A DSL-Supported Workflow for the Automated Assembly of Large Stochastic Models

Leonardo Montecchi, Paolo Lollini, Andrea Bondavalli

Dipartimento di Matematica e Informatica, University of Firenze
Viale Morgagni, 65 – I-50134 Firenze, Italy
{lmontecchi,lollini,bondavalli}@unifi.it

***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 improve the maintanability of models. Approaches based on Stochastic Petri Nets often compose submodels by state-sharing, following predefined "patterns", depending on the scenario of interest. However, such composition patterns are typically not formalized. Clearly defining libraries of reusable submodels, together with valid patterns for their composition, would allow complex models to be automatically assembled, based on a high-level description of the scenario to be evaluated. The contribution of this paper to this problem is twofold: on one hand we describe our workflow for the automated generation of large performability models, on the other hand we introduce the TMDL language, a DSL to concretely support the workflow. After introducing the approach and the language, we detail their implementation within the Eclipse modeling platform, and briefly show its usage through an example.**

***Keywords*-modularity; model-based evaluation; state-based; performability; template models; composition; model-driven engineering.**

I. Introduction

Model-based evaluation [28] plays a key role in dependability [4] and performability [25] evaluation of systems. Modeling allows the system to be analyzed at different levels of abstraction, can be used to perform sensitivity analyses, identify system bottlenecks, highlight problems in the design, guide experimental activities and provide answers to "what-if" questions. Most importantly, model-based analysis enables the evaluation of specific events without their actual occurrence in the real system. For this reason, modeling and simulation are widely used in the assessment of high-integrity systems and infrastructures, where the evaluation of the effects of faults and attacks can potentially lead to catastrophic consequences.

While ad-hoc simulators are used in some domains, (e.g., see [29]), state-based formalisms like Stochastic Petri Nets (SPNs) and their extensions [12] are widely used to assess non-functional properties across different domains. The use of such formalisms has several key advantages: they provide a convenient graphical notation, support different abstraction levels, enable modular modeling via state-sharing formalisms [33], and they are well-suited in 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 [9]. The high number of components, their interactions, and rapidly changing system configurations represent notable challenges for the application of model-based evaluation approaches. A large amount of work in literature propose techniques for the generation, handling, and numerical evaluation of large state-space models, which can be grouped in largeness avoidance and largeness tolerance approaches [28]. While techniques for the efficient evaluation of large models are fundamental, the growing complexity of modern systems is also posing challenges for the *specification* of analysis models itself.

A key principle in addressing the complexity in the specification of analysis models for large systems and infrastructures is *modularization*. When using approaches based on SPNs and their extensions, 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 maintanability of the obtained analysis model is therefore improved: submodels can be modified in isolation from the rest of the model, can be substituted with more refined implementations, can be rearranged based on modifications in system configuration.

However, while submodels can be precisely defined using well-established formalisms (e.g., SPNs), patterns for their composition are typically not formalized. As a result i) submodels libraries are difficult to be shared and reused, and ii) the overall system model for different scenarios must be assembled by hand by people who know the appropriate rules to follow. Moreover, even when rules for the composition of submodels have been properly specified, obtaining a *valid* (i.e., correctly assembled) model requires a lot of manual effort, involving error-prone, time-consuming, and repetitive tasks.

In this paper we define a workflow that addresses this problem, with the ultimate goal to support the automated assembly of large performability models, based on well-specified libraries of reusable submodels. The workflow is built around TMDL, a Domain-Specific Language (DSL) that is used to precisely specify and instantiate libraries of "template"

performability models.

The paper is organized as follows. The necessary background and motivation for our work are introduced in Section II, while a motivating example which can benefit of our approach is introduced in Section III. An overview of our workflow for the automated assembly of large performability models is provided in Section IV, while Section V details the TMDL language and sketches the automated model assembly algorithm. Section VI introduces the current implementation and provides an overview of its application to the previously introduced example. Conclusions are drawn in Section VII.

II. Related Work

The need of modularity and composition for tackling the complexity of modern systems has emerged in several engineering domains. The one in which this aspect is most evident is perhaps software engineering, where techniques for properly constructing a software system out of modular elements are well established [18, 24]. More in general, a wealth of methodologies and formalisms have been proposed in literature to address composability, i.e., the construction of a whole system from parts (components); an interesting survey can be found in [17]. Several work specifically addressed composability for high-integrity and real-time systems, by defining formal models for specifying interactions between components, e.g. see [7, 21].

Our focus is not on component-based system development, but rather on the modular construction of models for stochastic analysis by means of reusable elements. It should be noted that compositional modeling has also been the subject of much work present in literature; a nice discussion on this topic can be found in [28]. Many approaches exist that apply compositional modeling to take advantages of symmetries existing in the model, thus reducing the space or the time required for its numerical evaluation.

Although techniques for efficient analysis of performability models are of paramount importance, they are not the purpose for which – in this paper – we are looking at compositional modeling approaches. Indeed, while compositional modeling approaches (e.g., [30, 33]) were initially introduced for the purpose of reducing the size of the generated state-space, such approaches have later gained importance also in improving the *specification* of models, since they also carry with them a number of other practical advantages: submodels are usually simpler to be managed, can be reused, can be refined, and can be modified in isolation from other parts of the model.

Several approaches based on Petri nets and their extensions have recognized the benefits in applying "separation of concerns" principles [15] to the construction of performability analysis models. In such approaches (e.g., see [10, 20, 31, 37]) the overall analysis model is built out of well-defined submodels addressing specific aspects of the systems, which are then composed following predefined rules based on the actual scenario to be represented. When submodels are reused multiple times, this approach leads to a modeling paradigm that resembles object-oriented programming (OOP): libraries of "template" submodels are created for a given system, having fixed "interfaces" and "parameters". Such templates are then "instantiated" multiple times and then connected through their interfaces. Indeed, when referring to submodels and their properties, some of these works actually use OOP-derived terms like *interface* [31], *template*, *instance* [8] and even *inheritance* [5].

The main gap in applying this approach is that, while established formalisms exist both for defining the submodels and for physically composing them, the patterns to be followed for their composition – which depend on the system to be modeled, and the set of identified submodels – are typically not formalized. In many cases, submodels are based on SPNs and they are composed by state-sharing, but composition patterns are provided either informally or by examples. Even worse, those "rules" often are not even written somewhere, but they are only known to the person(s) that developed the submodel library for the system under analysis. For this reason, reusing such template models and composition patterns, and sharing them between different teams is currently impracticable.

In this paper we address exactly this problem, by defining a workflow which provides support for defining libraries of reusable submodels, *including library-specific patterns for their composition*. The ability to precisely define submodel composition patterns allows the overall performability model to be automatically assembled via model-transformation from a high-level specification of the scenario of interest. While automated assembly of complex models is useful per-se, it also provide a means for easier adaptation of performability analysis model to modifications of system configuration.

