Formalizing Model Transformations Within MDE

Zhi Zhu(✉), Yongling Lei, Qun Li, and Yifan Zhu

National University of Defense Technology, Changsha 410073, China
zhuzhi@nudt.edu.cn

Abstract. A recent approach to tackle the ever increasing complexity of military simulation system is model-driven engineering (MDE). However, it is used mostly to produce simulation software tools, and seldom can perform formal analysis on models, resulting in a low degree of simulation model engineering. Consequently, this raises many issues such as inefficient development as well as poor qualities of product, and falls short of non-functional requirements like extensibility, maintainability, and reuse. In general, many of the success of MDE are dependent on the descriptive power of modeling languages and how conceptual models are transformed toward final implementations. Hence, this paper presents contributions in two main aspects of MDE: customizing domain specific language by metamodeling and engineering model continuity by formalizing model transformations. A military simulation application called group fire control channel system is used as a motivating example to illustrate the whole process, transforming conceptual models into other formalisms that have precise definitions of semantics until they reach final executable simulation models.

Keywords: MDE · Metamodeling · Model transformation · Model continuity

1 Introduction

Traditional military simulation models are usually represented by UML which has not precise and unambiguous semantics defined using a mixture of OCL (Object Constraint Language) and informal text, or the semantics of simulation models are left to model interpreters or simulators which are defined by general-purpose programming languages, which is clearly unacceptable for formal analysis [1]. Meanwhile, although the syntax of current domain specific modeling languages (DSML) are formally described with a lot of general metamodeling tools like UML Profile [2], EMF [3], and GME [4] etc., the semantics are left toward other less than desirable means [5]. All of these accompanying with the lack of formal model transformations contribute to difficult formal analysis at a model level. Hence, it is a real challenge to describe simulation models formally, and to improve the model continuity that exist between different models in different development stages at different levels of abstraction [6], so as to reuse existing model assets and simulation services to a great degree.

Inconsistent terminology in the model-driven engineering (MDE) context [7] means it is necessary to define basic meanings of important frequently used terms to

H. Song et al. (Eds.): SIMUtools 2019, LNICST 295, pp. 25–42, 2019.
https://doi.org/10.1007/978-3-030-32216-8_3

provide a common understanding. Many of these terms are used alternatively in specific contexts, but providing their definitions and/or subtle distinctions is helpful to understand the methodologies, techniques, and tools used in model-driven development. For different modeling goals, there exist three typical issues, i.e. model composability, model heterogeneity, and model continuity.

Firstly, model composability [8] concentrates on the syntactical matching and semantic relations between different simulation models. Unlike the other two issues, it is usually discussed in a MDE context and emphasizes the integration of multiple simulation models to form an effective and meaningful simulation application.

Secondly, model heterogeneity comes from the joint use of several DSMLs dedicated to particular domains or applications. In many cases, it refers to the syntactical incompatibility between different used DSMLs during the language customization process and has four sources in general [9, 10]. First, the different technical or application domains involved in a simulation system under design require different model specifications, modeling formalisms, or simulation protocols [11]. Second, the different levels of abstraction need suitable modeling techniques. Third, a simulation system is always studied from different points of view, and lastly different stages of a development cycle may use different languages for different activities.

Thirdly, Model continuity refers to the generation of an approximate morphism relation between different phases of a development process [12]. In general, 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.

In a sense, model continuity is similar to the consistency between the source and target models, involving the syntactical correctness of target model, the completeness of source model consumed in model transformation, and the semantic relations preserved in target model [13]. We use M_{com} to represent model composability, M_h to model heterogeneity, and M_{con} to model continuity, such that $M_h \cup M_{con} \subset M_{com}$ and $M_h \cap M_{con} \neq \varnothing$, as shown in Fig. 1. It means, on the one hand, if simulation models in a simulation application development satisfy the model composability, then it also satisfy the model heterogeneity and model continuity. On the other hand, model heterogeneity and model continuity may not be disjoint in some cases, which means model heterogeneity is somehow equal to the syntactical discontinuity between different development stages.

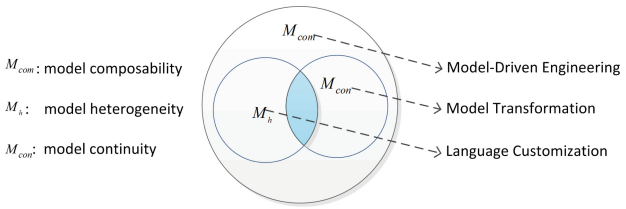

Fig. 1. Three typical issues identified in the MDE context.

