delay of the failover process, respectively. The current state of the component (*FailedState*) is provided as an observation point.

The *StreamProcessing* model template represents the processing of one multimedia stream by a component of the stage architecture. It realizes all the three model interfaces:

$$\begin{aligned}
&\texttt{StreamProcessing} = ((\mathcal{I}, \Delta, O, L_T), \Psi),\\
&\mathcal{I} = \{\texttt{IComponent}, \texttt{IStreamInput}, \texttt{IStreamOutput}\}, \quad \Delta = \emptyset,\\
&O = \{\texttt{StreamStateOut}\}, \quad L_T = \{s\}, \quad \Psi = [\text{SAN template}].
\end{aligned} \tag{4}$$

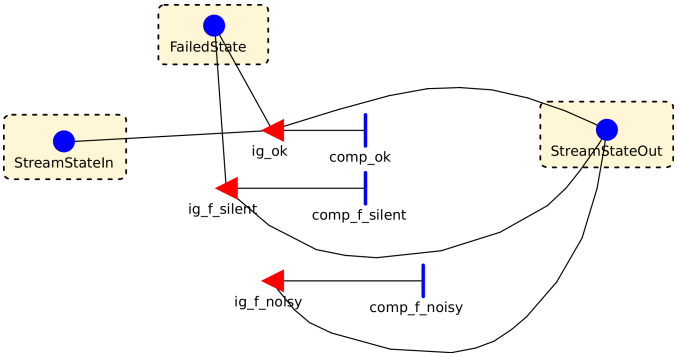

Figure 4: Implementation based on SANs of the *StreamProcessing* atomic template.

The SAN implementation of this model template is very simple (Figure 4). Basically, the current state of the stream produced as output (*StreamStateOut* from the *IStreamOutput* interface) is set based on the state of the stream received in input (*StreamStateIn* from the *IStreamInput* interface), and the current state of the component itself (*FailedState* from *IComponent*).

The *Component* and *StreamProcessing* atomic templates can be composed together using the *WONode* composite template, which represents a node of the WO stage architecture:

$$\begin{aligned} &\texttt{WONode} = ((\mathcal{I}, \Delta, O, L_T), \Psi), \\ &\mathcal{I} = \{\texttt{IStreamOutput}, \texttt{IStreamInput}\}, \\ &\Delta = \{s_{\texttt{Set\{Int\}}}\}, \quad O = \emptyset, \quad L_T = \emptyset, \quad \Psi = (\mathcal{B}, \mathcal{R}). \end{aligned} \tag{5}$$

The s parameter defines how many streams are flowing through the node, and their identifiers. The composite template has three blocks, which are formally defined as follows:

$$\begin{aligned} \mathcal{B} &= \{B_{component}, B_{streams}, B_{previousnode}\}, \\ B_{component} &= (\{\texttt{IComponent}\}, \emptyset, \{1\}), \\ B_{streams} &= \left(\left\{\begin{array}{l}\texttt{IComponent},\\ \texttt{IStreamInput},\\ \texttt{IStreamOutput}\end{array}\right\}, \emptyset, s_{\texttt{Set\{Int\}}}\right), \\ B_{previousnode} &= (\{\texttt{IStreamOutput}\}, \emptyset, \emptyset). \end{aligned} \tag{6}$$

The block $B_{component}$ represents the functional component associated to the architectural node (e.g., a workstation, a mixer), $B_{streams}$ are the streams that flow through the component, and $B_{previousnode}$ is the previous node in the architecture, i.e., another instance of the *WONode* template to which it is directly connected. Actually, it could be an instance of any model template that realizes the *IStreamOutput* model interface. $B_{component}$ has a constant multiplicity of $\{1\}$, while the multiplicity of $B_{streams}$ is given by parameter s: there is a model instance for each stream that flows through the node, and their identifiers are given by the values assigned to the set s. The multiplicity of block $B_{previousnode}$ is unspecified, meaning that there can be any number of its realizations; in fact, different streams con reach the node from different paths.