It should be noted that several other work in literature apply Model-Driven Engineering (MDE) [35] techniques for the automatic derivation of performance and dependability models, e.g., see [6, 13, 23, 26, 32]. However, the purpose of such approaches is usually to provide an application-specific abstraction to users of a certain domain, and then automatically derive analysis models defined by a domain expert. Reuse across different domains, or with different submodel libraries is not considered, and therefore composition patterns are typically embedded in the transformation algorithm, which is therefore different for every different library of submodels.

In our approach the definition of composition patterns is part of the submodel library, and it is therefore separated from the model generation algorithm. In this way, the model generation algorithm is defined (and implemented) *only once*, and it can be used to generate and assemble models from different model libraries. This of course requires adding some "intelligence" (and complexity) directly in the submodel library.

Within the performability community, the approach which is more related to ours is the one proposed by the OsMoSys framework [39]. In particular, it also promotes an Object-Oriented modeling approach, using OO-derived terms like "model class", which have strong analogies with terms adopted in this paper. The focus, and consequently the approach, is however quite different. OsMoSys provides a way to compose performability models created with different formalisms, and

to orchestrate their solution in order to evaluate the global system model. In this paper we focus on the *reuse* and *automation* of composition patterns for a specific class of formalisms, aiming at enabling the automatic assembly of large performability models with reduced effort for the user.

Finally, it should be noted that the Möbius framework [14] provides some means for reducing the effort required to specify complex models. Actually, its implementation of the Rep/Join composition formalism [33], allows multiple instances of the same Stochastic Activity Networks (SAN) [34] model to be used. However, instances of the same submodel are completely identical, and each instance still needs to be manually connected to the rest of the composed model.

III. Motivating Example

To better illustrate the approach, we introduce a motivating example of a large-scale distributed system, and show how the proposed approach improves the manageability and scalability of the analysis model.

A. A World Opera

The system we are considering as motivating example is the one envisioned by the World Opera (WO) consortium [36], aiming at conducting distributed, real-time, live opera performances across several world renowned opera houses. Each opera house represents a real-world stage with its own musicians, singers, dancers, and actors. Participating artists from different real-world stages are mapped to *virtual-world stages*, which are projected as video on display devices, and shown to the audience at the local opera house as well as audiences at geographically distributed (remote) opera houses. For example, the "Gilgamesh" performance, includes three real-world stages in Norway, Sweden, and Denmark, respectively. Additionally, virtual-world stages can display animated cartoon characters mimicking the behavior of the artists at remote stages [38].

The WO application exhibits several challenges, including synchronization across several dimensions: spatial, temporal, and precedence. Moreover, the infrastructure allowing for these applications includes a high number of specialized hardware and software components, whose slight malfunction could severely affect the performance, due to the strict functional requirements. Fault-tolerant architectural solutions are therefore necessary to ensure the correct execution of a WO performance. To design such a fault-tolerant solution it is essential to understand the interaction between the components and the potential effects of their failure, from the audience perspective, on the overall show.

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.). The activities at each stage in WO are logically divided into four tasks. *Capturing* involves corresponding components receiving activation signals and generating streams. There exist three principal stream types: video, audio and sensor (e.g., to track the movement of an artist on the stage). These

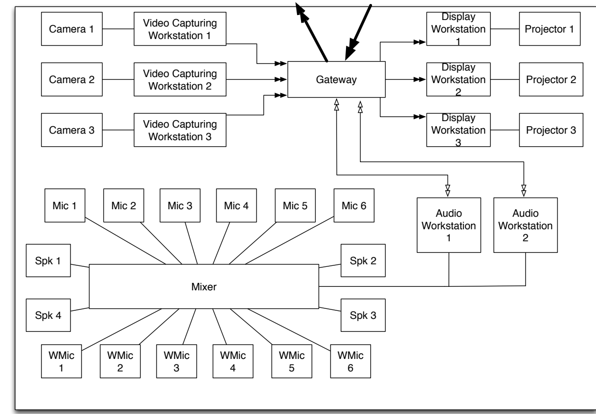

Figure 1. System architecture of a World Opera stage [37, 38]

generated streams collectively represent the real-world data. *Processing* is then performed on all generated streams to remove noise. Additionally, video streams are encoded to reduce the size of streams, timestamped, and processed using computer vision techniques for artistic reasons. *Streaming*, involves transmitting and receiving the streams to and from the remote stages. Finally, *rendering* involves processing (e.g., decoding) received streams, synchronizing them based on their timestamps and then rendering them to the virtual-world.

The architecture considered for a general WO stage is shown in Figure 1. A stage consists of: microphones and cameras to capture the multimedia streams from actors; projectors and speakers to render the streams to the audience; a mixer to route audio streams; workstations for processing of captured streams; a gateway for transmission of streams to/from remote stages. The actual number and kinds of components present in each stage depend on the number of artists present in the stage, and the role of the stage in the overall performance. In order to cope with faults that may affect stage components, most of them are required to implement some kind of fault-tolerance mechanism.

B. Performability Model

Model-based analysis of the WO system is being undertaken to: i) evaluate the impact of components failures, ii) compare the benefits in introducing different fault-tolerance solutions, and finally iii) evaluate the QoS perceived by users during a WO show under different configurations.

A set of metrics for the evaluation of the WO system have been defined in [37], and evaluated on a simplified case study. A more complex and faithful model of the system is being developed using a modular modeling approach as discussed before, in which the model is subdivided in well-specified "template" submodels, which are then instantiated multiple times and connected together following precise rules [27].

The model 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 are capturing it). The state of different streams as they are reproduced to the audience provides insights on the QoS perceived by the users and it is therefore the target of evaluation.

In the modeling of the WO system 4 basic SAN templates are involved [27]:

- *Component*: A physical component of the WO architecture. The interfaces of this submodel are `working_state`, providing the current state of the component, and `num_f` for each failure mode f, counting the number of components in the same group that have experienced failure mode f.
- *StreamCollector*: Models the capturing of a stream. Its interfaces are `num_f` for each failure mode f of associated capturing components, and `stream_out`, which represents the state of the captured stream.
- *StreamAdapter*: Models the processing of a stream. Its interfaces are `component_state`, representing the state of the associated processing components, `stream_in`, representing the state of the stream received as input, and `stream_out`, representing the state of the stream produced as output.
- *StreamPlayer*: Models the playback of a stream to the audience. Its interfaces are the state of the stream as received as input, `stream_in`, the state of associated playback components, `num_f`, and the state of the stream as reproduced to the audience, `stream_play`.

These templates have also a number of parameters, e.g., the failure rate and the number of spares in case of components. The actual implementation of these models depends on the behavior to be modeled; different implementations may for example be needed for representing different fault-tolerance mechanisms. For simplicity, we have assumed a hot-spare behavior for all components as in [37], thus leading to a single "Component" template. In case of components with different behavior, different templates would be needed. For example, "ComponentWithSpare" and "ComponentTMR" for distinguishing between components with hot-spare and those implementing triple modular redundancy.

