Model Continuity in Discrete Event Simulation: A Framework for Model-Driven Development of Simulation Models

DENİZ ÇETİNKAYA, ALEXANDER VERBRAECK, and MAMADOU D. SECK,
Delft University of Technology

17

Most of the well-known modeling and simulation (M&S) methodologies state the importance of conceptual modeling in simulation studies, and they suggest the use of conceptual models during the simulation model development process. However, only a limited number of methodologies refers to how to move from a conceptual model to an executable simulation model. Besides, existing M&S methodologies do not typically provide a formal method for model transformations between the models in different stages of the development process. Hence, in the current M&S practice, model continuity is usually not fulfilled. In this article, a model-driven development framework for M&S is presented to bridge the gap between different stages of a simulation study and to obtain model continuity. The applicability of the framework is illustrated with a prototype modeling environment and a case study in the discrete event simulation domain.

Categories and Subject Descriptors: I.6.5 [**Simulation and Modeling**]: Model Development—*Modeling methodologies*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Discrete event*

General Terms: Design, Theory, Languages

Additional Key Words and Phrases: Conceptual modeling, discrete event simulation, metamodeling, model-driven development, model transformation

ACM Reference Format:
Deniz Çetinkaya, Alexander Verbraeck, and Mamadou D. Seck. 2015. Model continuity in discrete event simulation: A framework for model-driven development of simulation models. ACM Trans. Model. Comput. Simul. 25, 3, Article 17 (April 2015), 24 pages.
DOI: http://dx.doi.org/10.1145/2699714

1. INTRODUCTION

Simulation is the process of developing a model and conducting experiments with it for a specific purpose, such as analysis, problem solving, decision support, training, entertainment, testing, research, or education [Shannon 1975; Balci 2001]. The fundamental prerequisite for simulation is a model, which is called a *simulation model*. A simulation model is developed through a modeling process, which is called *simulation model development*. The activities in a simulation study are collectively referred to as modeling and simulation (M&S) [Balci 2001].

Several methodologies have been proposed to guide modelers through various stages of M&S and to increase the probability of success in simulation studies [Shannon 1975;

Authors' addresses: D. Çetinkaya, Electrical and Electronics Engineering Department, Faculty of Engineering, University of Turkish Aeronautical Association, Ankara, Turkey; email: dcetinkaya@thk.edu.tr; A. Verbraeck, Systems Engineering Section, Multi Actor Systems Department, Faculty of Technology, Policy, and Management, Delft University of Technology, Delft, The Netherlands; email: A.Verbraeck@tudelft.nl; M. D. Seck, Engineering Management and Systems Engineering Department, Batten College of Engineering and Technology, Old Dominion University, Norfolk, VA; email: mseck@odu.edu.

DOI: http://dx.doi.org/10.1145/2699714

Roberts et al. 1983; Banks 1998; Balci 2012]. Each methodology suggests a body of methods, techniques, procedures, guidelines, patterns, and/or tools, as well as several required steps to develop and execute a simulation model. Almost all of the existing methodologies suggest developing a preliminary conceptual model at the beginning of the simulation model development process. However, only a limited number of methodologies refer to how to move from a conceptual model to an executable simulation model [van der Zee et al. 2010].

In general, conceptual modeling is a process that elicits the general knowledge about a problem or research domain and describes an abstract model that can be refined in subsequent steps to develop a solution for a given problem or a construct for a specific purpose [Olive 2007; Robinson 2008a]. There are several definitions of simulation conceptual modeling. In this research, we mainly follow Balci [2011, 2012] and Robinson [2008a, 2008b, 2011]. Balci et al. [2011] define a simulation conceptual model as a repository of high-level conceptual constructs and knowledge specified in a variety of communicative forms intended to assist in the design of an M&S application. A simulation conceptual model is a nonexecutable higher-level abstraction of the system under study [Fishwick 1995; Kotiadis and Robinson 2008]. It represents a high-level structure and the abstract behavior of a system according to the purpose of the simulation study.

A conceptual model needs to be transformed into a simulation model by using transformation rules, techniques, or patterns. However, conceptual models are often not used explicitly in the further stages of the simulation study, and a large semantic gap exists between the models at different stages of the simulation project. This gap causes a breach in model continuity in many cases. Model continuity refers to the generation of an approximate morphism relation between the different models within a development process through predefined transformation rules. Model continuity is obtained if the initial and intermediate models are effectively consumed in the later steps of a development process and the modeling relation is preserved [Hu and Zeigler 2005]. The main advantages of supporting model continuity are increasing the productivity, maintainability, consistency, and quality of a simulation study.

To address the model continuity problem, our research suggests the application of a model-driven development (MDD) approach throughout the whole set of M&S activities and proposes a formal MDD framework for M&S. MDD is a software engineering methodology that prescribes the systematic use of models as the primary means of a development process [Kleppe et al. 2003]. MDD introduces model transformations between the models at different abstraction levels and proposes the use of metamodels for specifying modeling languages. In MDD, models are transformed into other models to (semi)automatically generate the final software system. Due to the similarities between software development and simulation model development, MDD could potentially be a cost- and effort-saving approach for the M&S community while taking into account the peculiarities of simulation such as data intensiveness and time dependence.

MDD approaches were introduced to M&S in 2001 when Bakshi et al. [2001] presented a practical application of model integrated computing (MIC) into embedded system design and simulation. They provided a formal paradigm for specification of structural and behavioral aspects of embedded systems, an integrated model-based approach, and a unified software environment for embedded system design and simulation. One year later, Vangheluwe et al. [2002] introduced metamodeling and model transformation concepts for M&S. They present an approach to integrate three orthogonal directions of M&S research: multiformalism modeling, model abstraction, and metamodeling.

In the context of MDD, metamodeling is the process of complete and precise specification of a modeling language in the form of a metamodel. For many years, the term *metamodeling* was used in the simulation field in a different context. Here, metamodeling

referred to constructing computationally faster models with the same input–output behavior as complex simulation models [Barton 1998; Kleijnen 2009]. Such a metamodel is now called a *surrogate model*. Surrogate models mimic the complex behavior of an underlying simulation model. Metamodeling in the context of MDD was introduced to simulation later.

Tolk and Muguira [2004] introduced the model-driven architecture (MDA) [OMG 2003] for distributed simulation. They presented methods to merge the high-level architecture (HLA) and the Discrete Event System Specification (DEVS) within the MDA framework. Since the introduction of these initial ideas, there have been many efforts to use MDD concepts in M&S. In some cases, metamodeling was used to describe domain-specific modeling languages for one stage in the M&S methodology [Duarte and de Lara 2009; Levytskyy et al. 2003; Lei et al. 2009; Touraille et al. 2011; Sarjoughian and Markid 2012]. In all of these applications, MDD tools were mainly used for automatic code generation from a system specification, and they ignored the more abstract conceptual models.

More unified solutions are presented by following either the MDA approach [DeAntoni and Babau 2005; Guiffard et al. 2006; D'Ambrogio et al. 2010; Garro et al. 2013; Sarjoughian and Markid 2012] or the MIC approach [Topçu et al. 2008; Çetinkaya et al. 2010; Ledeczi et al. 2003]. Guiffard et al. [2006] provide a study that aims at applying a model-driven approach to M&S in the military domain. Their prototype demonstrates that the automated transformation from a source model to executable source code is possible. D'Ambrogio et al. [2010] introduce a model-driven approach for the development of DEVS simulation models. Their approach allows one to specify DEVS models in UML, and it automates the generation of DEVS simulations that make use of the DEVS service-oriented architecture implementation that is presented by Mittal et al. [2009]. Garro et al. [2013] propose an MDA-based process for agent-based M&S (MDA4ABMS) that uses the agent modeling framework of the Eclipse Agent Modeling Platform project. The MDA4ABMS process allows the automatic production of platform-specific simulation models starting from platform-independent simulation models. The source code can be generated automatically, which results in a significant reduction of programming and implementation efforts. Topçu et al. [2008] propose the Federation Architecture Metamodel (FAMM) for describing the architecture of a HLA-compliant federation. FAMM supports the behavioral description of federates based on sequence charts and formalizes the standard HLA object model and federate interface specification.

In addition to the MDA and MIC approaches, UML or SysML (Systems Modeling Language)-based methods for metamodel-based simulation have been proposed by using either extension mechanisms or model transformations [Pop et al. 2007; Schamai et al. 2009; Kerzhner et al. 2011; Batarseh and McGinnis 2012; Reichwein et al. 2012]. Pop et al. [2007] propose a UML profile for Modelica called *ModelicaML* (Modelica Modeling Language), which is a graphical modeling language that enables users to depict a Modelica simulation model graphically. The Modelica language provides a sound way for defining models in a declarative, modular, and hierarchical way [Fritzson 2010]. ModelicaML extends the graphical modeling capabilities of Modelica by using UML diagrams. ModelicaML enables the generation of executable Modelica code. Schamai et al. [2009] extend ModelicaML with constructs for model behavior. They also discuss the usage of executable UML state machines for system modeling and present a proof of concept for the translation of UML state machines into executable Modelica code [Schamai et al. 2010, 2013].

Paredis and Johnson [2008] present a generic approach for defining automated, bidirectional transformations between SysML and domain-specific languages. They propose metamodels for a specific domain and an SysML profile in which the domain

specific constructs are mapped to stereotypes. Then, graph transformations are created that use the domain-specific metamodel and the corresponding SysML profile. This approach has been formalized in Paredis et al. [2010] for the mapping of SysML to Modelica. The SysML-Modelica transformation is specified between the profile constructs and the Modelica language constructs as captured in the Modelica metamodel. The transformation approach simplifies the transformation to Modelica and facilitates model reuse by using existing model libraries.

In addition to these approaches, Risco-Martín et al. [2009] present a UML-based DEVS simulation method that is part of the DEVS Unified Process (DUNIP). DUNIP proposes a process that uses the DEVS formalism as a basis for automated generation of models from various requirement specifications [Mittal et al. 2007, 2008]. Mittal and Douglass [2012] show that an underlying DEVS metamodel provides model reusability across different platforms. Furthermore, there have been studies that use ontologies during the development of simulation models to provide a substantive basis for defining a system structure for a domain [Silver et al. 2007; McGinnis et al. 2011; Tolk and Miller 2011].