Composition rules in $\mathcal{R}$ are given in the TMDL notation:

```
1  template WONode realizes IStreamInput,IStreamOutput {
2    parameters { Set<Int> s }
3    block Component requires IComponent
4    block Streams[s]
5      requires IComponent,IStreamInput,IStreamOutput
6    block PreviousNode[] requires IStreamOutput
7    rules {
8      all FailedState { Component, Streams }
9      match StreamStateIn labels s {
10       Streams.StreamStateIn,
11       PreviousNode.StreamStateOut
12     }
13     forward StreamStateOut { Streams.StreamStateOut }
14   }
15 }
```

Lines 1–6 define the *WONode* template, its parameters, and its blocks. Three composition rules are defined. Line 8 defines an *all* rule, in which the `FailedState` interface variable of the `Component` block is composed with the one having the same name in each of the `Streams` blocks. Basically, this allows instances of the `StreamProcessing` template to access the current state of the component. Lines 9–12 specify that the `StreamStateIn` interface variables belonging to a given stream should be connected with the matching `StreamStateOut` interface variables from the previous node in the architecture. To be able to distinguish the states of different multimedia streams, a *match* rule is used, specifying that interfaces should be joined based on the identifiers associated to the "`s`" label.

Finally, line 11 specifies that the interface variables `StreamStateOut` of `Streams` models should be forwarded as interface of the composite template. This is what allows the states of the streams to be further propagated. In fact, they will have the role of `PreviousNode.StreamStateOut` (as in line 11) in another instance of the `WONode` model template.

C. Scenario Specifications

In this section we revise the scenario analyzed in [11] using the approach proposed in this paper, and show how alternative scenarios can be defined.

1) Scenario #1: The scenario consisted of a WO performance comprising three stages and five multimedia streams: audio of the orchestra (1); audio of actors (2); video of the orchestra (3); video of actors (4); and video of the director (5). Streams 1, 3, and 5 are captured in Stage A, while streams 2 and 4 are captured in Stage B. Stage C only contains the audience. All the streams are reproduced in all the three stages.

The composed model corresponding to one of the stages, Stage A, is depicted in Figure 5. Using the traditional approach, composing such model requires considerable manual effort. Each *Submodel* block is obtained by selecting a previously defined SANs model; if two submodels have the same behavior but different numerical parameters, the source model typically needs to be duplicated and modified. Then, for each *Join* block the state variables of submodels need to be carefully connected together, following the devised patterns. This has to be done recursively, starting from the bottom until reaching the root of the model, in this case the *Gateway_with_Streams* node in the top right part of the figure.

Furthermore, the one in Figure 5 is only one of the three stages that are needed in the model; the same procedure has to be repeated for the other two, which have a different structure. Then, all the three models have to be composed together. Note that this is just for the evaluation of a *single scenario*. If the architecture of a stage changes, or a different show is planned

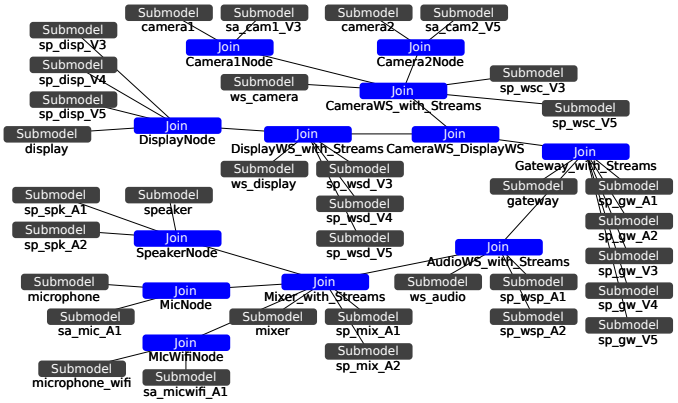

Figure 5: A SAN composed model for a WO stage, built from multiple instances of the four identified atomic templates. Figure adapted from [11].