By just composing multiple instances of these basic building blocks, a wide variety of different scenarios can be assessed. As an example, Figure 2 shows the composed model for a WO stage consisting of 6 microphones, 2 cameras, 4 speakers, 2 projectors, 3 workstations, 1 mixer and 1 gateway, and employed in a WO show consisting of 5 application streams. The composed model in Figure 2 consists of 33 atomic model instances, created from the fixed set of 4 templates listed above. Actually, such model is only a part of a bigger model addressing a three-stage WO performance, comprising ~50 components, modeled by ~100 instances of the 4 template models described above.

By changing the way in which such templates are arranged, or adding (or removing) specific instances, it is possible to assess different scenarios in an efficient way. As an example, it could be interesting to evaluate how the target metrics change if the workstations that are in charge of processing each stream are changed, of if different combinations of components are used to reproduce multimedia streams.

C. Current Limitations

Currently, the applicability of this approach is hampered by two major practical limitations: i) the lack of formalization of template composition patterns, and consequently ii) the lack of means for automated application of such patterns.

As described above, the overall system model is obtained by composing instances of template atomic models following *predefined patterns*. Such patterns are specifically tailored to the library of template models that is being developed: they can be considered a part of the library itself. However, such rules are not formally specified: they are part of the expertise of the person(s) that developed the model library. For this reason, sharing such libraries and reusing them across different projects becomes nearly impossible. Examples of rules for creating instances in the WO model are:

- *For each application stream to be modeled create an instance of the "StreamCollector" template model*
- *For each component of the stage create an instance of the "Component" template model, and an instance of the "StreamAdapter" template model for each of the streams the component processes.*

Unfortunately, it is not sufficient to simply add atomic model instances to the overall model. Depending on the semantic of the defined templates it is required to properly connect them based on: i) their predefined interfaces, and ii) the specific scenario that should be modeled. Referring to Figure 2, for each "Rep" or "Join" node (red and blue nodes, respectively) there exist a precise pattern to be followed to correctly assemble all the instances of atomic templates (black nodes). Manually performing such a task requires considerable effort at the increasing of model largeness. Examples of rules for connecting instances in the WO model are:

- *Connect each "StreamCollector" instance with the instances of "Component" templates corresponding to components that are used to capture the stream. Interfaces to be connected are `num_f` in all models.*
- *Connect "StreamAdapter" instances with the associated instance of the "Component" template. Interfaces to be connected are `working_state` and `component_state`.*

For large systems actually remembering or following such patterns is difficult; even more after modifications occur in system configuration. For example, the highlighted part of Figure 2 models the playback of three video streams (*v_orchestra*, *v_director*, *v_scene*) on two different projectors (*projector_orchestra* and *projector_scene*). Adding a third projector dedicated to the *v_director* stream only would require to: i) add a new instance of the "Component" template for modeling the projector, ii) removing the corresponding instance *s1_play_v_director* of the "StreamPlayer" model from the "Projector_Orchestra_and_Director" Join composition, iii)

Figure 2. A SAN composed model for a WO stage, built out of the 4 SAN templates described in Section III-B [27].

Figure 3. A composition system allows components to be assembled based on provided composition programs.

properly connect the new "Component" instance and the existing *s1_play_v_director* instance into a new Join composition, and iv) properly connect the new Join as child of the "DisplayWS_with_Streams" Join.

The approach we propose in this paper tackle these challenges by introducing: i) a custom language to precisely specify template models libraries and composition patterns, and ii) automated model-transformations to automatically instantiate the model templates and properly connect them.

IV. Approach Overview

In this section we provide an overview of the proposed approach, which defines a workflow (Section IV-B) for the automated assembly of performability models for large-scale systems. The approach is loosely based on the notion of "composition system", briefly introduced in Section IV-A.

A. Composition Systems

According to [2, 19], a *composition system* is defined by three elements: i) composition technique, ii) component model, and iii) composition language. The *composition technique* defines how components are physically connected. The *component model* defines what a component is and how it can be accessed. The *composition language* is used to specify "composition programs", i.e., a kind of script that specifies which components should be connected, and how, in order to obtain the intended composed model (Figure 3).

Reformulating our previous discussion, in order to realize our target "composition system", languages to specify components exist (SPNs and their extensions), as well as the composition technique (Rep/Join and state-sharing formalisms in general). What is currently missing is a composition language capable to specify how submodels should be assembled. More details are provided in the following.

- *Component Model.* In our system, components are models in some state-space stochastic formalism (e.g., Markov Chains, Stochastic Petri Nets). The interfaces that a component may expose to other components are the state-variables which define its state-space. In particular, we consider SANs models as components, exposing specific places as their interfaces.
- *Composition Technique.* As composition technique we consider state-sharing. When specific interfaces of two or more components are connected together, the associated state-variables are shared between the components, i.e., they are constrained to have the same value. In particular, we consider the Rep/Join formalism as state-sharing formalism.
- *Composition Language.* The composition language is the distinguishing contribution of our approach. We define the Template Models Description Language (TMDL), an ad-hoc language to support our approach. TMDL, which is described in Section V, enables the specification of i) components and their interfaces, and ii) composition programs.

B. The Workflow

The overall workflow proposed by our approach is depicted in Figure 4 and it described in the following.

Step 1. Starting from system requirements and architecture, an expert in performability modeling develops a "Template Models Library", i.e., a library of reusable components and composition patterns. Such library is composed of two parts:

- *TMDL Library*: A specification in the TMDL language, containing i) the definition of elementary template models and their interfaces, and ii) a set of valid composition templates, which specify valid patterns for composing the basic components.
- *Components Implementation*: The internal implementation of elementary components specified in the TMDL Library. This part depends on the specific tools and formalisms that are adopted in the modeling process. As an example, the PNML language[1] can be used for models in formalisms derived from Petri Nets.

Step 2. In the second step the different system scenarios and configurations that should be analyzed are defined. Clearly, different levels of detail can be adopted in providing a (semi-)formal description of system scenarios. At the level of detail we are using, this step corresponds in creating

[1]Petri Net Markup Language (PNML) is an exchange format for Petri Nets based on XML. It is undertaking a standardization process as ISO/IEC 15909.