Although the aforementioned efforts show the applicability of the MDD approach in the simulation field, most of the applications are formalism or specification dependent. As a result, to the best of our knowledge, there is no generic theoretical framework that provides guidance for formal model transformations while moving from a conceptual model to an executable simulation model. One of the reasons for this issue could be that there is not a commonly accepted terminology and formal specification for MDD [Favre 2004]. Although MDD has been advocated as a cost- and effort-saving development approach for software projects [Selic 2003], the MDD principles for the M&S field are currently only described informally. The M&S research community has recently started to move toward a theoretical framework for simulation, such as in the work of Tolk et al. [2013] and Diallo et al. [2011], which use model theory.

The contribution of this work is to bridge the gap between different stages of a simulation study in a formal manner as well as proposing a theory for MDD within M&S. A MDD framework for M&S, called *MDD4MS*, has already been introduced in Çetinkaya et al. [2011]. The proposed framework defines three metamodels for different stages of a simulation study and suggests model transformations based on these metamodels. The improved version of the framework is represented in Çetinkaya et al. [2013a]. The MDD4MS prototype implementation is illustrated in Çetinkaya et al. [2012, 2013b]. Although our earlier work introduced the initial ideas, basic concepts, and practical aspects, the formal theory of our framework is presented in this article. In this theory, MDD principles have been clearly defined and specified for the M&S field. A comprehensive definition of the MDD4MS framework is presented and formal relations between the key concepts are defined.

Section 2 provides a comparative analysis between the selected M&S methodologies to highlight the common steps in simulation studies. Section 3 proposes the theory of modeling, metamodeling, model transformation, and MDD. Section 4 introduces a mapping between the MDD concepts and the M&S concepts. Section 5 proposes the theory of the MDD4MS framework. Section 6 presents a case study with the MDD4MS prototype to illustrate the applicability of the framework in the discrete event simulation domain. Finally, conclusions are drawn in Section 7.

2. SIMULATION MODEL DEVELOPMENT AS A MODEL-BASED PROCESS

In this section, a set of M&S methodologies will be analyzed to see if they can be identified as a model-based approach and if an MDD approach can be incorporated into these methodologies. Each methodology prescribes several steps for conducting a simulation study [Balci 2012]. Looking at the existing methodologies, some similar

Table I. Analysis of M&S Methodologies According to Their Support for Model Continuity

Analysis Criteria	Shannon [1975], p. 24	Roberts et al. [1983], p. 8	Nance [1984], p. 76	Fishwick [1995], p. 4	Shannon [1998], p. 9	Banks [1998], p. 15	Law [2003], p. 67	Robinson [2004], p. 52	Wainer [2009], p. 27	van Daalen et al. [2009], p. 1129	Sargent [2010], p. 169	Balci [2012], p. 4
Has multiple models?	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
How many models?	2	3	4	6	2	2	2	2	3	3	2	2
Has a conceptualization step?	P[1]	P[1]	Y	Y	P[1]	Y	Y	Y	Y	Y	Y	Y
Has a formal specification step?	N	Y	N	Y	N	N	N	P[3]	Y	Y	P[3]	P[3]
Has a programming step?	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Proposes conceptual model transformation?	N	P	Y	Y	P	Y	N	Y	N	Y	N	Y
Proposes automatic code generation?	N	N	Y	Y	N	Y	N	N	N	N	N	Y
Provides formal model transformations?	N	N	N	P[2]	N	N	N	N	N	N	N	N

Y, yes; N, no; P, partially.
[1]Conceptual model is not mentioned, but a diagram such as a flow diagram is suggested.
[2]The framework does not provide formal transformation methods but proposes some heuristics, explicit translation rules, and general code-writing rules.
[3]A software-specific description that determines how to structure the model is proposed.

terms and patterns can be distinguished. Each simulation study roughly consists of four main stages: problem definition, model development, simulation model execution, and analysis of results. In the problem definition stage, the purpose of the simulation study, the boundaries of the problem, and the requirements are defined. In the simulation model development stage, an executable simulation model is developed, which is ready to be executed on a platform, and data for the simulation study is collected. Developing a conceptual model is often recommended at the beginning of this stage. In the simulation model execution stage, experiments are designed and the simulation model is executed using a subset of the collected data. In the analysis stage, the experimentation results are presented and analyzed.

Table I presents an analysis of different M&S methodologies. Each methodology is evaluated according to a set of questions to analyze their support for model continuity [Atkinson and Kühne 2003]. The results in Table I show that all of the methodologies introduce multiple steps for carrying out a simulation study. With regard to the outputs of these steps, it is identified that the development process highly relies on models such as conceptual models, formal models, computer models, and experimentation models.

All of the methodologies require a conceptualization step, and many of them suggest transforming the conceptual model into a simulation model but do not formally define how to do that. All of the methodologies prepare for computer simulation and require a programming step or model coding step. Some of them mention automatic code generation. Therefore, M&S can be identified as a model-based process [Ören 2007], and an MDD method can be incorporated into these methodologies. On the other hand, some of the methodologies are lacking a formal specification step that is important to separate the formal system specification from programming concerns. Moreover,

almost all of the methodologies ignore model transformations, and none of them provides formal model transformation methods between the models in different steps. Hence, model continuity is not supported. Well-defined model-driven approaches are needed to provide model continuity. The MDD4MS framework provides a generic MDD framework for M&S that supports model continuity [Çetinkaya et al. 2011]. It provides a formalism-independent solution via formalizing the steps that can be incorporated into existing methodologies. Before presenting the theory of the MDD4MS framework, the next section gives a formal definition of models, modeling languages, metamodels, and model transformations, and it proposes a theory for MDD within M&S.

3. THEORY OF MODELING, METAMODELING, AND MDD

This section provides a theoretical basis for modeling, metamodeling, and model transformations. Although a sound and comprehensive reference is lacking in software engineering literature, the related work on mathematical explanation of MDD concepts [Favre 2004; Kühne 2006; Jouault and Bézivin 2006; Jackson and Sztipanovits 2009] provided us with the preliminary definitions and ideas.

3.1. What Is a Model?

Modeling is the process of representing a source system for a specific purpose in a form that is ultimately useful for an interpreter. The concrete form that represents the system is called the *model*. Broadly speaking, a model is a representation of something. The represented thing is called the *source*, which can, for instance, be an object, an idea, a phenomenon, an organization, a process, or an event. A model can also be a representation of another model. In systems thinking, a model is a representation of a source system. We see a system as a set of interrelated elements [Klir 1969; von Bertalanffy 1968]. A model is developed for a purpose related to the source system [Klir 1969; Shannon 1975; Ackoff 1978]. This purpose can be achieved by executing or interpreting the model and thereby gaining knowledge about the source system. The interpretation can only be validated in a given context. The context includes the purpose of the modeling process; information about the environment of the source system; and the constraints, assumptions, and facts that affect the modeling process. A context can formally be defined in a model as well [Theodorakis et al. 2002].

A model is specified in a modeling language [Milicev 2009]. A modeling language consists of an abstract syntax, concrete syntax, and semantics [Atkinson and Kühne 2003]. The abstract syntax describes the vocabulary of the concepts provided by the modeling language and how they can be connected to create models. The abstract syntax consists of the concepts, the relationships, and well-formedness rules, where well-formedness rules state how the concepts may be combined. The concrete syntax provides a way to show the modeling elements in a concrete form that we see and work with on paper or on the computer screen [Rumpe 1998]. The expression generated with the concrete syntax is called the *model*, and it is often in a structured textual form. The semantics of a modeling language is the additional information to explain what the abstract syntax actually means. We use the grammar notion of formal language theory to define the abstract syntax and assume that a default concrete syntax is provided by the metalanguage (see Appendix A). In this article, we use the following primitive terms and relations:

—S is the infinite set of all source systems. Variables such as "$s, s_1, s_2, \ldots$" range over S.
—C is the infinite set of all contexts. Variables such as "$c, c_1, c_2, \ldots$" range over C.
—L is the infinite set of all formal languages. Variables such as "$l, l_1, l_2, \ldots$" range over L.

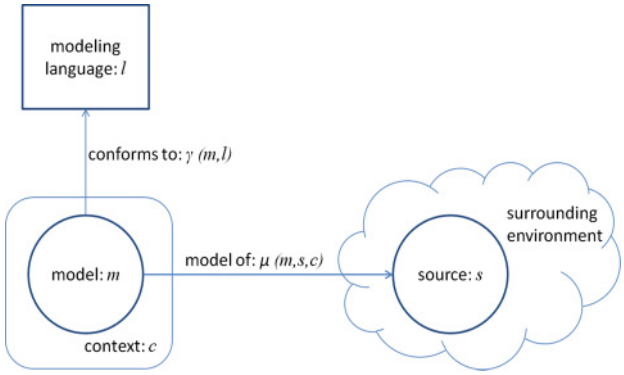

Fig. 1. The relationships between model, source, and language.

The usual set operations are applicable to S, C, and L, such as union, intersection, difference, complement, subset, proper subset, Cartesian product, and powerset. The $+$ function $c_1 + c_2 : C \times C \rightarrow C$ is used to represent the composition of two contexts, where $c_1 + c_1 = c_1$, $c_1 + c_2 = c_2 + c_1$ and $c_1 \leq c_1 + c_2$. We next propose the following definitions using formal language theory.

Definition 3.1 (*Model*). Let $g = \{T, N, I, P\}$ be a grammar where $T, N, I,$ and P are defined in Appendix A, and the language that g generates is $l(g)$. If an expression $m \in l(g)$ is a representation of a source system s within a context c, then m is said to be a model of s in c. The infinite set of all models is denoted by M. Variables such as "$m, m_1, m_2, \ldots$" range over M. The ternary relation "model-of" is denoted by μ: $M \times S \times C$.

Definition 3.2 (*Conforms-to Relation*). If an expression $m \in l(g)$ and $m \in M$, then the language l is a modeling language and m conforms-to l. The binary relation "conforms-to" is denoted by γ: $M \times L$.