(i.e., different streams) the model has to be composed again reflecting the different structure.

Our approach facilitates the composition of the model, which is automated, as well as the specification and update of the scenario. An excerpt of the TMDL specification corresponding to this scenario is reported in the following:

```
1  variant Camera from Component {
2    lambda=0.002, s=1, c=0.95, t=60 }
3  variant Workstation from Component {
4    lambda=1.0E-5, s=1, c=0.95, t=5 }
5  ...
6  variant Camera1Node from WONode {
7    s=(3)
8    with Camera as Component,
9    with StreamAcquiring as Streams
10 }
11 variant Camera2Node from WOnode { ... }
12 variant VideoWorkstationNode from WONode {
13   s=(3,5)
14   with Workstation as Component,
15   with StreamProcessing as Streams,
16   with Camera1Node,Camera2Node as PreviousNode
17 }
18 ...
19 variant StageA from Stage {
20   out=(1,3,5), in=(2,4)
21   with GW as GatewayNode
22 }
23 variant StageB from Stage { ... }
24 variant StageC from Stage { ... }
25 ...
26 variant root Performance3Stages from WOPerformance {
27   with StageA,StageB,StageC as Stages
28 }
```

Lines 1–4 define two variants of the `Component` model template, setting the parameters corresponding to cameras and workstations. Lines 6–10 define a node of the WO architecture based on the `WONode` template. As specified by the `s` parameter, the node will handle stream number 3, i.e., the video of the orchestra, and will use an instance of the `Camera` variant as its `Component` block. A similar structure is defined in line 11 for another camera; in this case the involved stream is stream number 5 (video of the director).

Lines 12–17 define another variant of the `WONode` template, in this case representing a video workstation node. This node handles streams 3 and 5, and uses `Camera1Node` and `Camera2Node` to fill `PreviousNode` block. This means that the workstation receives the video streams from the two cameras. Lines 19–24, define variants for each of the involved stages, identifying which streams are received or transmitted

```
1  variant VideoMixer from Component { ... }
2  variant VideoMixerNode from WONode {
3    s=(3,5)
4    with VideoMixer as Component,
5    with StreamProcessing as Streams,
6    with Camera1Node, Camera2Node as PreviousNode
7  }
8  variant VideoWorkstationNode from WONode {
9    s=(3,5)
10   with Workstation as Component,
11   with StreamProcessing as Streams,
12   with VideoMixerNode as PreviousNode
13 }
```

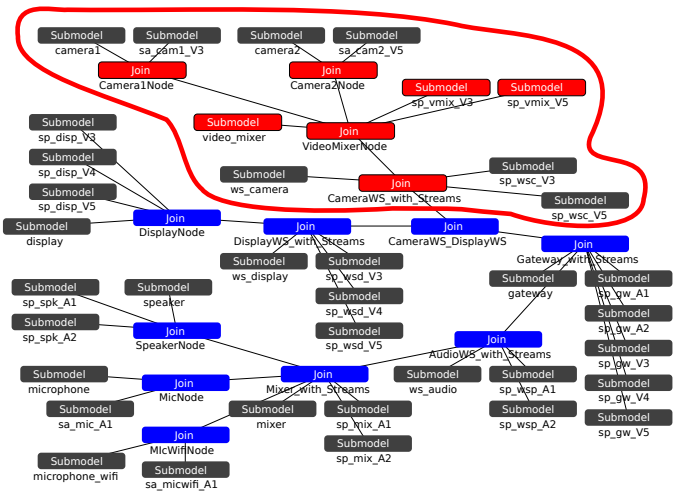

Figure 6: Modification of the base scenario to add a video mixer.

by the stage, and the model variant that represents the gateway. Finally, lines 26–28 define the root variant that represents the scenario.