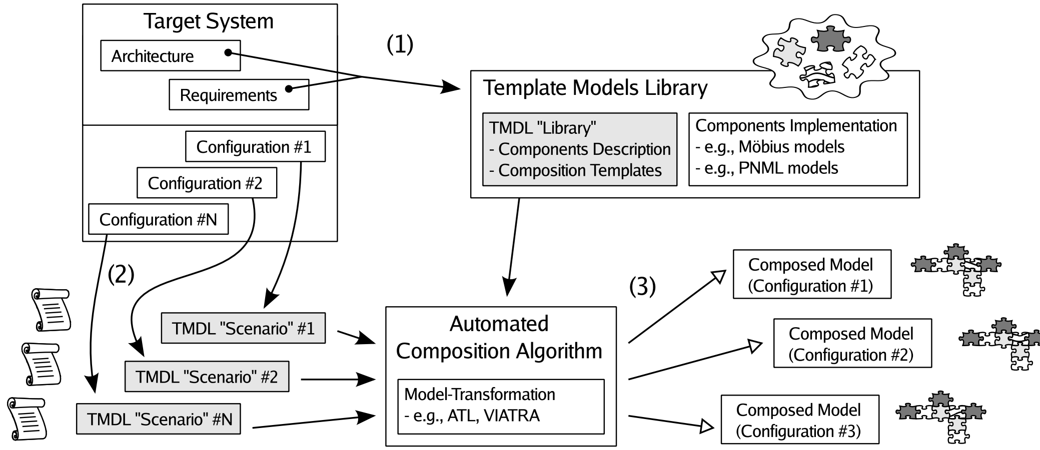

Figure 4. Our workflow for the automated generation of performability models. Elements depicted in gray are specified using the TMDL language.

composition programs in our composition system. This step is performed using the TMDL language as well, with a specification which describes *which* submodels are selected for modeling that particular scenario and the values of their parameters (*TMDL Scenario*).

From a practical perspective, TMDL "Scenario" specifications can be created manually starting from informal descriptions of scenarios to be analyzed (e.g., provided using the natural language), but could also be automatically generated from architectural UML-like models, by applying model-transformation techniques.

Step 3. Starting from the Template Models Library defined in Step 1 (i.e., our components), and the description of scenarios provided in Step 2 (i.e., our composition programs), the models for all the different scenarios and configurations are automatically assembled and then 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 properly assembling the "Template Implementations" based on the patterns specified in the "TMDL Library" (see Figure 4).

It is important to note that the "Automated Composition Algorithm" of Figure 4 is *the same for every library of template models*, i.e., it is specified and implemented *only once*, and can be reused to automatically assemble models of different systems. This it the key point of our approach, and the main reason to develop the TMDL language. From a composition systems' point of view, such algorithm is the *composition engine*, which interprets composition programs specified in TMDL, properly assembling the specified submodels.

V. Template Models Description Language

A. Main Concepts

In the following we first introduce the main concepts on which the TMDL language builds.

The basic building blocks of TMDL are **model templates**. Model templates have a set of **interfaces**, which specify how they can be connected to other model templates, and a set of **parameters**, which specify variable elements in the component (e.g., the initial number of tokens in a certain place). Component templates can be either **atomic templates** or **composition templates**. A set of model templates constitutes a **library**.

Atomic templates are associated to an **implementation** in the selected state-based formalism (e.g., a PNML file, or a tool-specific format like the XML-based format used by Möbius [14]). Composition templates group together a set of **submodels**, i.e. other model templates. For each of the referenced submodels a **multiplicity** attribute may be specified. Composition templates can have parameters as well, which allow for example parametric multiplicity values to be specified. Composition templates include a set of **merging rules**, which specify the patterns for connecting the interfaces of their submodels.

A **model class** is obtained from a component template by associating concrete values to its parameters. An **atomic class** is defined by a reference to an atomic template, and possibly a set of values for its parameters. Similarly, a **composed class** is a reference to a composition template, and possibly a set of values for its parameters. In addition to the set of values for its parameters, a **composed class** can also contain references to other model classes, which are used as concrete submodels, provided that they are compatible with the specification of the template.

A model class can be used more than once in the overall composed system model, e.g., in case multiple identical elements are present in the system. A **model instance** is an individual instance (copy) of a model class. An **atomic instance** is a copy of the template implementation, where all the parameters have been set as specified by the atomic class. A **composed instance** is an instance of a composed model class, i.e., a collection of model instances composed according the specified rules.

Submodels of a composition can in turn be other composed models. Therefore, a composed instance can be considered the root of a tree, in which internal nodes are associated with other composed instances, and leaves are atomic instances. The

overall model that represents a certain scenario is therefore identified by an instance of a composed component. Accordingly, a **scenario** (i.e., our "composition program") is specified as a set of model classes, and a "root" instance that represents the overall system model.

One of the goals of our approach is to save the modeler from specifying and connecting multiple identical component instances. However, at the same time, the language should be flexible enough to be able to represent complex scenarios and dependencies between components. In this perspective, a key role is played by the **multiplicity** and **index** concepts.

Multiplicity allows multiple model instances to be specified by specifying the model class and a numeric value. In the model generation phase, an index is automatically assigned to each of the generated instances, allowing them to be distinguished from each other. By default, indices are set based on the multiplicity of the model instance, i.e. a multiplicity of n generates n instances, with indices ranging from 1 to n. For greater flexibility, TMDL allows the user to directly specify an array of indices in place of a *multiplicity* value. For example, by specifying $\{3, 4, 5\}$ as a multiplicity value, in the model generation phase 3 identical component instances will be created, having indices 3, 4, and 5. Moreover, indices can be associated with textual prefix, thus allowing to distinguish indices related to different dimensions.

When any interface of a submodel becomes an interface of a composed model, an index (and possibly a prefix) is appended to its name. Such name is the **instance name** of the interface.

B. The TMDL Language

The metamodel of the TMDL language is shown in Figure 5. For simplicity, only the main language elements are shown in the figure: data types and other supporting elements have been omitted. As described before, the purpose of TMDL is twofold: to define libraries of template models, and to define scenarios in which such templates should be instantiated. For this reason, a specification in the TMDL language (`specification` element) may contain a model library (`library` element), and a certain number of scenarios (`scenario` elements).

1) TMDL "Library": The `library` part of TMDL supports the first step in the workflow of Figure 4. A `library` is composed of a set of `template` elements. Each template has a distinguished *name*, and a set of parameters (`param`). The `template` element is an abstract element, which is refined in atomic templates (`atomic`) and composition templates (`composition`). Each template may also have a textual *prefix*, which can be specified for indexing purposes. When the Boolean *replica* attribute is set, the composition template is simply a replica of another template; in this case the composed template can be used in any place where the template it replicates is expected.

An `atomic` template has a *body* attribute, which specifies where to actually find an implementation of such model, and a set of atomic interfaces (`interface_atomic`). An atomic interface can be either a single interface (`interface_single`), or an array of similar interfaces (`interface_array`). In the latter case, the interface has associated a *multiplicity* value and, optionally, a textual `prefix`.

A `composition` template contains a set of `block` elements, which specify which kind of subcomponents are allowed for the composition template. Each `block` element has a distinguished *name*, and references a `template` element. A *multiplicity* value can be specified to define multiple instances of the same model class as submodels.