Axiom 3.3 (*Transitive-μ*). $\mu(x, y, c_1) \wedge \mu(y, z, c_2) \wedge y \in M \Rightarrow \mu(x, z, c_1 + c_2)$.

Figure 1 shows the relationships between a model, a source, and a language. Although the terminal symbols of the grammar provide a default concrete syntax (primary view), it is possible to define other concrete syntax mappings (secondary views). For example, it is very common that a model has a textual view and a diagrammatic view in computer science. We suggest using an extended grammar to define a mapping from the default concrete syntax to a set of concrete symbols. In this way, all of the possible final or intermediate productions of a given grammar with both terminal and nonterminal symbols can be included in the mapping. The language defined with the extended grammar allows defining composed modeling elements for a modeling language. Most important is that the primary view has all of the specified information and the secondary views will be syntactically either equivalent to or weaker than the primary view.

Definition 3.4 (*Extended Grammar*). Let $g = \{T, N, I, P\}$ be a grammar and $l(g)$ be a modeling language. Let $\widehat{g} = \{T \cup N, N, I, P\}$ be another grammar and $\widehat{l}(\widehat{g})$ be all expressions generated by $\widehat{g}$. $\widehat{l}$ includes all of the possible expressions with both terminal and nonterminal symbols of g. Hence, $l \subseteq \widehat{l}$. The grammar $\widehat{g}$ is the extended grammar of g.

Definition 3.5 (*Concrete Syntax Mapping*). Let $g = \{T, N, I, P\}$ be a grammar and $l(g)$ be a modeling language. A concrete syntax mapping for $l(g)$ is a binary relation

from $\widehat{l}(\widehat{g})$ to cs, where cs is a set of concrete symbols, and the relation is denoted as $\psi(\widehat{l}(\widehat{g}), cs)$.

Definition 3.6 (*Secondary View*). Let $m \in M$ be a model that conforms-to a modeling language $l(g)$. For a given set of concrete symbols cs, if m can be mapped to an expression $v \in cs^*$, by using a concrete syntax mapping $\psi(\widehat{l}(\widehat{g}), cs)$, then v is a secondary view of the model m. cs^* is the infinite set of all expressions, which can be obtained by composing zero or more symbols from cs. The infinite set of all secondary views is denoted by V. Variables such as "$v, v_1, v_2, \ldots$" range over V.

To specify, view, or change models on computer platforms, we use model editors. A model editor for a modeling language l provides a way to specify models according to the syntax of l and save them according to a specific file format. In addition, the editor uses a language parser to decompose a model according to the abstract syntax of l [Muller et al. 2008] and shows the model on the screen using one or more views from V. If a concrete syntax mapping is defined, the editor can show a different view of the model, such as by changing the shape of a specific type of element in a model. Although secondary views of a model are based on the default concrete syntax, semantically they can be more powerful than the primary view. In many cases, the model editor provides extra features, such as verification or syntax highlighting. Most model editors are also capable of hiding aspects of a model for the user. In this case, the concrete syntax remains unchanged. It is not easy to define a grammar and a concrete syntax for a modeling language from scratch. Furthermore, developing a model editor for a language requires advanced software engineering skills. The metamodeling approach, introduced in the next section, solves most of these problems.

3.2. Metamodeling

Metamodeling is the process of specifying a grammar of a modeling language in the form of a model, which in turn can be used to specify models in that language. Hence, metamodeling is a modeling process where the source is a grammar. The following definitions can be derived from the earlier ones.

Definition 3.7 (*Metamodel*). Let $mm \in M$ be a model and $\exists\mu(mm, s, c)$, where $s \in S$ and $c \in C$. A model mm is a metamodel if and only if s is a grammar. The infinite set of all metamodels is denoted by M', where $M' \subset M$. Variables such as "$mm, mm_1, mm_2, \ldots$" range over M'.

Definition 3.8 (*Metamodeling Language*). For any $mm \in M'$ and $\gamma(mm, l)$, the language l is a metamodeling language. The infinite set of all metamodeling languages is denoted by L', where $L' \subset L$. Variables such as "$l', l'_1, l'_2, \ldots$" range over L'.

Similar to a modeling language, a metamodeling language has one abstract syntax and at least one concrete syntax. To specify, view, or change metamodels on computer platforms, we use metamodel editors. All information about models can also be applied to metamodels and metamodeling languages.

The purpose of the metamodeling process is representing the grammar of a modeling language to provide a proper way to develop models with that language. For example, instead of developing a model for a specific problem in a certain domain, first a metamodel that defines the concepts that apply to a larger set of problems in that domain is specified. Then, the metamodel is used to develop a specific model. In this case, a model is said to be an "instance-of" the metamodel. If a model m is an instance of a metamodel mm, where mm is a model of a grammar g, then the model m *conforms-to* $l(g)$. Figure 2 illustrates the relations between a model and a metamodel. If mm is a model of a grammar g, then we write $l(mm) \cong l(g)$.

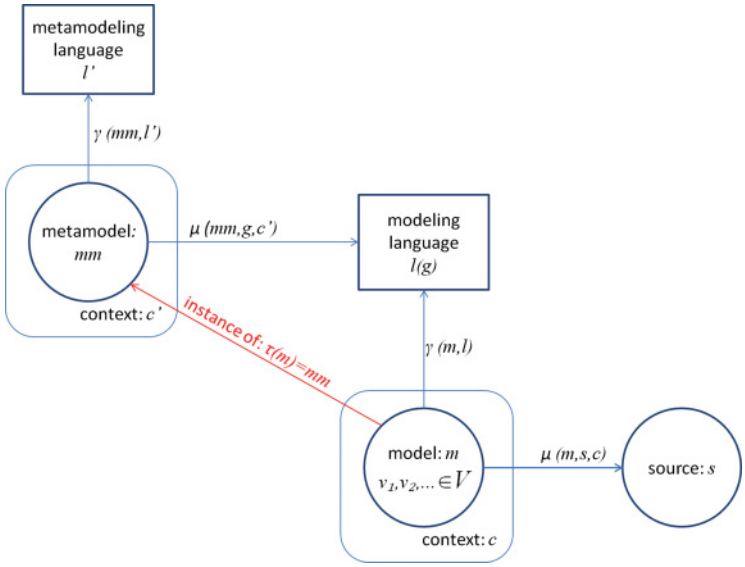

Fig. 2. Metamodeling.

Definition 3.9 (*Instance-of Relation*). Let $mm \in M'$ be a metamodel, $m \in M$ be a model, and $\gamma(m, l(mm))$. Then, m is an instance-of mm if and only if every element of m is an instance of some element in mm. The binary function 'is-instance-of' is denoted by $\tau(m) : M \rightarrow M'$.

LEMMA 3.10 (FORMALIZED CONFORMS-TO). $(\tau(x) = y) \Rightarrow \gamma(x, l(y))$ *(by Definition 3.9).*

Popular metamodeling languages are MOF [OMG 2006], Ecore [Eclipse 2014], KM3 [IRISA 2011], and MetaGME [Emerson et al. 2004]. One can easily notice that the metamodeling pattern can be applied again, and a metamodeling language can also be defined with a metamodel. In this case, the metamodel that is a model of the grammar of a metamodeling language is called a *meta-metamodel* and is specified in a meta-metamodeling language.

Although, it is possible to increase the number of metamodeling levels theoretically, a four-level metamodeling architecture that was introduced in the UML specification by OMG in 1999 [OMG 1999] is generally used in practice. In the metamodeling architecture introduced by OMG, the M3 level is for representing a metamodeling language as a meta-metamodel, the M2 level is for representing a modeling language as a metamodel, the M1 level is for representing a system without specific user data, and the M0 level is for representing a system with user data. In addition, OMG introduces the concept of a UML profile at the M1 level. It is a way of specifying an incomplete (parameterizable) model so that the details can be filled in later. We will use the term *template models* for these incomplete models within the modeling process. A set of template models in a modeling language for a domain is called a *domain-specific modeling library*, whereas the set of modeling elements provided by the grammar of the language is called the *basic* (or *core*) *modeling library* [Atkinson and Kühne 2002]. We accept these features as implementation-specific concepts, and many state-of-the-art metamodel editors provide them.

In general, the metamodel editors provide extra features, such as model-to-model (M2M) transformation, code generation, or model interpretation. In this case, the

complete tool set is called a *metamodeling environment*. A full-featured metamodeling environment provides a way to specify a metamodel *mm* and automatically generate a model editor for the modeling language $l(mm)$. The resulting editor may either work within the metamodeling environment or less commonly be produced as a separate stand-alone program. Once we have metamodels and models, we can discuss metamodel-based model transformations. The next section explains model transformations in the MDD context.

3.3. Model Transformations

A model transformation is the process of converting a model into another form according to a set of transformation rules. According to the output type or target language of the process, the model transformation can be classified as an M2M transformation, a model to text (M2T) transformation, or a model to code transformation (M2C or code generation). As the name implies, an M2M transformation converts a model into another model, whereas an M2T transformation converts a model into text. M2T transformation is generally used for supportive document creation. If the model is used to generate source code, then the transformation is called *code generation*. Well-known M2M transformation languages are ATL (ATLAS Transformation Language) and QVT (Query/View/Transformation) [Çetinkaya and Verbraeck 2011].

In the MDD context, we are only interested in formal model transformations. A formal transformation requires that the models are specified in well-defined modeling languages and the rules are defined with a model transformation language. A transformation rule consists of two parts: a left-hand side that accesses the given model and a right-hand side that generates the target system. Hence, a model transformation is performed with a well-defined model transformation pattern. To provide model continuity, the target model should contain as much as possible from the source model and the initial modeling relation should be preserved [Ehrig and Ermel 2008].

Definition 3.11 (*Model Transformation Pattern*). A model transformation pattern p is defined as a triple $p = \{l_x, l_y, r\}$, where

l_x is the source modeling language,

l_y is the target language (any language such as a programming language, a modeling language, or any well-defined language), and

r is a finite set of transformation rules from l_x to l_y, which are defined with a model transformation language.