2) Scenario #2: Let us now suppose that the architecture of Stage A changes, and that a pre-processing step is added before sending the video streams to the workstation; this is performed by a video mixer, which synchronizes the two streams and possibly applies effects like closed captions. Using the TMDL this can be modeled by simply modifying the `Scenario` specification as shown in the listing in Figure 6. Another variant of `WONode` representing the mixer has been added, using `Camera1Node` and `Camera2Node` as its `PreviousNode` block. The `VideoWorkstationNode` is modified to use this newly created variant as `PreviousNode`.

The lower part of the figure highlights the modifications that would be needed if changes were manually applied on the model. Note that each connection (*Join* nodes) that is changed involves selecting and connecting with the proper pattern the interface variables of submodels.

3) Scenario #3: Further, we can imagine adding another multimedia stream, Stream 6, which needs to be acquired by another camera. Suppose the camera is from a different vendor, thus having different parameters. In this case, it is sufficient to create a new variant of the `Component` and `WONode` templates, and use the latter to fill the `PreviousNode` block for the node corresponding to the mixer (TMDL listing in Figure 7).

Even these simple operations, if performed manually, would

```
variant NewCamera from Component { ... }
variant NewCameraNode from WONode {
   s=(6)
   with NewCamera as Component,
   with StreamProcessing as Streams
}
variant VideoMixerNode from WONode {
   s=(3,5,6)
   with VideoMixer as Component,
   with StreamProcessing as Streams,
   with Camera1Node,Camera2Node,NewCameraNode as
        PreviousNode
}
```

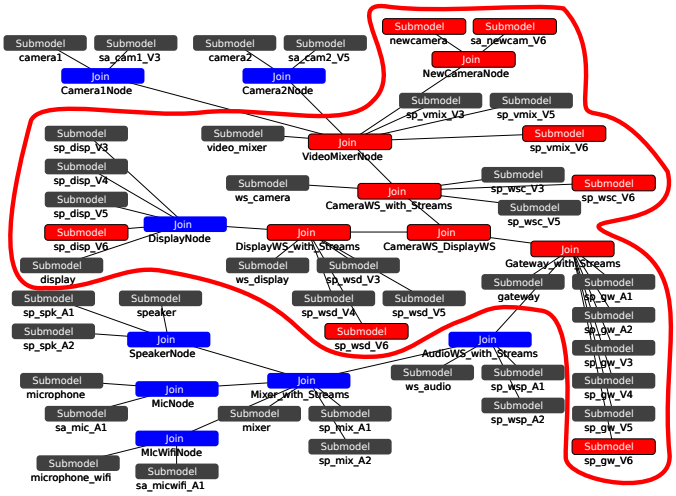

Figure 7: Modification of the base scenario to add another camera, which is used to capture a new video stream.

require considerable effort for the modeler. Manually performing the same modifications would require the following steps: i) duplicating the atomic model for the camera, modifying the parameters as needed; ii) adding an instance of the new camera node to the composed model; and iii) properly connecting the interfaces of the new model across the involved *Join* nodes. As shown in Figure 7 this would lead to modify the shared variables within all the *Join* nodes until the root of the overall model, a process which would also be prone to human errors.

D. Specification of Metrics

Reward structures and variables are then defined as part of the model evaluation process. In fact, different reward variables can be defined on the same model, based on the desired metrics and available observation points. Furthermore, the specification of reward variables is closely related to the evaluation method, e.g., steady state or transient evaluation, time points, accuracy, etc. For this reason, we decided not to provide constructs for defining reward variables [47] directly in the TMDL. Instead, we only provide means for identifying "interesting" states of the state space, through the concept of observation points. In this way, we leave flexibility on the kind of metrics that can be defined, while keeping TMDL simple.