Additionally, a `composition` template specifies a set of rules that should be followed in connecting together its subcomponents (`mergerule`). Three kinds of merge rules are supported by TMDL: `mergeall`, `mergebyname`, and `forward`. The `mergeall` rule specifies that all the selected interfaces should be connected together, to form a single interface of the composed component. The `forward` rule specifies that a single interface of a subcomponent should directly become an interface of the composed component; in this case no interfaces are joined together. The `mergebyname` specifies that, within the selected interfaces, those with the same *instance name* should be merged together, to form a single interface of the composed component. The instance name is formed by the "base name", i.e., the name specified in the composition template, and any indices and prefixes appended during model generation. Without further parameters, instance names need to be exactly the same for merging to occur. The user may however specify a set of *prefixes* to which the comparison should be restricted; in this case for merging to occur it is sufficient that the different interfaces have the same index values for the specified prefixes, while the rest of the interface name is not taken into account. If some of the selected interfaces cannot be merged with any other interface they are forward as interfaces of the composed model.

Which interfaces are selected for each `mergerule` is specified by `mergeitem` elements. Each `mergeitem` element references a single `interface` element, and optionally, a specific `block` element and a *multiplicity* value. If a `block` is specified, the rule is restricted to subcomponents derived from such block only; similarly, if the *multiplicity* attribute is specified, the rule is restricted to subcomponents having the specified indices only. Each `mergerule` contains one or more `mergeitem` elements.

Finally, a `composition` template may specify a set of bindings between its parameters and parameters of its subcomponents (`parambinding`); in such case, parameters of subcomponents are constrained to hold the same value as parameter of the parent component.

2) TMDL "Scenario": The `scenario` part of TMDL supports the second step in the workflow of Figure 4.

A `scenario` is composed of a set of classes (`class` elements). Each `class` has a distinguished *name*, and references a specific `template` in the model library. Moreover, a `class` may contain a set of `assignments`, which specify concrete values for the parameters specified in the component template.

In case of a composed class, i.e., a `class` element

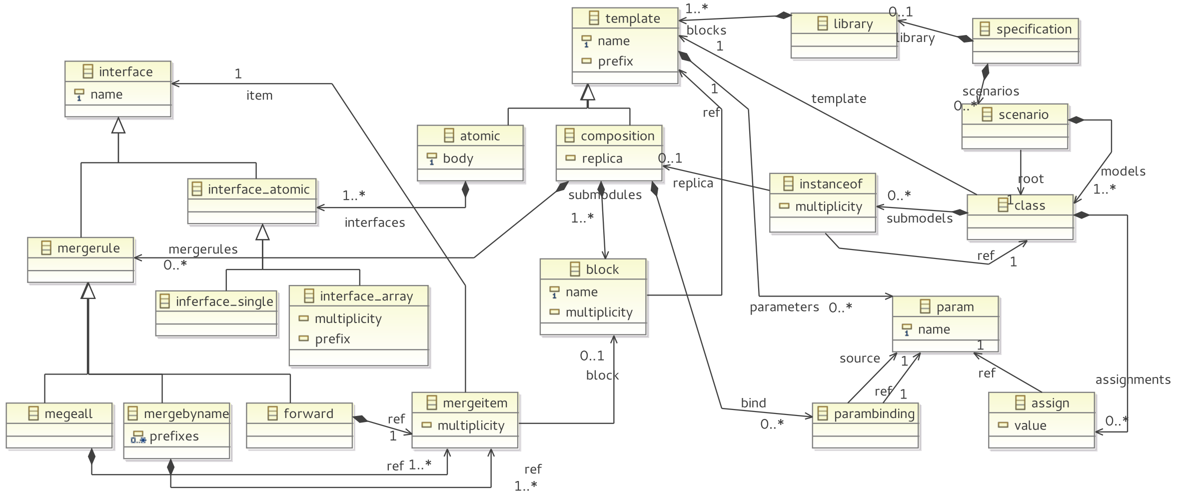

Figure 5. Simplified version of the TMDL metamodel. For simplicity, data types and other supporting elements (e.g., arrays) are not shown in the figure.

which references a `composition` template, submodels may be explicitly defined with `instanceof` elements. Each `instanceof` element references another model class in the same scenario, and possibly a *multiplicity* value. An `instanceof` element may also specify a *replica* behavior, and a "replica" `composition template`. In this case the selected model class is first replicated using the specified replica composition template (if they are compatible). It should be noted however that `instanceof` elements are not always needed. When a model template has no parameters, or only one model class derived from it exists in the scenario, submodel instances can be automatically generated by the transformation algorithm based on default parameter values specified in the library.

Finally, the *root* attribute of the `scenario` element defines the model class which, once instantiated, represents the overall system scenario (i.e., the "entry point" of the composition program).

C. Model Generation

The third step of the workflow in Figure 4 is performed by the automated model generation algorithm, which is organized in two phases: *instances generation*, in which component instances are generated, and *instances composition*, in which the generated component instances are connected together.

The generation algorithm uses two data structures: a *queue* Q containing model classes that still need to be instantiated, and a *stack* T containing model instances that have been instantiated but whose interfaces still need to be connected. More in details, given a TMDL "Library" $\mathcal{L}$, and a TMDL "Scenario" $\mathcal{S}$, the steps to assemble the overall performability model are summarized in the following:

— *Instances generation* —

1) Based on the *root* element in $\mathcal{S}$, the root model class c is identified. The pair $\{c, 1\}$ is enqueued in Q.
2) The pair $\{c, m\}$ is dequeued from Q:
   a) instances of c are created based on multiplicity m;
   b) an index is assigned to each instance;
   c) each instance is pushed into the stack T.
3) If c is a composed class:
   a) For each model class c_i, referenced as submodel of c with multiplicity m_i, the pair $\{c_i, m_i\}$ is enqueued in Q.
   b) If the composition template corresponding to c requires additional submodels that have not been specified in the scenario, the corresponding default pairs $\{c_j, m_j\}$ are created and enqueued in Q.
4) If Q is not empty then go back to Step 2. Otherwise stop.

— *Instances composition* —

1) The instance i is removed from the stack T.
2) If i is a composed instance, then the interfaces of its submodels are connected based on the rules defined by the related model template.
3) If T is not empty, then go to Step 1. Otherwise the whole model generation process ends.

VI. Application Example

A. Prototype Implementation

It is commonly agreed (e.g., see [40]) that the development of custom DSLs and the related model-driven workflows is a complex task that should be addressed with an *iterative* process. Useful feedback for the formalization of domain concepts is obtained by the definition and implementation of the language; feedback for the definition of the language is obtained by the definition and implementation of model-transformations; feedback on model-transformations is obtained by validating the produced artifacts.

According to this view, we developed a prototype implementation of the entire workflow, which will guide us through the validation and refinement of the entire approach. The implementation is based on the Eclipse Platform and its "Modeling" components [16]. The TMDL meta-model has been defined as an Ecore model; Xtext [41] has then been used to define a textual syntax, and to generate the editor, parser, and syntax highlighter. A prototype version of the transformation algorithm has been implemented using the ATL

language [3]. In the following we describe the application of the approach to the World Opera system described in Section III.