The infinite set of all model transformation patterns is denoted by P. Variables such as "$p, p_1, p_2, \ldots$" range over P.

Definition 3.12 (*Formal Model Transformation*). Let $m \in M$ be a model that conforms to l_1, and $p = \{l_1, l_2, r\}$ be a model transformation pattern. If an expression $e \in l_2$ is derived from m by applying p, then the derivation is called a *formal model transformation*. We say that m is transformed-to e by applying p and denote it as $\theta(m, p) = e$.

Definition 3.13 (*Formal Model-to-Model Transformation*). If $\theta(x, p) = y$ and $y \in M$, then the transformation is called a *formal model-to-model transformation*.

THEOREM 3.14. *If the model transformation pattern p preserves the system related information in the source model, then*

$$(\theta(x, p) = y) \wedge \mu(x, s, c) \wedge (y \in M) \Rightarrow \mu(y, s, c + p).$$

Figure 3 shows the relationships in an M2M transformation.

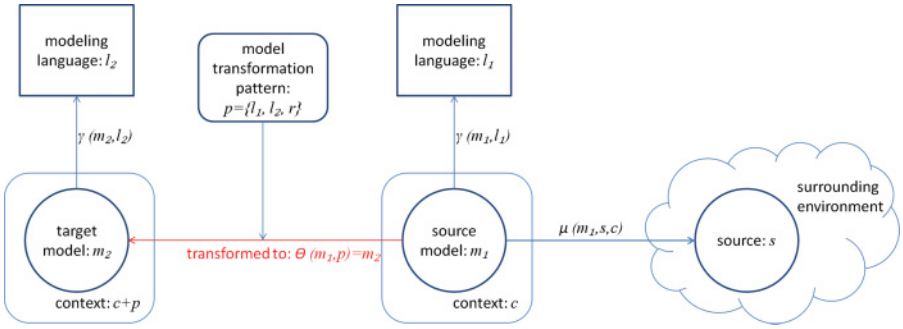

Fig. 3. The relationships in an M2M transformation.

Definition 3.15 (*Code Generation*). If $\theta(x, p) = y$ and y is a source code in a software programming language, then the transformation is called *code generation*.

LEMMA 3.16. $(\theta(x, p) = y) \wedge (p = \{l_1, l_2, r\}) \Rightarrow \gamma(x, l_1)$ *(by Definition 3.12).*

LEMMA 3.17. $(\theta(x, p) = y) \wedge (y \in M) \wedge (p = \{l_1, l_2, r\}) \Rightarrow \gamma(y, l_2)$ *(by Definitions 3.12 and 3.13).*

For a metamodel-based model transformation, either the source model is an instance of a metamodel or the target model is an instance of a metamodel, or preferably both.

LEMMA 3.18. $(\theta(x, p) = y) \wedge (p = \{l(mm_1), l_2, r\}) \wedge (mm_1 \in M') \Rightarrow (\tau(x) = mm_1)$ *(by Definition 3.12).*

LEMMA 3.19. $(\theta(x, p) = y) \wedge (y \in M) \wedge (p = \{l_1, l(mm_2), r\}) \wedge (mm_2 \in M') \Rightarrow (\tau(y) = mm_2)$ *(by Definitions 3.12 and 3.13).*

The goal of model transformations is to automatically generate different representations of a system at different abstraction levels and to enable the reuse of information that was once modeled. A key point here is the model transformation language. A model transformation language is a language that provides a way to write transformation rules for the expressions of a formal grammar. Given two formal grammars (one is for the source language and one is for the target language) and a model specified in the source language, a language parser can parse the transformation rules, an interpreter can interpret the source model, and a model transformation engine can apply a consecutive set of the rules on the source model and generate a target model according to the interpretation. A model transformation tool combines all of these concepts and usually is embedded in a metamodeling environment. Sometimes a model transformation is called a *graph transformation* if the models are specified as graphs, and the transformation tool is then called a *graph transformation tool* [Agrawal et al. 2004]. A model transformation is also known as a model morphism [Agostinho et al. 2010].

The presented definitions and relations provide a generic framework allowing anything to be the source of a modeling process. In other words, anything can be modeled. For example, it is possible to develop a model of a model transformation pattern [Bézivin et al. 2006]. In addition, a metamodel can be defined to represent the grammar of the model transformation language.

In the M&S domain, metamodeling has been applied for formal transformations. For example, Paredis and Johnson [2008] discuss the use of a graph transformation approach to define a mapping between domain-specific metamodels and a corresponding SysML profile. Paredis et al. [2010] provide an overview of the formal transformation between SysML and Modelica. Risco-Martín et al. [2009] propose XML/XSLT-based

transformations to generate executable DEVS models from UML models by using a DEVS metamodel. McGinnis et al. [2011] discuss how ontologies can be effectively deployed in simulation and present formal transformations to generate Arena and AnyLogic models from SysML models. All of these studies show that the transfer of modeling information between different models is improved by formalizing the steps.

3.4. What Is an MDD Process?

MDD is a software engineering methodology that uses well-defined modeling languages to specify models and applies model transformations to automatically generate source code from an initial software design model. In an MDD approach, there is usually a chain of several M2M transformations and a final M2C transformation. By using the definitions given in the previous sections, we define an MDD process as follows.

Definition 3.20 (*MDD Process*). A *model-driven development* process for a software application development is defined as a tuple

$$mdd = \{n, MML, ML, MO, SL, pl, MTP, STP, MT, sc, TO\}$$

where

n is the number of models;
$MML = \{l'_0, l'_1, \ldots, l'_{n-1}\}$ is an ordered set of metamodeling languages (can be defined with meta-metamodels);
$ML = \{l_0(mm_0), l_1(mm_1), \ldots, l_{n-1}(mm_{n-1}) \mid \gamma(mm_i, l'_i)(0 \leq i < n)\}$ is an ordered set of modeling languages (preferably defined with metamodels);
$MO = \{m_0, m_1, \ldots, m_{n-1} \mid \gamma(m_i, l_i)(0 \leq i < n)\}$ is an ordered set of models, m_0 is the initial model, and m_{n-1} is the final model;
SL is a set of supplementary languages (including at least a model transformation language for writing transformation rules);
pl is a programming language for final code generation;
$MTP = \{p_0, p_1, \ldots, p_{n-2}, p_{n-1}\}$ is a set of formal model transformation patterns, where p_i is an M2M transformation pattern, p_{n-1} is a code generation pattern, and $(p_{n-1} = \{l_{n-1}(mm_{n-1}), pl, r\}) \in MTP$ (including at least the final code generation pattern);
STP is a set of other supplementary model transformation patterns;
$MT = \{(\theta(x, p) = y) \mid (x \in M) \wedge (p \in MTP)\}$ is a set of formal model transformations, where $(\theta(m_{n-1}, p_{n-1}) = sc) \in MT$ (including at least the final code generation),
sc is the final source code; and
TO is a set of tools to ease the activities.

The most notable advantages of MDD are rapid software development and increased productivity. Hence, computerized tool support is very important in MDD approaches. We recommend the following basic tools for MDD: a full-featured metamodeling environment, a set of model editors, and a set of model transformation tools [Atkinson and Kühne 2003; Emerson et al. 2004].

Although MDD principles are described by different specifications, the most commonly used terminology is introduced by MDA [OMG 2003]. MDA introduces three types of viewpoints. The computation independent viewpoint focuses on the environment of the system and the requirements for the system. The platform-independent viewpoint focuses on the structure and operation of the system while hiding the details necessary for a particular platform. The platform-specific viewpoint focuses on the use of a specific platform by a system. Based on these viewpoints, three types of models are used in MDA. A computation-independent model (CIM) is a representation of a system from the computation-independent viewpoint that does not show the details of the system. A platform-independent model (PIM) is a representation of a system that exhibits a specified degree of platform independence to be usable with several different

platforms. A platform-specific model (PSM) is a representation of a system that combines the specifications in the PIM with the details that specify how that system uses a particular type of platform. MDA also defines the PIM-to-PSM transformation and requires that the PSM will include the source code. However, a CIM-to-PIM transformation is not clearly defined in MDA, as a CIM is assumed to be a kind of requirements specification.

When following the MDA viewpoints, we can say that at least three types of models are produced during a software development lifecycle: a CIM for the analysis stage, a PIM for the design stage, and a PSM for the implementation stage [Sommerville 2007]. The final source code will be generated from the PSM. In more generic terms, an MDD process is supposed to have an initial model, a number of intermediate models, a final model, and final source code. The aim is to obtain a large part of the models and the final code through successive model transformations. An MDD process supports model continuity by formal model transformations.

Definition 3.21 (*Model Continuity*). Let *mdd* be an MDD process, m_0 be the initial model of this process, and $\mu(m_0, s, c)$.

We say that model continuity is obtained in an *mdd* process if and only if $n \geq 2$ and $\mu(m_{final}, s, c + x)$, where m_{final} is the final model of the MDD process, it is generated through formal model transformations, and it preserves the modeling relation.

As a result, MDD approaches place models in the core of the entire software development process. They suggest better and faster ways of developing software systems through automated model transformations that are specified with well-defined modeling languages. As a final remark, because different MDD tools apply the same principles, an MDD expert can easily combine different approaches and tools to carry out an MDD process [Bézivin 2005]. For example, MDA concepts such as PIM and PSM metamodels can be defined with the MIC tool suite [ISIS 1997] and GME [Emerson et al. 2004]. The next section extends the MDD concepts for M&S, and Section 5 presents the resulting MDD4MS framework.

4. APPLYING MDD IN M&S

A simulation model is a representation of a system that can be simulated by means of experimentation [Kleijnen 2008]. It may be a physical model, a mathematical model, a computer model, or a combination of these [Roberts et al. 1983]. A simulation model is executed on a simulation platform to generate simulation results. Following the earlier definitions, we say that modeling for computer simulation is the process of representing a system for a specific purpose in a form that is executable by a simulator. Figure 4 shows the basic concepts and relations based on the general modeling principles given in Section 3.1.