This paper proposes a set of formal theories of model transformations for engineering model continuity, transforming models represented by various modeling languages into other formalisms that have precise definitions of semantics until they reach final executable simulation models [14]. A motivating example named group fire control channel system (GFCCS) [15] is used through this paper, commencing with its customization of DSML and transforming its conceptual models represented in this DSML to final executable simulation models. After that, a military simulation system in support of engineering modeling and composable simulating is capable of integrating those executable simulation models and reusing them for multiple simulation applications.

2 Model Transformations

2.1 The Basic Mode of Model Transformation

Model transformation is a process that takes a source model in a specific form as inputs and outputs another form of the target model according to a set of predefined rules. This process does not build new models from scratch, but reuse existing information when conducting a model transformation. A formal model transformation requires that the models involved in the transformation are represented clearly by well-defined modeling languages that have accurate syntax and unambiguous semantics. Furthermore, it requires that the transformation rules are written by a well-defined transformation language to ensure the transformation is conducted under a well-defined transformation template [16]. To ensure model continuity, the target model should preserve as much as possible the initial model information and modeling relations that are embedded in the source model [17].

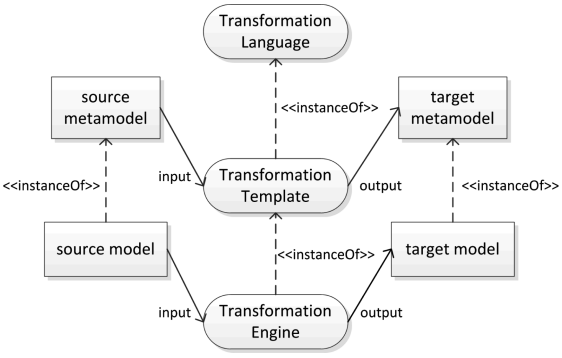

Fig. 2. The basic mode of model transformation.

Figure 2 shows the basic mode of model transformation. In this mode, each node at a certain layer conforms to or is an instance of the node at a higher layer. The middle column is the transformation mechanism that inputs the left source nodes and outputs

the right target nodes. For example, the transformation engine is an instance of the transformation template which is further an instance of the transformation language, which means the transformation template is written by the transformation language and prescribes the internal mechanism of the transformation engine. This engine inputs the source model which is an instance of the source metamodel and outputs the target model which is an instance of the target metamodel, and both metamodels are respectively taken as the inputs and outputs of the transformation template.

According to the concrete form of target model, model transformation has two typical categories: model to model (M2M) and model to text (M2T). In practice, M2T transformation is also called code generation when the text is in the form of source code. In general, a model-driven development process contains a sequence of M2M transformations and a final code generation. In addition, model transformation is endogenous when the source metamodel is similar to the target metamodel, and exogenous when they are different [18].

2.2 MDA Based Model Transformations

MDA (model-driven architecture) introduces three model development roles, and two transform mechanism types [19]. Using these MDA models, i.e. conceptual independent (CIM), platform independent (PIM), and platform specific models (PSM), developers can be classified into comparable roles, i.e. conceptual and simulation modelers, and simulation programmers, respectively, where later stages only can commence when developers for the former stages reach a consensus on an artifact. For example, once the problem owner and the conceptual modeler agree on a conceptual model, the simulation modeler can transform it into a formal model. In addition, M2M and M2T model transformation mechanisms are used as a bridge to reduce the gap between these roles. We adopt the formal MDA process as depicted in Definition 1 [12].

Definition 1. A MDA process is defined as

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

- $n = 3(CIM, PIM, PSM)$,
- $MML = \{ll_0, ll_1, ll_2\}$ is an ordered set of metamodeling languages,
- $ML = \{l_0(mm_{CIM}), l_1(mm_{PIM}), l_2(mm_{PSM})\}$ such that
  $confromTo(mm_{CIM}, ll_0)$,
  $confromTo(mm_{PIM}, ll_1)$,
  $confromTo(mm_{PSM}, ll_2)$,

This means metamodels must conform to their corresponding metamodeling languages.

- $MO = \{CIM, PIM, PSM\}$ such that CIM is the initial model, PSM is the final model, and

$instanceOf(CIM) = mm_{CIM}$,
$instanceOf(PIM) = mm_{PIM}$,
$instanceOf(PSM) = mm_{PSM}$,

This means models must conform to their corresponding metamodels.

- SL is a set of model transformation languages,
- pl is a programming language with simulation capabilities,
- $MTP = \{p_{CIM}, p_{PIM}, p_{PSM}\}$ such that
  $p_{CIM} = \{l_0(mm_{CIM}), l_1(mm_{PIM}), r_0\}$,
  $p_{PIM} = \{l_1(mm_{PIM}), l_2(mm_{PSM}), r_1\}$,
  $p_{PSM} = \{l_2(mm_{PSM}), pl, r_2\}$,

This represents model transformation patterns that a source language to a target language through some rules.