B. Application to the World Opera system

1) Library Specification: As introduced in Section III, the performability model for a World Opera performance is based on 4 atomic templates: *Component*, *StreamCollector*, *StreamAdapter*, *StreamPlayer*. For simplicity, we assume here that components are subject to two failure modes: a "silent" failure mode, in which the component just stops working, and a "noisy" failure mode, in which the component produces noisy/incorrect output. Selected portions of the TMDL "Library" specification for this system are shown in Listing 1. Ellipses (. . .) have been used to mark parts of the specifications which have been omitted.

Atomic templates are defined in lines 1–16; for each of them, an implementation of the model is referenced using the *body* attribute. The atomic template *component* (lines 1–9) has five parameters: *failrate*, which specifies the failure rate of the component, *spares*, which specifies the number of spares allowed for the component, *fprobnoisy*, which specifies the probability that the component fails with the noisy failure mode, *sw_delay* and *sw_prob*, which specify the delay and failure probability of switching to a spare component. The *component* atomic template exposes four interfaces. As discussed in Section III, *working_state* provides the current working state of the individual component, while *num_components*, *num_failed_noisy*, and *num_failed_silent* are used to record, for components in the same group, the number of them that are currently working or failed.

The *streamcollector* template (lines 10–14) models the recording of a stream. It has no parameters, and its interfaces are *stream_out*, *num_components*, *num_failed_noisy*, and *num_failed_silent*. It should be noted that the *streamcollector* template is associated with the "s" (for stream) *prefix*. The index associated to instances of this template model relates to the index of application streams available in the scenario. The *streamadapter* and *streamplayer* templates have a similar structure and they are not shown here for the sake of brevity. They are also associated with the "s" *prefix*.

Lines 18–49 depict the specification of some composition templates. The *repcomponent* template (lines 18–27) is a "replica" template for the *component* template model. This template specifies which interfaces should be connected together when composing multiple identical instances of the *component* template. This template covers the "Rep" nodes of Figure 2: "Rep1", "Rep2", "rep01", "rep02". The template has one parameter, *num*, specifing the number of components to be replicated. The interfaces which are connected together are *num_components*, *num_failed_noisy*, and *num_failed_silent*, while *working_state* interfaces are not connected.

Lines 29–42 depict the specification of the *node_displayws*, corresponding to the composition of a display workstation with its "StreamAdapter" models, and with the models of the projectors that are under its control. This composition template covers the node "DisplayWS_with_Streams" of Figure 2. The template has one parameter, *streams*, describing which streams (in the form of numerical indices) should be handled by the display workstation represented by the model.

Submodels of this template are: one instance of the *component* template to represent the workstation ("ws"), a certain number of instances of the *component* template to represent the projectors ("proj"), and a certain number of *streamadapter* templates ("sa"). As shown in the listing, the multiplicity of the *streamadapter* templates is set based on the *streams* parameter. Three mergerules are defined: i) merge `working_state` in the *component* model with `component_state` in all the *streamadapter* models; ii) merge `stream_out` of the *streamadapter* models with the `stream_in` of *node_proj* models having the same indices (i.e., referring to the same stream), and iii) forward the `stream_in` of *streamadapter* models as interfaces of the composed model (they will be either connected with corresponding interfaces of the mixer, or forwarded up as interfaces of the whole stage).

The specification of the overall WO model corresponds to the *stageset* template (lines 44–49). As submodels it has a certain number of the *stage* template model. The *stage* template model, not shown here for the sake of brevity, corresponds to the top-level Join of Figure 2, i.e., the "Gateway_with_Streams" node in the top right part of the figure. Intuitively, each *stage* submodel has one interface for each stream in which the stage is involved. More in detail, for each stream that is acquired in the stage, the `stream_out` interface of the corresponding *streamadapter* model is forwarded as `outgoing_out`; similarly, for each stream that is received from another stage, the `stream_in` interface of the corresponding *streamadapter* model is forwarded as `incoming_in`.

The *mergebyname* specification in the *stageset* template model specifies that `outgoing_out` and `incoming_in` interfaces of *stage* models should be connected based on their indices having prefix "s". For example, if stage A has an interface whose instance name is "incoming_in_s3", and stage B has an interface whose instance name is "outgoing_out_s3" the two interfaces will be connected together.

Listing 1. (Selected portions of the) TMDL "Library" specification for the World Opera system.

```
1   atomic component {
2     body "Component.xml"
3     parameters {
4       failrate def 1.0E−4, spares def 0, fprobnoisy def 0,
5       sw_delay def 1, sw_fprob def 0.05
6     }
7     interfaces { num_components, num_failed_noisy,
8       num_failed_silent, working_state }
9   },
10  atomic streamcollector prefix "s" {
11    body "StreamCollector.xml"
12    interfaces { stream_out, num_components,
13      num_failed_noisy, num_failed_silent }
14  },
15  atomic streamadapter prefix "s" { ... },
16  atomic streamplayer prefix "s" { ... },
17
18  composition replica repcomponent {
19    parameters { num def 1 }
20    submodules {
```

```
21        block c { component mult paramref { num } }
22      }
23      mergerules {
24        mergeall num_components { "component.num_components" },
25        mergeall num_failed_noisy { "component.num_failed_noisy" },
26        mergeall num_failed_silent { "component.num_failed_silent" }
27    } },
28    ...
29    composition node_displayws {
30      parameters { streams def { 1 } }
31      submodules {
32        block ws { component mult 1 },
33        block proj { node_proj },
34        block sa { streamadapter mult paramref { "node_proj.streams" } }
35      }
36      mergerules {
37        mergeall component_state {
38          "component.working_state", "streamadapter.component_state" },
39        mergebyname streams_out {
40          "streamadapter.stream_out", "node_proj.stream_in" },
41        forward streams_in { "streamadapter.stream_state_in" }
42    } },
43    ...
44    composition stageset {
45      submodules { block sg { stage } }
46      mergerules {
47        mergebyname inout prefixes "s" {
48          "stage.incoming_in", "stage.outgoing_out" }
49    } }
```

2) Specification of Scenario: Listing 2 shows a subset of the TMDL "Scenario" specification for a WO performance comprising three stages and five multimedia streams, listed in the following:

1. *a_orchestra* – audio of the orchestra;
2. *a_scene* – audio of actors;
3. *v_orchestra* – video of the orchestra;
4. *v_scene* – video of actors;
5. *v_director* – video of the orchestra director.

Streams 1, 3, and 5 are captured in stage #1, while streams 2 and 4 are captured in stage #2. All the streams are reproduced in all the three stages. Stage #1, corresponds to the model depicted in Figure 2.

Listing 2 focuses on the specification of projectors of Stage #1, i.e., the highlighted part of Figure 2. Using the *component* template, a model class for a projector is created (lines 3–8), and values are specified for all its parameters, except *sw_fprob* and *spares*, for which the default value specified in the template is used.