The main difference from a general modeling process is that the model is interpreted (i.e., executed) by a model interpreter (i.e., simulator). In the context of computer simulation, a simulator is a computer program that can be executed on a computer or embedded in hardware. The output of the simulator is called the *simulation results*. A computer simulation model needs to be specified in a programming language that provides or can be extended to provide simulation capabilities (preferably with a suitable model editor). Simulation is the process of conducting experiments with a model so that the behavior of the system is simulated over time. We propose the following definitions based on the MDD concepts from the previous section.

Definition 4.1 (*Simulation Model*). Let $m \in M$ be a model, $s \in S$ be a source system, and $\mu(m, s, c)$. m is a simulation model if and only if there exists a simulator that can simulate m over time.

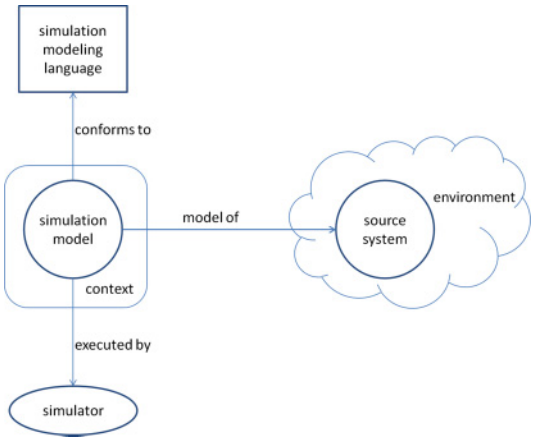

Fig. 4. M&S in general.

Definition 4.2 (*Simulation Modeling Language*). Let $l(g)$ be a modeling language. The language $l(g)$ is a simulation modeling language if and only if there exists a simulation model m such that $\gamma(m, l(g))$.

System specification formalisms can be used to define simulation models in a precise way. In the M&S field, the term *computer simulation model* often refers directly or indirectly to the final executable source code (PSM). Hence, simulation model code generation can be perceived as an M2C transformation.

5. THEORY OF THE MDD4MS FRAMEWORK

The MDD4MS framework proposes an MDD-based simulation model development method [Çetinkaya et al. 2011]. Similar to the stages in a simulation study explained in Section 2, the MDD4MS framework addresses the steps in a simulation study from conceptual modeling through final implementation by applying metamodeling and model transformations. In this way, it can be embedded into the existing M&S methodologies easily. Appendix B presents the MDD4MS lifecycle [Çetinkaya et al. 2011, 2013a; Çetinkaya 2013].

The MDD4MS framework introduces model and metamodel definitions for various stages, transformations between different models, methods to support the transformation steps, and a tool architecture for the overall process. When the model types in the M&S lifecycle are analyzed, it is most likely that these models need to be specified in different modeling languages. The MDD4MS framework introduces three intermediate models: a simulation conceptual model (CM), a platform-independent simulation model (PISM), and a platform specific simulation model (PSSM). In addition, it introduces three metamodels for specifying the CM, PISM, and PSSM to support model transformations:

—CMmetamodel represents the grammar of a conceptual modeling language.
—PISMmetamodel represents the grammar of a system specification formalism.
—PSSMmetamodel represents the grammar of a simulation model programming language.

CM is an instance-of CMmetamodel, PISM is an instance-of PISMmetamodel, and PSSM is an instance-of PSSMmetamodel. All of the metamodels are specified in a metamodeling language. Although it is practical to use the same metamodeling language

for all metamodels, it is also possible to use different metamodeling languages as long as there is a suitable model transformation language for each transformation.

The metamodels are expected to be specified in a metamodeling environment, where model editors can be generated automatically. By using these model editors, the models are specified as the instances of the metamodels. After introducing the metamodels, the MDD4MS framework introduces model transformations between the different models of the M&S lifecycle to automatically generate a subset of the simulation model source code. The following metamodel-based model transformations are proposed:

—CMtoPISM is an M2M transformation from CM to PISM.
—PISMtoPSSM is an M2M transformation from PISM to PSSM.
—PSSMtoSM is an M2C transformation from PSSM to SM.

Model transformations are orthogonal to metamodeling as shown in Figure 5. The figure illustrates the proposed framework with a single meta-metamodel. There are three metamodeling stacks, which are presented vertically in the figure. A practical application of the MDD4MS framework is an MDD process called an *MDD4MS process*. Although we propose an MDD4MS process with three models and one final set of executable source code, it is possible to increase the number of models in the framework.

In the CMtoPISM and PISMtoPSSM transformations, extra information needs to be added to obtain a full model. Without extra information, either the CM should contain a full specification, which is not likely, or the target model can only be partially generated. When moving from a CM to a PISM, more detailed behavior of the system should be added, and while moving from a PISM to a PSSM, the implementation details should be added. CMtoPISM transformation contains domain-specific constructs to add execution semantics. A PSSMtoSM transformation is generally a one-to-one transformation for a programming language and its metamodel. Defining precise metamodels and transformation rules are the challenging activities of MDD. If the modeling language is defined informally or semiformally, which is quite common for conceptual modeling languages, it is not possible to define a single and precise metamodel for that modeling language.

The expressiveness of the metamodel and the existence of component libraries are very important to be able to generate an executable and useful simulation model. Domain-specific modeling languages or component libraries can be used to increase the percentage of automatically generated code. In this case, the transformations can be written in a more generic and compact way since the domain knowledge is already added via the language elements or components [Huang 2013]. In addition, higher-level conceptual and platform-independent models have a high potential to support efficient compositions of components developed for different environments [Diallo et al. 2011; Tolk et al. 2012, 2013].

The domain-specific metamodels and model transformations need to consider the data-intensive nature of simulation as well. In the MDD4MS framework, data is added on the PISM level. It is expected that the modeler will enter the data either during the CMtoPISM model transformation or add it to the PISM model itself. It is also possible that the modeler needs to manually add data into the autogenerated code. In this case, data continuity can only be guaranteed if there are parameterized components on the lowest level, and all data is entered by specifying parameter values.

MDD4MS clearly separates the conceptual modeling, model formulation (specification), implementation, and model coding with the associated models CM, PISM, PSSM, and SM. However, according to the different needs in simulation projects, it is also possible to merge some stages by using the same modeling language to specify the different models (e.g., the same language can be used for CM and PISM in small scale projects). Then, the CM will be less detailed than the PISM. For example, an incomplete visual

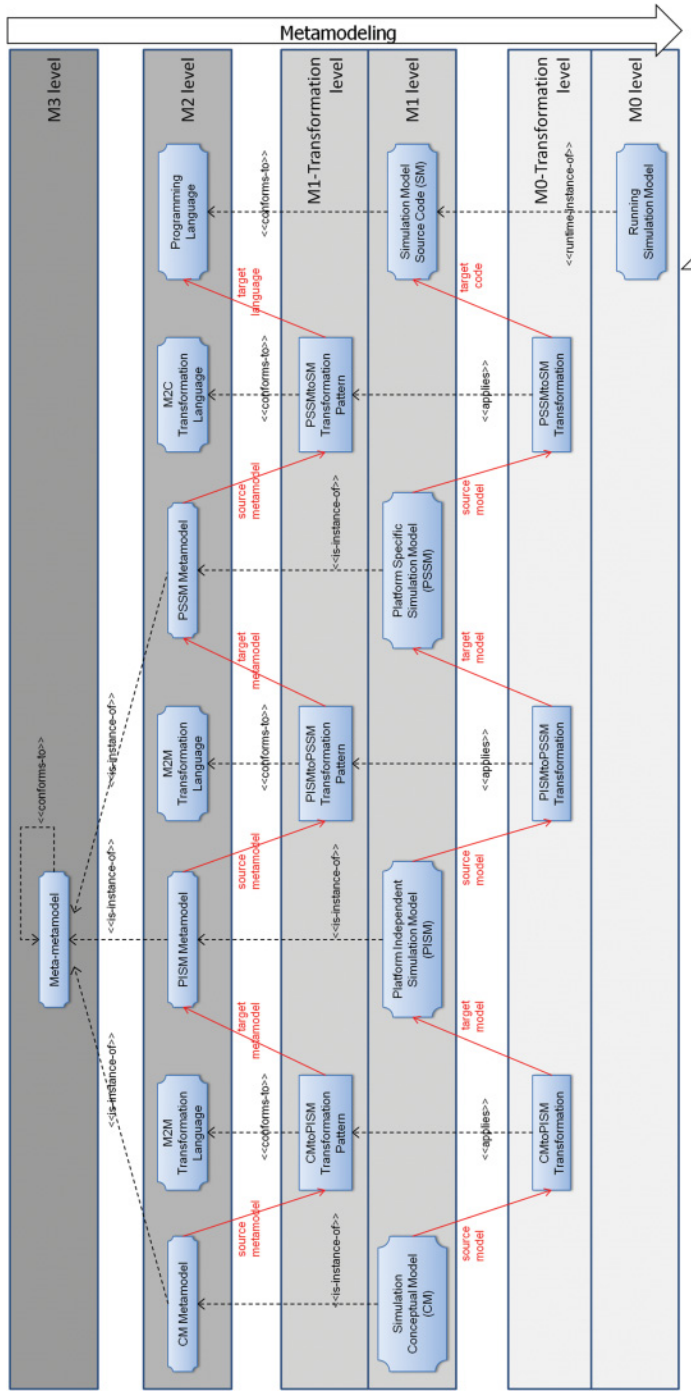

Fig. 5. MDD4MS framework: models, metamodels, and model transformations [Çetinkaya 2013].

DEVS diagram can serve as a conceptual model if the problem owner is familiar with DEVS. Besides, different model transformation patterns can be defined for a metamodel (e.g., a CM can be used to generate multiple PISMs, and a PISM can be used to generate multiple PSSMs). An implementation pattern for applying the MDD concepts in the M&S domain is given as follows.