In the case study in this section, the composed model generated by the framework will have an observation point `StreamStateOut` for each instance of `StreamProcessing` template. Such state variable represents the state of the stream at a given point of the processing chain. Quantitative metrics can be defined by defining proper reward variables over those observation points.

Referring to the previously defined metrics $R^j(T)$ and $A^j(0,T)$, they can be evaluated by defining, for each observation point $\texttt{StreamStateOut}^j$, a rate reward function $\mathcal{R}^j$ such that:

$$\mathcal{R}^j(v) = \begin{cases} 1 & \text{if } v = \{(\texttt{StreamStateOut}^j, 1)\} \\ 0 & \text{otherwise,} \end{cases}$$

and then evaluate the reward functions corresponding to reliability and interval-of-time availability, as in [47]. The actual evaluation of a wide range of metrics for the WO system can be found in [11].

E. Approach Evaluation

To quantify the benefits provided by our approach, we compare the number of *actions* that the user needs to perform when using the manual approach, and when using the proposed framework. Because they involve different kinds of task, it is somewhat difficult to precisely compare the two approaches. To perform a fair comparison, we selected actions that are at a comparable level of abstraction.

For the "Manual" approach, we considered a user creating a model with the Möbius [12] tool, based on preexisting atomic models. We identified the following basic actions: i) *duplicating a SAN atomic model*; ii) *modifying*[2] *an element in an atomic SAN model, e.g., adding a place or changing the name of a variable*; iii) *modifying a node in a Rep/Join model*; and iv) *modifying a shared state variable in a Rep/Join node*. For the "TMDL" approach proposed in this paper we instead consider only: v) *modifying an element in a TMDL scenario specification*, as the whole specification process is performed using the TMDL. With "element" we mean here an instance of a metaclass of the metamodel.

The number of actions required to construct the WO models presented in this section is reported in Table I. We analyzed the effort required to create specifications for the three scenarios presented in this section, as well as the effort to change a scenario into another. As illustrated in the table, the proposed approach reduces the number of actions in all the analyzed tasks. As expected, the greatest gain (50% action reduction) is obtained when modifying existing scenarios, while the difference is smaller when creating new scenarios from scratch. We note that this comparison is very conservative, as actions for the "TMDL" approach are typically performed much faster. In fact, they simply consist in changing some words in a textual specification. Referring to Figure 7, just modifying the text highlighted in the listing counts as 11 actions (2 `Variant`, 6 `Assignment`, and 3 `BlockRealization` elements need to be added or modified).

On the other hand, a single action in the "Manual" approach typically consists of several sub-tasks. For example, *"modify a shared state variable in the Rep/Join model"* consists at least of i) selecting the node in the Rep/Join model, ii) selecting the variable to be modified, iii) modifying the variable, iv) clicking on a confirmation button. It seems thus reasonable to

[2]We consider creation and deletion as special cases of modification.

Table I: Number of *actions* required by the user to specify the WO model of Stage 1 for different scenarios. Comparison between using a traditional manual approach and the proposed framework.

	Manual	TMDL	Reduction
Create Scenario #1	141	85	39.7%
Create Scenario #2	155	99	36.1%
Create Scenario #3	183	109	40.4%
Modify Scenario #1 to Scenario #2	18	11	38.9%
Modify Scenario #1 to Scenario #3	46	23	50.0%
Modify Scenario #2 to Scenario #3	28	14	50.0%

expect that the time saved in using the proposed approach will be much higher. Concerning performance, the execution time needed of the automated composition algorithm is of the order of magnitude of seconds. It is thus negligible with respect to the time required to actually simulate such kind of models, typically hours or days.

Finally, it should be noted that many correctness checks are performed by default by the Ecore framework [49], which prevents users from specifying incorrect compositions. In particular, it warns the user if the model under specification (*library* or *scenario*) does not conform to the TMDL metamodel (Figure 3). Ecore has also a validation framework, which can be used to check additional constraints. For example, requiring that the value assigned to a parameter is compatible with its type, or that only state variables of the same type are connected together.