Listing 2 also shows the definition of two different model classes based on the same template. In particular, classes *s1_proj_orchestra_director* and *s1_proj_scene* are created from the same *node_proj* model template; the two nodes "Projector_Orchestra_and_Director" and "Projector_Scene" in Figure 2 are instances of these two classes.

The specification of *s1_proj_orchestra_director* states that such composed class has an instance of the *projector* class as submodels, and that it handles streams 3 and 5. Submodels of kind "StreamAdapter" do not need to be specified: their multiplicity is derived from the *stream* parameter, in a similar way as for the *node_displayws* template (see Listing 1). Similarly, the specification of *s1_proj_scene* specifies that the corresponding projector should handle stream 4.

A class derived from the *node_displayws* is shown in the listing, specifying that the display workstation should process streams 3, 4, and 5. Also in this case, the "StreamPlayer" models do not need to be specified.

Listing 2. (Selected portions of the) TMDL/Scenario specification for a WO performance composed of three stages and five streams.

```
1   scenario { root wo_show
2     ...
3     class projector { usetemplate component
4       assignments {
5         "component.failrate" value 0.006,
6         "component.fprobnoisy" value 0.1,
7         "component.sw_delay" value 60.0,
8     } },
9     class workstation { usetemplate component ... }
10    ...
11    class s1_proj_orchestra_director { usetemplate node_proj
12      assignments { "node_proj.streams" value { 3,5 } }
13      submodels { projector }
14    },
15    class s1_proj_scene { usetemplate node_proj
16      assignments { "node_proj.streams" value { 4 } }
17      submodels { projector }
18    },
19    ...
20    class s1_node_displayws { usetemplate node_displayws
21      assignments { "node_displayws.streams" value { 3,4,5 } }
22      submodels {
23        workstation, s1_proj_orchestra_director, s1_proj_scene
24    } },
25    ...
26    class wo_show { usetemplate stageset ... }
27  }
```

One of the major advantages in using this approach, once the TMDL "Library" specification is established, is the ability to automatically obtain the model for different system scenarios by simply changing few lines of the TMDL "Scenario" specification. Listing 3 shows the modifications needed to perform the modification discussed in Section III, i.e., introducing a new projector for stream 5, *v_director*.

Listing 3. Modified TMDL "Scenario" specification for adding a new projector dedicated to stream 5, *v_director*, to the model.

```
1     class s1_proj_orchestra { usetemplate node_proj
2       assignments { "node_proj.streams" value { 3 } }
3       submodels { projector }
4     },
5     class s1_proj_director { usetemplate node_proj
6       assignments { "node_proj.streams" value { 5 } }
7       submodels { projector }
8     },
9     class s1_proj_scene { usetemplate node_proj
10      assignments { "node_proj.streams" value { 4 } }
11      submodels { projector }
12    },
13    ...
14    class s1_node_displayws { usetemplate node_displayws
15      assignments { "node_displayws.streams" value { 3,4,5 } }
16      submodels {
17        workstation, s1_proj_orchestra, s1_proj_director, s1_proj_scene
18    } },
```

Let us now suppose that the architecture of Stage #1 changes, and that stream *v_scene* is now reproduced on two identical projectors. The few required modifications to the TMDL "Scenario" specification, with respect to Listing 2, are shown in Listing 4.

Listing 4. Modified TMDL "Scenario" specification for a scenario where stream *v_scene* is reproduced on two identical projectors.

```
1     class s1_proj_scene { usetemplate node_proj
2       assignments { "node_proj.streams" value { 4 } }
3       submodels { projector replica repcomponent mult 2 }
4     },
```

Listing 5 considers the case in which the two projectors have instead different properties. In this case, two different classes *projector_a* and *projector_b* should be defined from the same template, having different parameters. Then, the two classes are set as submodels of the *s1_proj_scene* model class, which refers to stream *v_scene*.

Listing 5. Modified TMDL "Scenario" specification for a scenario where stream *v_scene* is reproduced on two projectors having different properties.

```
1  class projector_a { usetemplate component
2    assignments { "component.failrate" value 0.006, ... }
3  },
4  class projector_b { usetemplate component
5    assignments { "component.failrate" value 0.001, ... }
6  },
7  class s1_proj_scene { usetemplate node_proj
8    assignments { "node_proj.streams" value { 4 } }
9    submodels { projector_a, projector_b }
10 },
```

Even these simple operations, if performed directly on the model, would require considerable effort for the modeler. Performing the same modification that is specified in Listing 5 would require the following steps: i) duplicating the atomic model for the "Projector", modifying the needed parameters, ii) adding an instance of the new "Projector" atomic model to the composed model of Figure 2, iii) properly connecting the interfaces of the new model to the "Projector_Scene" Join. More complex modifications would require even more steps to be performed, in the worst case leading to modify the shared variables within all the Join nodes until the root of the overall model. In the proposed approach, once the model library has been specified, such modifications are instead handled automatically by the automated model generation.

VII. Conclusions and Future Work

In this paper we have proposed an approach for the automatic assembly of large stochastic models, based on template model libraries and composition patterns. The approach is built around the TMDL language, which allows libraries of template models to be precisely defined, and then applied to specify different system configurations and generate the related analysis model. With respect to other MDE approaches for stochastic analysis, the use of TMDL allows to apply the generative approach to very different systems and different contexts, without the need to redefine ad-hoc transformation algorithms. Ongoing work is aiming at completing the implementation to generate analysis models that can be directly used as input by the Möbius tool [14], and in validating the workflow in other domains, e.g. mobile networks [8], power grid systems [11], or System-of-Systems evaluation [1].

An existing limitation of the proposed approach resides in the definition of metrics to be evaluated on the generated model, which are not addressed by the current workflow, and must therefore still be specified manually; this aspect constitutes a possible direction of future work. Another interesting research direction we are planning to investigate consists in the possibility to couple the model generation process with some specific decomposition techniques for largeness avoidance (e.g., [22]). Finally, understanding how the TMDL approach can be extended to support transition superposition (i.e., action synchronization) is also a possible direction for future work.

Acknowledgment

This work has been partially supported by the TENACE PRIN Project (n.20103P34XC) funded by the Italian Ministry of Education, University and Research, and by the European Union Seventh Framework Programme FP7/2007-2013 under grant agreement No. 610535, AMADEOS [1]. The authors would like to thank N. Raghavan, R. Vitenberg, and H. Meling for the precious discussions on the "World Opera" infrastructure and its challenges.

References

[1] AMADEOS: *Architecture for Multi-criticality Agile Dependable Evolutionary Open System-of-Systems*. Seventh Framework Programme, FP7-ICT-2013-10. 2013.

[2] U. Aßmann. *Invasive Software Composition*. Springer, 2003.

[3] *Atlas Transformation Language (ATL)*. http://www.eclipse.org/atl/ (Accessed: 10/05/2014).

[4] A. Avižienis et al. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Transactions on Dependable and Secure Computing* 1 (2004), pp. 11–33.