Definition 5.1 (*MDD4MS Process with Three Models*). An MDD4MS process with three models is a specialized MDD process (Definition 3.20):

$$mdd4ms = \{n, MML, ML, MO, SL, pl, MTP, STP, MT, SM, TO\}$$

where

$n = 3$ (*CM, PISM, PSSM*);
$MML = \{l'_0, l'_1, l'_2\}$ is an ordered set of metamodeling languages (can be defined with meta-metamodels);
$ML = \{l_0(CMmetamodel), l_1(PISMmetamodel), l_2(PSSMmetamodel)\}$ such that

—$\gamma(CMmetamodel, l'_0)$,
—$\gamma(PISMmetamodel, l'_1)$, and
—$\gamma(PSSMmetamodel, l'_2)$;

$MO = \{CM, PISM, PSSM\}$ such that CM is the initial model, PSSM is the final model, and

—$\tau(CM) = CMmetamodel$,
—$\tau(PISM) = PISMmetamodel$, and
—$\tau(PSSM) = PSSMmetamodel$;

SL is a set of model transformation languages;
pl is a programming language with simulation capabilities;
$MTP = \{p_{cm}, p_{pism}, p_{pssm}\}$ such that

—$p_{cm} = \{l_0(CMmetamodel), l_1(PISMmetamodel), r_0\}$,
—$p_{pism} = \{l_1(PISMmetamodel), l_2(PSSMmetamodel), r_1\}$, and
—$p_{pssm} = \{l_2(PSSMmetamodel), pl, r_2\}$;

STP is a set of other supplementary formal model transformation patterns;
$MT = \{(\theta(CM, p_{cm}) = PISM), (\theta(PISM, p_{pism}) = PSSM), (\theta(PSSM, p_{pssm}) = SM)\}$;
SM is the final executable simulation model; and
TO is a set of tools to ease the activities.

Finally, we propose that any practical application of the MDD4MS framework obtains model continuity.

THEOREM 5.2. *An MDD4MS process with three models (performed according to the Definition 5.1) obtains model continuity.*

PROOF. For a given $mdd4ms = \{n, MML, ML, MO, SL, pl, MTP, STP, MT, SM, TO\}$, by Definition 5.1, where $n = 3$ and $MO = \{CM, PISM, PSSM\}$, we have

—$CM, PISM, PSSM \in M$,
—$p_{cm} = \{l_0(CMmetamodel), l_1(PISMmetamodel), r_0\}$,
—$p_{pism} = \{l_1(PISMmetamodel), l_2(PSSMmetamodel), r_1\}$,
—$p_{pssm} = \{l_2(PSSMmetamodel), pl, r_2\}$
—$\theta(CM, p_{cm}) = PISM$,
—$\theta(PISM, p_{pism}) = PSSM$,
—$\theta(PSSM, p_{pssm}) = SM$, and
—*SM* is the final executable simulation model.

We assume that CM is the initial model, $\mu(CM, s, c)$, and s is a system. Although $PSSM$ is the final model in software engineering, we accept that SM is the final model in M&S.

(1) $(\theta(CM, p_{cm}) = PISM) \wedge \mu(CM, s, c) \wedge PISM \in M \Rightarrow \mu(PISM, s, c + p_{cm})$ (by Theorem 3.14).
(2) $(\theta(PISM, p_{pism}) = PSSM) \wedge \mu(PISM, s, c + p_{cm}) \wedge PSSM \in M \Rightarrow \mu(PSSM, s, c + p_{cm} + p_{pism})$ (by 1 and Theorem 3.14).
(3) If SM is a simulation model, then $SM \in M$ (by Definition 4.1).
(4) $(\theta(PSSM, p_{pssm}) = SM) \wedge SM \in M \wedge \mu(PSSM, s, c + p_{cm} + p_{pism}) \Rightarrow \mu(SM, s, c + p_{cm} + p_{pism} + p_{pssm})$ (by 1, 2, 3 and Theorem 3.14).
(5) $n \geq 2 \wedge \mu(SM, s, c + p_{cm} + p_{pism} + p_{pssm}) \Rightarrow$ *mdd4ms process obtains model continuity* (by Definition 3.21). □

The next section presents an overview of a case study to apply MDD4MS in the discrete event simulation domain.

6. PROOF OF CONCEPT: DISCRETE EVENT SIMULATION OF BPMN MODELS

To show the applicability of the MDD4MS process, we have chosen a sample problem from the electronic payments sector that is presented in Sun et al. [2009]. The crucial role of M&S within this example is to document the business processes as much as possible in a visualized way to enable different parties to gain insight into the issues and into potential solutions. A manually developed simulation model is available [Sun et al. 2009] for comparison with our autogenerated model. We used the MDD4MS prototype implementation to develop the simulation model [Çetinkaya et al. 2013b]. The MDD4MS prototype implementation is an Eclipse-based application of the MDD4MS tool architecture, which is defined in the MDD4MS framework [Çetinkaya et al. 2011].

We used BPMN (Business Process Model and Notation) for conceptual modeling, DEVS for system specification, and Java for the simulation model code. Please note that these languages are used only to provide a proof of concept implementation. They are not part of the MDD4MS framework, and any other conceptual modeling language, system specification language, or simulation model programming language could have been chosen. For example, UML activity diagramming technique could be used as conceptual modeling language, Petri nets could be used as system specification language, and C++ could be used as simulation model programming language. In this case, new metamodels and transformation rules would have to be developed.

According to the MDD4MS framework, the following metamodels and model transformations are needed:

—BPMN metamodel as the CMmetamodel,
—DEVS metamodel as the PISMmetamodel,
—JAVA metamodel as the PSSMmetamodel,
—BPMNtoDEVS transformation as the CMtoPISM transformation,
—DEVStoJAVA transformation as the PISMtoPSSM transformation, and
—JAVAtoJAVACode transformation as the PSSMtoCode transformation.

Definition 6.1 (*MDD4MS Case Example*). MDD of DEVS-based simulation models from BPMN models is an MDD4MS process defined as

$$case = \{n, MML, ML, MO, SL, pl, MTP, STP, MT, SM, TO\},$$

where

$n = 3$ (*CM, PISM, PSSM*);
$MML = \{Ecore, Ecore, Ecore\}$ is the ordered set of metamodeling languages;
$ML = \{l_0(BPMNmetamodel), l_1(DEVSmetamodel), l_2(JAVAmetamodel)\}$ such that

—$\gamma(BPMNmetamodel, Ecore)$,
—$\gamma(DEVSmetamodel, Ecore)$, and
—$\gamma(JAVAmetamodel, Ecore)$;

$MO = \{CM, PISM, PSSM\}$ such that CM is the initial model, PSSM is the final model, and

—$\tau(CM) = BPMNmetamodel$,
—$\tau(PISM) = DEVSmetamodel$, and
—$\tau(PSSM) = JAVAmetamodel$;

$SL = \{ATL, JAVA\}$ is the set of model transformation languages;
$pl = JAVA$ is the programming language and it is extended with simulation libraries;
$MTP = \{p_{cm}, p_{pism}, p_{pssm}\}$ such that

—$p_{cm} = \{l_0(BPMNmetamodel), l_1(DEVSmetamodel), bpmn2devs.atl\}$,
—$p_{pism} = \{l_1(DEVSmetamodel), l_2(JAVAmetamodel), devs2java.atl\}$, and
—$p_{pssm} = \{l_2(JAVAmetamodel), JAVA, java2code.java\}$;

$STP = \{\}$ is the set of other supplementary formal model transformation patterns;
$MT = \{(\theta(CM, p_{cm}) = PISM), (\theta(PISM, p_{pism}) = PSSM), (\theta(PSSM, p_{pssm}) = SM)\}$;
SM is the final executable simulation model; and
$TO = \{Eclipse\ and\ a\ set\ of\ plugins\ (GEMS, ATL, PDE, EMF, GEF)\}$ is the set of tools to support the activities.

The MDD4MS prototype implementation includes metamodels, model editors, model transformation rules, and model interpreters for DEVS-based simulation of BPMN models [Çetinkaya et al. 2011, 2012, 2013b]. The BPMNtoDEVS transformation produces atomic and coupled models with ports, couplings, and templates for the system behavior. The DEVStoJAVA transformation produces valid Java visual models with information for Java classes. The JAVAtoJAVACode transformation generates Java code. To support the transformation process and to generate fully executable DEVS models, a DEVS simulation model component library for BPMN was used [Rust et al. 2011]. By using validated DEVS components, which are implemented in Java and executable with the DEVSDSOL simulation library, we ensured that the semantics of the DEVS model was preserved in the Java model. The case example showed that model continuity between the different models within the M&S lifecycle was obtained when the MDD4MS framework is applied. The metamodels specified in Ecore [Eclipse 2014] and more information about the case study can be found in Appendix C.

7. CONCLUSIONS

Applying MDD in M&S provides new capabilities for efficient development of reliable, error-free, and maintainable simulation models. Availability of tools and techniques for both metamodeling and model transformations is one of the practical advantages of MDD. Metamodeling provides a precise way for specifying models and their modeling languages.

This article proposes a comprehensive theoretical framework for MDD of simulation models. The framework recommends the use of three intermediate models in addition to the executable simulation model. We showed that the proposed MDD4MS framework obtains model continuity via formal, metamodel-based M2M transformations. The main objective of this research was to improve and speed up the simulation model development process by using software engineering methods. The most important outcomes of applying MDD in the M&S domain are providing syntactic and semantic validity of the models, formalizing the modeling steps and the process, increasing productivity, and providing model continuity.

The case study illustrated that the framework is applicable in the DEVS-based discrete event simulation domain. Considering that there are other successful studies on applying MDD and component-based approaches in the simulation field, and because MDD4MS provides a generic framework, we believe that MDD4MS is also applicable in other domains. Future work will therefore include testing the MDD4MS framework in different domains and in large-scale real-life M&S studies. In the presented case study, there was still some work for the modeler, such as adding data during the CMtoPISM model transformation and selecting appropriate components during the PISMtoPSSM model transformation. Further research will study possible extensions to the framework to be able to select automatically among alternatives at various levels and testing the quality of these choices.

MDD is different from traditional development approaches and requires a learning period and change of standard M&S development habits. When modeling languages for the different stages are not yet available and the team members have little or no knowledge about MDD, it may take quite some time to develop metamodels and model transformations. However, once these are developed, further development time and costs decrease significantly. A model-driven approach is therefore expected to have most benefits for multiple projects in the same domain, as well as for large-scale and critical simulation projects.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

The authors would like to thank to Associate Professor Joseph Barjis for his valuable comments on the case example. In addition, the authors appreciate the insightful suggestions and critical comments from the reviewers.