VII. Conclusions and Future Work

In this paper we have proposed an approach for the automatic assembly of complex state-based models, based on the concepts of model templates libraries and composition rules. We have formally defined all the concepts involved in the framework, and provided a DSL that can be used to concretely specify them. This language, called TMDL, allows libraries of template models to be precisely defined, and then applied to specify different system configurations and compose the concrete analysis model.

This paper introduces the formalization of the proposed framework. In its concrete application, users will be supported by editors and tools in all the phases; we thus believe that improvements over manual practice will be much greater. Tools under development include means to: i) edit template implementations using graphical abstractions, ii) write textual TMDL specifications, iii) write TMDL specifications using graphical abstractions, and iv) define commonly used metrics using predefined reward variables.

A current limitation of the proposed approach is that it does not provide specific countermeasures to the state-space explosion problem, as it focuses on the *specification* aspect only. As such, generated models would be most likely need to be solved by discrete-event simulation. However, many techniques for the exact evaluation of complex performability models exist in the literature, e.g., [1], [19]. Future work will investigate the possibility to integrate some of these techniques directly in the model composition algorithm, based on the characteristics of the involved model templates.

Another possible improvement consists in providing better mechanisms for the specification of metrics. Solution to be explored consist in i) the definition of metrics using probabilistic temporal logics (e.g., [50]), or ii) the adoption of observation patterns as in [28] as specific kinds of model templates. In the context of MDE, techniques like code generation and model-weaving [51] are also promising: they can be exploited to generate TMDL specifications from system descriptions in other forms, e.g., UML models or structured textual files.

Acknowledgment

This work has been partially supported by the REGIONE TOSCANA POR FESR 2014-2020 SISTER "SIgnaling & Sensing Technologies in Railway application", and by the DEVASSES (DEsign, Verification and VAlidation of large-scale, dynamic Service SystEmS) project, funded by European Union's Seventh Framework Programme under grant agreement PIRSES-GA-2013-612569. This work is related to the activities of the H2020 MSCA-RISE-2018 project ADVANCE "Addressing Verification and Validation Challenges in Future Cyber-Physical Systems".

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] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[3] J. Meyer, "On evaluating the performability of degradable computing systems," *IEEE Transactions on Computers*, vol. C-29, no. 8, pp. 720–731, 1980.

[4] C. McLean, Y. T. Lee, S. Jain, and C. Hutchings, "Modeling and Simulation of Critical Infrastructure Systems for Homeland Security Applications," U.S. National Institute of Standard and Technology (NIST), Tech. Rep. NISTIR 7785, September 2011.

[5] G. Ciardo, R. German, and C. Lindemann, "A characterization of the stochastic process underlying a stochastic petri net," *Software Engineering, IEEE Transactions on*, vol. 20, no. 7, pp. 506–515, 1994.

[6] M. Jamshidi, Ed., *Systems of Systems Engineering – Innovations for the 21st Century*. Wiley, 2009.

[7] A. Bondavalli, S. Bouchenak, and H. Kopetz, Eds., *Cyber-Physical Systems of Systems – Foundations – A Conceptual Model and Some Derivations: The AMADEOS Legacy*, ser. Programming and Software Engineering. Springer International Publishing, 2016, vol. 10099.

[8] L. Montecchi, P. Lollini, and A. Bondavalli, "A DSL-Supported Workflow for the Automated Assembly of Large Performability Models," in *Proceedings of the 10th European Dependable Computing Conference*, Newcastle upon Tyne, UK, May 13-16, 2014.

[9] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: Structure, behavior, and application," in *International Workshop on Timed Petri Nets*, Turin, Italy, July 1-3, 1985, pp. 106–115.

[10] W. Sanders and J. Meyer, "Stochastic activity networks: formal definitions and concepts," in *Lectures on formal methods and performance analysis*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2002, vol. 2090, pp. 315–343.