[5] S. Bernardi and S. Donatelli. "Stochastic Petri nets and inheritance for dependability modelling". In: *10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'04)*. Papeete, Tahiti, French Polynesia, 2004, pp. 363–372.

[6] S. Bernardi, J. Merseguer, and D. C. Petriu. "Dependability modeling and analysis of software systems specified with UML". In: *ACM Computing Surveys* 45.1 (2012).

[7] S. Bliudze and J. Sifakis. "The Algebra of Connectors – Structuring Interaction in BIP". In: *IEEE Transactions on Computers* 57.10 (2008), pp. 1315–1330.

[8] A. Bondavalli, P. Lollini, and L. Montecchi. "QoS Perceived by Users of Ubiquitous UMTS: Compositional Models and Thorough Analysis". In: *Journal of Software* 4.7 (2009).

[9] A. Bondavalli, A. Ceccarelli, and P. Lollini. "Architecting and Validating Dependable Systems: Experiences and Visions". In: *Architecting Dependable Systems VII*. Vol. 6420. LNCS. Springer, 2010, pp. 297–321.

[10] S. Chiaradonna, P. Lollini, and F. Di Giandomenico. "On a Modeling Framework for the Analysis of Interdependencies in Electric Power Systems". In: *37th IEEE/IFIP Dependable Systems and Networks (DSN'07)*. 2007, pp. 185–195.

[11] S. Chiaradonna, F. Di Giandomenico, and P. Lollini. "Definition, implementation and application of a model-based framework for analyzing interdependencies in electric power systems". In: *International Journal of Critical Infrastructure Protection* 4.1 (2011), pp. 24–40.

[12] G. Ciardo, R. German, and C. Lindemann. “A characterization of the stochastic process underlying a stochastic Petri net”. In: *5th International Workshop on Petri Nets and Performance Models*. 1993, pp. 170–179.

[13] M. Cinque, D. Cotroneo, and C. Di Martino. “Automated Generation of Performance and Dependability Models for the Assessment of Wireless Sensor Networks”. In: *IEEE Transactions on Computers* 61.6 (2012), pp. 870–884.

[14] T. Courtney et al. “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 (DSN’09)*. Estoril, Portugal, 2009, pp. 353–358.

[15] E. W. Dijkstra. “On the role of scientific thought”. In: *Selected Writings on Computing: A Personal Perspective*. Ed. by E. W. Dijkstra. Springer, 1982, pp. 60–66.

[16] *Eclipse Modeling Framework (EMF)*. http://www.eclipse.org/modeling/emf/ (Accessed: 10/05/2014).

[17] J. Fredriksson et al. *Component Based Software Engineering for Embedded Systems – A literature survey*. Tech. rep. 102. Mälardalen Real-Time Research Centre (MRTC), 2003.

[18] E. Gamma et al. *Design Patterns*. Addison-Wesley, 1995.

[19] J. Johannes. “Controlling Model-Driven Software Development through Composition Systems”. In: *Proc. of 7th Nordic Workshop on Model Driven Software Engineering (NW-MODE09)*. 2009.

[20] K. Kanoun and M. Ortalo-Borrel. “Fault-tolerant system dependability-explicit modeling of hardware and software component-interactions”. In: *IEEE Transactions on Reliability* 49.4 (2000), pp. 363–376.

[21] 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*. 2003, pp. 51–60.

[22] P. Lollini, A. Bondavalli, and F. Di Giandomenico. “A Decomposition-Based Modeling Framework for Complex Systems”. In: *IEEE Transactions on Reliability* 58.1 (2009), pp. 20–33.

[23] I. Majzik, A. Pataricza, and A. Bondavalli. “Stochastic Dependability Analysis of System Architecture Based on UML Models”. In: *Architecting Dependable Systems*. Vol. 2677. LNCS. Springer, 2003, pp. 219–244.

[24] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[25] J. Meyer. “On Evaluating the Performability of Degradable Computing Systems”. In: *IEEE Transactions on Computers* C-29.8 (1980), pp. 720 –731.

[26] 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*. Las Vegas, USA, 2011, pp. 157–166.

[27] L. Montecchi et al. *Stochastic Activity Networks model for the evaluation of the World Opera system*. Tech. rep. RCL-131001. Università degli Studi di Firenze, Resilient Computing Lab, Oct. 2013.

[28] D. M. Nicol, W. H. Sanders, and K. S. Trivedi. “Model-based evaluation: from dependability to security”. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 48–65.

[29] P. Pederson et al. *Critical Infrastructures Interdependency Modeling: A Survey of U.S. and International Research*. Tech. rep. Idaho National Laboratory (INL), 2006.

[30] B. Plateau and K. Atif. “Stochastic automata network for modeling parallel systems”. In: *IEEE Transactions on Software Engineering* 17.10 (1991), pp. 1093–1108.

[31] M. Rabah and K. Kanoun. “Performability evaluation of multipurpose multiprocessor systems: the ”separation of concerns” approach”. In: *IEEE Transactions on Computers* 52.2 (2003), pp. 223–236.

[32] A.-E. Rugina, K. Kanoun, and M. Kaâniche. “A System Dependability Modeling Framework Using AADL and GSPNs”. In: *Architecting Dependable Systems IV*. Vol. 4615. LNCS. Springer, 2007, pp. 14–38.

[33] W. Sanders and J. Meyer. “Reduced base model construction methods for stochastic activity networks”. In: *IEEE Journal on Selected Areas in Communications* 9.1 (1991), pp. 25–36.

[34] W. Sanders and J. Meyer. “Stochastic activity networks: formal definitions and concepts”. In: *Lectures on formal methods and performance analysis*. Vol. 2090. LNCS. Springer, 2002, pp. 315–343.

[35] D. C. Schmidt. “Guest Editor’s Introduction: Model-Driven Engineering”. In: *Computer* 39.2 (2006), pp. 25–31.

[36] *The World Opera*. http://www.theworldopera.org (Accessed: 10/05/2014).

[37] N. Veeraragavan et al. “Understanding the Quality of Experience in Modern Distributed Interactive Multimedia Applications in Presence of Failures: Metrics and Analysis”. In: *Proc. of the 28th ACM Symposium on Applied Computing*. Coimbra, Portugal, 2013.

[38] N. Veeraragavan, R. Vitenberg, and H. Meling. “Reliability Modeling and Analysis of Modern Distributed Interactive Multimedia Applications: A Case Study of a Distributed Opera Performance”. In: *Distributed Applications and Interoperable Systems*. Vol. 7272. LNCS. Springer, 2012, pp. 185–193.

[39] V. Vittorini et al. “The OsMoSys approach to multi-formalism modeling of systems”. In: *Software and Systems Modeling* 3.1 (2004), pp. 68–81.

[40] M. Völter. “MD* Best Practices”. In: *Journal of Object Technology* 8.6 (2009), pp. 79–102.

[41] *Xtext – Language Development Made Easy!* http://www.eclipse.org/Xtext/ (Accessed: 10/05/2014).