REFERENCES

Russell L. Ackoff. 1978. *The Art of Problem Solving*. John Wiley and Sons, New York, NY.

Carlos Agostinho, Filipe Correia, and Ricardo Jardim-Goncalves. 2010. Interoperability of complex business networks by language independent information models. In *Proceedings of the 17th International Conference on Concurrent Engineering*. 111–124.

Aditya Agrawal, Gyula Simon, and Gabor Karsai. 2004. Semantic translation of Simulink/stateflow models to hybrid automata using graph transformations. *Electronic Notes in Theoretical Computer Science* 109, 43–56.

Colin Atkinson and Thomas Kühne. 2002. Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation* 12, 4, 290–321.

Colin Atkinson and Thomas Kühne. 2003. Model-driven development: A metamodeling foundation. *IEEE Software* 20, 5, 36–41.

Amol Bakshi, Viktor K. Prasanna, and Akos Ledeczi. 2001. MILAN: A model based integrated simulation framework for design of embedded systems. In *Proceedings of the Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, New York, NY.

Osman Balci. 2001. A methodology for certification of modeling and simulation applications. *ACM Transactions on Modeling and Computer Simulation* 11, 4, 352–377.

Osman Balci. 2012. A life cycle for modeling and simulation. *Simulation* 88, 7, 870–883.

Osman Balci, James D. Arthur, and William F. Ormsby. 2011. Achieving reusability and composability with a simulation conceptual model. *Journal of Simulation* 5, 157–165.

Jerry Banks. 1998. *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*. John Wiley and Sons, New York, NY.

Russell R. Barton. 1998. Simulation metamodels. In *Proceedings of the 30th Winter Simulation Conference (WSC'98)*. IEEE, Los Alamitos, CA, 167–176.

Ola Batarseh and Leon F. McGinnis. 2012. SysML to discrete-event simulation to analyze electronic assembly systems. In *Proceedings of the Symposium on Theory of Modeling and Simulation: DEVS Integrative M&S Symposium*. Article No. 48.

Jean Bézivin. 2005. On the unification power of models. *Journal of Software and Systems Modeling* 4, 2, 171–188.

Jean Bézivin, Fabian Buttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. 2006. Model transformations? Transformation models! In *Model Driven Engineering Languages and Systems*. Lecture Notes in Computer Science, Vol. 4199. Springer, 440–453.

Deniz Çetinkaya. 2013. *Model Driven Development of Simulation Models: Defining and Transforming Conceptual Models into Simulation Models by Using Metamodels and Model Transformations*. Ph.D. Dissertation. Delft University of Technology, The Netherlands.

Deniz Çetinkaya, Saurabh Mittal, Alexander Verbraeck, and Mamadou D. Seck. 2013a. Model-driven engineering and its application in modeling and simulation. In *Netcentric System of Systems Engineering with DEVS Unified Process*, S. Mittal and J. L. Risco-Martín (Eds.). CRC Press, 221–248.

Deniz Çetinkaya and Alexander Verbraeck. 2011. Metamodeling and model transformations in modeling and simulation. In *Proceedings of the Winter Simulation Conference*. IEEE, Los Alamitos, CA, 3048–3058.

Deniz Çetinkaya, Alexander Verbraeck, and Mamadou D. Seck. 2010. A metamodel and a DEVS implementation for component based hierarchical simulation modeling. In *Proceedings of the 43rd Annual Simulation Symposium (ANSS'10)*. Article No. 170.

Deniz Çetinkaya, Alexander Verbraeck, and Mamadou D. Seck. 2011. MDD4MS: A model driven development framework for modeling and simulation. In *Proceedings of the Summer Computer Simulation Conference*. 113–121.

Deniz Çetinkaya, Alexander Verbraeck, and Mamadou D. Seck. 2012. Model transformation from BPMN to DEVS in the MDD4MS framework. In *Proceedings of the Symposium on Theory of Modeling and Simulation: DEVS Integrative M&S Symposium*. 304–309.

Deniz Çetinkaya, Alexander Verbraeck, and Mamadou D. Seck. 2013b. BPMN to DEVS: Application of MDD4MS framework in discrete event simulation. In *Netcentric System of Systems Engineering with DEVS Unified Process*, S. Mittal and J. L. Risco-Martín (Eds.). CRC Press, 609–636.

Andrea D'Ambrogio, Daniele Gianni, José Luis Risco-Martín, and Alessandra Pieroni. 2010. A MDA-based approach for the development of DEVS/SOA simulations. In *Proceedings of the Spring Simulation Multiconference*.

Julien DeAntoni and Jean-Philippe Babau. 2005. A MDA-based approach for real time embedded systems simulation. In *Proceedings of the 9th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'05)*. 257–264.

Saikou Y. Diallo, Heber Herencia-Zapana, Jose J. Padilla, and Andreas Tolk. 2011. Understanding interoperability. In *Proceedings of the Emerging M&S Applications in Industry and Academia Symposium*. 84–91.

Jaidermes Nebrijo Duarte and Juan de Lara. 2009. ODiM: A model-driven approach to agent-based simulation. In *Proceedings of the 23rd European Conference on Modelling and Simulation*.

Eclipse. 2014. Eclipse Modeling Framework (EMF). Retrieved January 3, 2014, from http://projects.eclipse.org/projects/modeling.emf.

Hartmut Ehrig and Claudia Ermel. 2008. Semantical correctness and completeness of model transformations using graph and rule transformation. In *Proceedings of the 4th International Conference on Graph Transformations (ICGT'08)*. 194–210.

Matthew J. Emerson, Janos Sztipanovits, and Ted Bapty. 2004. A MOF-based metamodeling environment. *Journal of Universal Computer Science* 10, 1357–1382.

Jean-Marie Favre. 2004. Towards a basic theory to model driven engineering. In *Proceedings of the International Workshop on Software Model Engineering (WISME'04)*.

Paul A. Fishwick. 1995. *Simulation Model Design and Execution: Building Digital Worlds*. Prentice Hall, Englewood Cliffs, NJ.

Peter Fritzson. 2010. The Modelica object-oriented equation-based language and its OpenModelica environment with metamodeling, interoperability, and parallel execution. In *Simulation, Modeling, and Programming for Autonomous Robots*. Lecture Notes in Computer Science, Vol. 6472. Springer, 5–14.

Alfredo Garro, Francesco Parisi, and Wilma Russo. 2013. A process based on the model-driven architecture to enable the definition of platform-independent simulation models. In *Simulation and Modeling Methodologies, Technologies, and Applications*. Advances in Intelligent Systems and Computing, Vol. 197. Springer, 113–129.

Eric Guiffard, Dahbia Kadi, Jean-Paul Mochet, and Régis Mauget. 2006. CAPSULE: Application of the MDA methodology to the simulation domain. In *Proceedings of the European Simulation Interoperability Workshop*. 181–190.

Xiaolin Hu and Bernard P. Zeigler. 2005. Model continuity in the design of dynamic distributed real-time systems. *IEEE Transactions on Systems, Man, and Cybernetics—Part A: Systems and Humans* 35, 6, 867–878.

Yilin Huang. 2013. *Automated Simulation Model Generation*. Ph.D. Dissertation. Delft University of Technology, The Netherlands.

IRISA. 2011. Kermeta workbench. Retrieved July 14, 2012, from http://www.kermeta.org/.

ISIS. 1997. Model Integrated Computing. Retrieved July 14, 2012 from http://www.isis.vanderbilt.edu/research/MIC.

Ethan Jackson and Janos Sztipanovits. 2009. Formalizing the structural semantics of domain-specific modeling languages. *Journal of Software and Systems Modeling* 8, 1, 451–478.

Frédéric Jouault and Jean Bézivin. 2006. KM3: A DSL for metamodel specification. In *Formal Methods for Open Object-Based Distributed Systems*. Lecture Notes in Computer Science, Vol. 4037. Springer, 171–185.

Aleksandr A. Kerzhner, Jonathan M. Jobe, and Christiaan J. J. Paredis. 2011. A formal framework for capturing knowledge to transform structural models into analysis models. *Journal of Simulation* 5, 202–216.

Jack P. C. Kleijnen. 2008. *Design and Analysis of Simulation Experiments*. Springer Science and Business Media, New York, NY.

Jack P. C. Kleijnen. 2009. Kriging metamodeling in simulation: A review. *European Journal of Operational Research* 192, 707–716.

Anneke Kleppe, Jos Warmer, and Wim Bast. 2003. *MDA Explained—The Model Driven Architecture: Practice and Promise*. Addison-Wesley, Boston, MA.

George J. Klir. 1969. *An Approach to General Systems Theory*. Litton Educational Publishing, New York, NY.

Kathy Kotiadis and Stewart Robinson. 2008. Conceptual modelling: Knowledge acquisition and model abstraction. In *Proceedings of the 40th Winter Simulation Conference (WSC'08)*. IEEE, Los Alamitos, CA, 951–958.

Thomas Kühne. 2006. Matters of (meta-) modeling. *Journal of Software and Systems Modeling* 5, 4, 369–385.

Averill M. Law. 2003. How to conduct a successful simulation study. In *Proceedings of the Winter Simulation Conference (WSC'03)*. 66–70.

Akos Ledeczi, James Davis, Sandeep Neema, and Aditya Agrawal. 2003. Modeling methodology for integrated simulation of embedded systems. *ACM Transactions on Modeling and Computer Simulation* 13, 1, 82–103.

Yonglin Lei, Wang Weiping, Li Qun, and Zhu Yifan. 2009. A transformation model from DEVS to SMP2 based on MDA. *Simulation Modelling Practice and Theory* 17, 1690–1709.

Andriy Levytskyy, Eugène J. H. Kerckhoffs, Ernesto Posse, and Hans Vangheluwe. 2003. Creating DEVS components with the metamodelling tool ATOM3. In *Proceedings of the 15th European Simulation Symposium*.