[11] N. R. Veeraragavan, L. Montecchi, N. Nostro, R. Vitenberg, H. Meling, and A. Bondavalli, "Modeling QoE in Dependable Tele-Immersive Applications: A Case Study of World Opera," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2667–2681, 2016.

[12] T. Courtney, S. Gaonkar, K. Keefe, E. W. D. Rozier, and W. H. Sanders, "Möbius 2.3: An extensible tool for dependability, security, and performance evaluation of large and complex system models," in *39th IEEE/IFIP International Conference on Dependable Systems Networks*, Estoril, Portugal, 2009, pp. 353–358.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides., *Design Patterns*. Addison-Wesley, 1995.

[14] H. Kopetz and N. Suri, "Compositional design of RT systems: a conceptual basis for specification of linking interfaces," in *6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Hokkaido, Japan, May 16, 2003, pp. 51–60.

[15] S. Bliudze and J. Sifakis, "The Algebra of Connectors – Structuring Interaction in BIP," *IEEE Transactions on Computers*, vol. 57, no. 10, pp. 1315–1330, 2008.

[16] 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.

[17] B. Plateau and K. Atif, "Stochastic automata network for modeling parallel systems," *IEEE Transactions on Software Engineering*, vol. 17, no. 10, pp. 1093–1108, 1991.

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

[19] P. Lollini, A. Bondavalli, and F. Di Giandomenico, "A decomposition-based modeling framework for complex systems," *IEEE Transactions on Reliability*, vol. 58, no. 1, pp. 20–33, 2009.

[20] G. Ciardo, Y. Zhao, and X. Jin, "Ten Years of Saturation: A Petri Net Perspective," in *Transactions on Petri Nets and Other Models of Concurrency V*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 6900, pp. 51–95.

[21] K. Kanoun, M. Borrel, T. Morteveille, and A. Peytavin, "Availability of CAUTRA, a subset of the French air traffic control system," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 528–535, 1999.

[22] K. Kanoun and M. Ortalo-Borrel, "Fault-tolerant system dependability-explicit modeling of hardware and software component-interactions," *IEEE Transactions on Reliability*, vol. 49, no. 4, pp. 363–376, 2000.

[23] M. Rabah and K. Kanoun, "Performability evaluation of multipurpose multiprocessor systems: the "separation of concerns" approach," *IEEE Transactions on Computers*, vol. 52, no. 2, pp. 223–236, 2003.

[24] A. Bondavalli, P. Lollini, and L. Montecchi, "QoS Perceived by Users of Ubiquitous UMTS: Compositional Models and Thorough Analysis," *Journal of Software*, vol. 4, no. 7, 2009.

[25] E. Battista, V. Casola, N. Mazzocca, R. Nardone, and S. Marrone, "A compositional modelling approach for large sensor networks design," in *8th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, Compiegne, France, October 28-30, 2013, pp. 422–429.

[26] S. Chiaradonna, F. D. Giandomenico, and G. Masetti, "A stochastic modelling framework to analyze smart grids control strategies," in *2016 IEEE Smart Energy Grid Engineering (SEGE)*, Oshawa, ON, Canada, August 21-24, 2016, pp. 123–130.

[27] G. Nencioni, B. E. Helvik, and P. E. Heegaard, "Including Failure Correlation in Availability Modeling of a Software-Defined Backbone Network," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 1032–1045, 2017.

[28] N. Ge, M. Pantel, and S. D. Zilio, "Formal Verification of User-Level Real-Time Property Patterns," in *11th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, Sophia Antipolis, France, September 13-15, 2017, pp. 1–8.

[29] E. Best, R. Devillers, and M. Koutny, "The box algebra — a model of nets and process expressions," in *Application and Theory of Petri Nets 1999. ICATPN 1999*, ser. Lecture Notes in Computer Science, Williamsburg, Virginia, USA, June 21-25, 1999, pp. 344–363.