Leon F. McGinnis, Edward Huang, Ky Sang Kwon, and Volkan Ustun. 2011. Ontologies and simulation: A practical approach. *Journal of Simulation* 5, 190–201.

Dragan Milicev. 2009. *Model-Driven Development with Executable UML*. Wiley Publishing, Indianapolis, IN.

Saurabh Mittal and Scott A. Douglass. 2012. DEVSML 2.0: The language and the stack. In *Proceedings of the Symposium on Theory of Modeling and Simulation—DEVS Integrative M&S Symposium*. Article No. 17.

Saurabh Mittal, José Luis Risco-Martín, and Bernard P. Zeigler. 2007. DEVSML: Automating DEVS execution over SOA towards transparent simulators. In *Proceedings of the Spring Simulation Multiconference*. 287–295.

Saurabh Mittal, José Luis Risco-Martín, and Bernard P. Zeigler. 2009. DEVS/SOA: A cross-platform framework for net-centric modeling and simulation in DEVS Unified Process. *Simulation* 85, 7, 419–450.

Saurabh Mittal, Bernard P. Zeigler, José Luis Risco-Martín, Ferat Sahin, and Mo Jamshidi. 2008. Modeling and simulation for systems of systems engineering. In *System of Systems: Innovation for the 21st Century*, M. Jamshidi (Ed.). John Wiley and Sons, Hoboken, NJ, 101–149.

Pierre-Alain Muller, Frédéric Fondement, Franck Fleurey, Michel Hassenforder, Rémi Schnekenburger, Sébastien Gérard, and Jean-Marc Jézéquel. 2008. Model-driven analysis and synthesis of textual concrete syntax. *Journal of Software and Systems Modeling* 7, 4, 423–442.

Richard E. Nance. 1984. Model development revisited. In *Proceedings of the Winter Simulation Conference*. IEEE, Los Alamitos, CA, 74–80.

Antoni Olive. 2007. *Conceptual Modeling of Information Systems*. Springer-Verlag, Berlin, Heidelberg.

OMG. 1999. *UML Specification Version 1.3*. Technical Report. Object Management Group. Available at http://www.omg.org/spec/UML/1.3/.

OMG. 2003. *Model Driven Architecture (MDA) Guide Version 1.0.1*. Technical Report. Object Management Group. Available at http://www.omg.org/mda/specs.htm.

OMG. 2006. *Meta Object Facility (MOF) Core Specification, Version 2.0*. Technical Report. Object Management Group. Available at http://www.omg.org/spec/MOF/2.0/.

Tuncer I. Ören. 2007. The importance of a comprehensive and integrative view of modeling and simulation. In *Proceedings of the Summer Computer Simulation Conference*. 996–1006.

Christiaan J. J. Paredis, Yves Bernard, Roger M. Burkhart, Hans-Peter de Koning, Sanford Friedenthal, Peter Fritzson, Nicolas F. Rouquette, and Wladimir Schamai. 2010. An overview of the SysML-Modelica transformation specification. In *Proceedings of the INCOSE International Symposium*.

Christiaan J. J. Paredis and Thomas Johnson. 2008. Using OMG's SysML to support simulation. In *Proceedings of the 40th Winter Simulation Conference (WSC'08)*. 2350–2352.

Adrian Pop, David Akhvlediani, and Peter Fritzson. 2007. Towards unified system modeling with the ModelicaML UML profile. In *Proceedings of the International Workshop on Equation-Based Object-Oriented Languages and Tools*. Linköping University Electronic Press.

Axel Reichwein, Christiaan J. J. Paredis, Arquimedes Canedo, Petra Witschel, Philipp Emanuel Stelzig, Anjelika Votintseva, and Rainer Wasgint. 2012. Maintaining consistency between system architecture and dynamic system models with SysML4Modelica. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*. ACM, New York, NY, 43–48.

José Luis Risco-Martín, Jesús M. De La Cruz, Saurabh Mittal, and Bernard P. Zeigler. 2009. eUDEVS: Executable UML with DEVS theory of modeling and simulation. *Simulation* 85, 11–12, 750–777.

Nancy Roberts, David F. Andersen, Ralph M. Deal, Michael S. Garet, and William A. Shaffer. 1983. *Introduction to Computer Simulation: A System Dynamics Modeling Approach*. Productivity Press, Portland, OR.

Stewart Robinson. 2004. *Simulation: The Practice of Model Development and Use*. John Wiley and Sons, Hoboken, NJ.

Stewart Robinson. 2008a. Conceptual modelling for simulation Part I: Definition and requirements. *Journal of the Operational Research Society* 59, 3, 278–290.

Stewart Robinson. 2008b. Conceptual modelling for simulation Part II: A framework for conceptual modelling. *Journal of the Operational Research Society* 59, 3, 291–304.

Stewart Robinson. 2011. Choosing the right model: Conceptual modeling for simulation. In *Proceedings of the Winter Simulation Conference (WSC'11)*. 1423–1435.

Bernhard Rumpe. 1998. A note on semantics (with an emphasis on UML). In *Proceedings of the 2nd ECOOP Workshop on Precise Behavioral Semantics*. Technische Universität München, Germany.

Igor Rust, Deniz Çetinkaya, Mamadou D. Seck, and Ivo Wenzler. 2011. Business process simulation for management consultants: A DEVS-based simplified business process modelling library. In *Proceedings of the 23rd European Modelling and Simulation Symposium*.

Robert G. Sargent. 2010. Verification and validation of simulation models. In *Proceedings of the Winter Simulation Conference (WSC'10)*. IEEE, Los Alamitos, CA, 124–137.

Hessam S. Sarjoughian and Abbas Mahmoodi Markid. 2012. EMF-DEVS modeling. In *Proceedings of the Symposium on Theory of Modeling and Simulation: DEVS Integrative M&S Symposium*.

Wladimir Schamai, Peter Fritzson, and Chris J. J. Paredis. 2013. Translation of UML state machines to Modelica: Handling semantic issues. *Simulation* 89, 4, 498–512.

Wladimir Schamai, Peter Fritzson, Christiaan J. J. Paredis, and Adrian Pop. 2009. Towards unified system modeling and simulation with ModelicaML: Modeling of executable behavior using graphical notations. In *Proceedings of the 7th Modelica Conference*.

Wladimir Schamai, Uwe Pohlmann, Peter Fritzson, Christiaan J. J. Paredis, Philipp Helle, and Carsten Strobel. 2010. Execution of UML state machines using Modelica. In *Proceedings of the 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*.

Bran Selic. 2003. The pragmatics of model-driven development. *IEEE Software* 20, 5, 19–25.

Robert E. Shannon. 1975. *Systems Simulation: The Art and Science*. Prentice Hall, Englewood Cliffs, NJ.

Robert E. Shannon. 1998. Introduction to the art and science of simulation. In *Proceedings of the Winter Simulation Conference (WSC'98)*. 7–14.

Gregory A. Silver, Osama Al-Haj Hassan, and John A. Miller. 2007. From domain ontologies to modeling ontologies to executable simulation models. In *Proceedings of the 39th Winter Simulation Conference (WSC'07)*. 1108–1117.

Ian Sommerville. 2007. *Software Engineering* (8th ed.). Pearson Education Limited, England.

Jessica W. Sun, Joseph Barjis, Alexander Verbraeck, Marijn Janssen, and Jacco Kort. 2009. Capturing complex business processes interdependencies using modeling and simulation in a multi-actor environment. In *Advances in Enterprise Engineering III*. Lecture Notes in Business Information Processing, Vol. 34. Springer, 16–27.

Manos Theodorakis, Anastasia Analyti, Panos Constantopoulos, and Nicolas Spyratos. 2002. A theory of contexts in information bases. *Information Systems* 27, 151–191.

Andreas Tolk, Saikou Y. Diallo, and Jose J. Padilla. 2012. Semiotics, entropy, and interoperability of simulation systems: Mathematical foundations of M&S standardization. In *Proceedings of the Winter Simulation Conference (WSC'12)*. 1–12.

Andreas Tolk, Saikou Y. Diallo, Jose J. Padilla, and Heber Herencia-Zapana. 2013. Reference modelling in support of M&S: Foundations and applications. *Journal of Simulation* 7, 69–82.

Andreas Tolk and John A. Miller. 2011. Enhancing simulation composability and interoperability using conceptual/semantic/ontological models. *Journal of Simulation* 5, 133–134.

Andreas Tolk and James A. Muguira. 2004. M&S within the model driven architecture. In *Proceedings of the Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC)*.

Okan Topçu, Mehmet Adak, and Halit Oguztüzün. 2008. A metamodel for federation architectures. *ACM Transactions on Modeling and Computer Simulation* 18, 3, 10:1–10:29.

Luc Touraille, Mamadou Kaba Traoré, and David R. C. Hill. 2011. A model-driven software environment for modeling, simulation and analysis of complex systems. In *Proceedings of the Symposium on Theory of Modeling and Simulation: DEVS Integrative M&S Symposium*. 229–237.

C. Els van Daalen, Wil A. H. Thissen, Alexander Verbraeck, and Pieter W. G. Bots. 2009. Methods for the modeling and analysis of alternatives. In *Handbook of Systems Engineering and Management* (2nd ed.), A. P. Sage and W. B. Rouse (Eds.). John Wiley and Sons, Hoboken, NJ, 1127–1169.

Durk-Jouke van der Zee, Kathy Kotiadis, Antuela A. Tako, Mike Pidd, Osman Balci, Andreas Tolk, and Mark Elder. 2010. Panel discussion: Education on conceptual modeling for simulation—challenging the art. In *Proceedings of the Winter Simulation Conference (WSC'10)*. 290–304.

Hans Vangheluwe, Juan de Lara, and Pieter J. Mosterman. 2002. An introduction to multi-paradigm modelling and simulation. In *Proceedings of the AI, Simulation, and Planning in High Autonomy Systems Conference*.

Ludwig von Bertalanffy. 1968. *General System Theory: Foundations, Development, Applications (revised)*. George Braziller, New York, NY.

Gabriel A. Wainer. 2009. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. CRC Press, Boca Raton, FL.

Received April 2013; revised September 2014; accepted December 2014