[30] K. Jensen and L. Kristensen, *Coloured Petri Nets — Modelling and Validation of Concurrent Systems*. Springer Berlin Heidelberg, 2009.

[31] J. K. Muppala, G. Ciardo, and K. S. Trivedi, "Stochastic reward nets for reliability prediction," in *Communications in Reliability, Maintainability and Serviceability*, vol. 1, no. 2, 1994, pp. 9–20.

[32] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[33] S. Bernardi, J. Merseguer, and D. C. Petriu, "Dependability modeling and analysis of software systems specified with UML," *ACM Computing Surveys*, vol. 45, no. 1, 2012.

[34] M. Cinque, D. Cotroneo, and C. Di Martino, "Automated generation of performance and dependability models for the assessment of wireless sensor networks," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 870–884, 2012.

[35] L. Montecchi, P. Lollini, and A. Bondavalli, "Towards a MDE Transformation Workflow for Dependability Analysis," in *16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, Las Vegas, USA, 2011, pp. 157–166.

[36] S. Bernardi, S. Marrone, J. Merseguer, R. Nardone, and V. Vittorini, "Towards a model-driven engineering approach for the assessment of non-functional properties using multi-formalism," *Software & Systems Modeling*, 2018.

[37] G. Masetti, S. Chiaradonna, and F. D. Giandomenico, "A Stochastic Modeling Approach for an Efficient Dependability Evaluation of Large Systems with Non-anonymous Interconnected Components," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, Toulouse, France, October 23-26, 2017, pp. 46–55.

[38] C. Atkinson and T. Kühne, "Model-Driven Development: A Metamodeling Foundation," *IEEE Software*, vol. 20, no. 5, pp. 36–41, 2003.

[39] A. Zimmermann, *Stochastic Discrete Event Systems — Modeling, Evaluation, Applications*. Springer Berlin Heidelberg, 2008.

[40] O. Hauge *et al.*, "Common Variability Language (CVL)," OMG Revised Submission, August 2012, OMG document: ad/2012-08-05.

[41] ISO/IEC 15909-2:2011, "Systems and software engineering – High-level Petri nets – Part 2: Transfer format," February 2011.

[42] Object Management Group (OMG), "XML Metadata Interchange (XMI) Specification," formal/2014-04-14, April 2014, version 2.4.2.

[43] ——, "OMG Meta Object Facility (MOF) Core Specification," formal/2016-11-01, November 2016, version 2.5.1.

[44] U. Aßmann, *Invasive Software Composition*. Springer-Verlag Berlin Heidelberg, 2003.

[45] ISO/IEC 15909-1:2004, "Systems and software engineering – High-level Petri nets – Part 1: Concepts, definitions and graphical notation," December 2004.

[46] L. Montecchi, P. Lollini, and A. Bondavalli, "Stochastic Activity Networks Templates," Resilient Computing Lab, Tech. Rep. RCL180401, v1.0, April 2018, http://rcl.dsi.unifi.it/publication/show/849.

[47] W. H. Sanders and J. F. Meyer, "A Unified Approach for Specifying Measures of Performance, Dependability, and Performability," in *Dependable Computing for Critical Applications*. Springer, Vienna, 1991, vol. 4, pp. 215–237.

[48] M. Voelter, *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, January 2013.

[49] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, 2009.

[50] S. Donatelli, S. Haddad, and J. Sproston, "Model Checking Timed and Stochastic Properties with CSL^{TA}," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 224–240, 2009.

[51] R. Hawkins, I. Habli, D. Kolovos, R. Paige, and T. Kelly, "Weaving an Assurance Case from Design: A Model-Based Approach," in *IEEE 16th International Symposium on High Assurance Systems Engineering (HASE'15)*, Daytona Beach Shores, FL, USA, January 8-10, 2015, pp. 110–117.