- STP is a set of other supplementary formal model transformation patterns,
- $MT = \{$
  $transformTo(CIM, p_{CIM}) = PIM$,
  $transformTo(PIM, p_{PIM}) = PSM$,
  $transformTo(CIM, p_{PSM}) = SM$
  $\}$,

This means model transformations that a source model to a target model using some patterns.

- SM is the final executable simulation model,
- TO is a set of tools to ease the activities.

Above definition is suitable for general model development base on the MDA principles. Given this definition, we can conclude a process for the GFCCS development as Definition 2, which will be illustrated by later sections. In the GFCCS process, we take the GFCCS DSML as the conceptual modeling language to describe CIM, P-DEVS [20] as the modeling formalism to define PIM, and JAVA as the programming language to build PSM. Hence, the GFCCS process involves the following types of metamodels and model transformations.

1. The CIM metamodel is GFCCS metamodel
2. The PIM metamodel is P-DEVS metamodel
3. The PSM metamodel is JAVA metamodel
4. The CIM-PIM transformation is GFCCS to P-DEVS transformation
5. The PIM-PSM transformation is P-DEVS to JAVA transformation
6. The PSM-SM transformation is JAVA to java code transformation.

Definition 2. A GFCCS simulation modeling process is defined as

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

- $n = 3(CIM, PIM, PSM)$,
- $MML = \{Ecore, Ecore, Ecore\}$ is an ordered set of metamodeling languages,
- $ML = \{l_0(mm_{GFCCS}), l_1(mm_{DEVS}), l_2(mm_{JAVA})\}$ such that
  $conformsTo(mm_{GFCCS}, Ecore)$,
  $conformsTo(mm_{DEVS}, Ecore)$,
  $conformsTo(mm_{JAVA}, Ecore)$,
- $MO = \{CIM, PIM, PSM\}$, and
  $instanceOf(CIM) = mm_{GFCCS}$,
  $instanceOf(PIM) = mm_{DEVS}$,
  $instanceOf(PSM) = mm_{JAVA}$,
- $SL = \{ATL, Acceleo\}$ is a set of model transformation languages,
- $pl = JAVA$ is the final programming language
- $MTP = \{p_{CIM}, p_{PIM}, p_{PSM}\}$ such that
  $p_{CIM} = \{l_0(mm_{GFCCS}), l_1(mm_{DEVS}), gfccs2devs.atl\}$,
  $p_{PIM} = \{l_1(mm_{DEVS}), l_2(mm_{JAVA}), devs2java.atl\}$,
  $p_{PSM} = \{l_2(mm_{JAVA}), JAVA, java2code.mtl\}$,
- $STP = \varnothing$ dictates there exists no other supplementary formal model transformation patterns,
- $MT = \{$
  $transformTo(CIM, p_{CIM}) = PIM$,
  $transformTo(PIM, p_{PIM}) = PSM$,
  $transformTo(CIM, p_{PSM}) = SM$
  $\}$,
- SM is the final executable simulation model,
- $TO = \{ATL, Acceleo, EMF, GMF, Eclipse\_IDE\}$.

2.3 Criteria for Evaluating Model Continuity

Model transformation is an automated process of modifying and creating one or several target models from one or several source models. The aim of model transformation is to save effort and reduce information loss as much as possible by automating model building and modification where possible. The key to designing a successful model transformation is a set of formal transformation rules to improve model continuity. Although there is no general guidance to define a good model transformation, we can evaluate model continuity according to the following criteria.

1. Correctness. A model transformation is syntactically correct if the target model conforms to the target metamodel specification [21], and semantically correct if the target model contains information as much as possible from the source model [22].
2. Completeness. A model transformation is complete if the target model has a corresponding element for each element in the source model.
3. Uniqueness. A model transformation is unique if there are no two identical elements in the generated target model.
4. Determinism. A model transformation is determinate if it produces a uniquely defined target model output for each specific source model input.

3 Customizing a DSML Based on Metamodeling

3.1 Metamodeling Based on EMF

Metamodeling is an important mean to design DSMLs [23, 24], especially for EMF usually has close relationships with a set of OMG standards, like UML, MOF, XMI, and MDA, etc. Firstly, UML is widely used to capture various concerns of a certain system by an object-oriented method, emphasizing multi-view to describe the structure, behavior, function, and deployment, etc. While, EMF as a way of defining metamodels is only concerned with one aspect of a system, i.e. class structure. Secondly, EMF/Ecore focuses on the tool sets not the metadata warehouse management, thus avoiding some of the complex issues such as data structure, package relationships, and associations compared to MOF. Thirdly, XMI is a widely accepted serializing standard which is not only used as the format for serializing EMF models, but also suitable for serializing the metamodel, i.e. Ecore itself. This method is very different with the UML profiling mechanism because it defines metamodels from scratch without considering the UML rules [25]. Hence, it has the potential for the most direct and succinct expression of domain concepts. Furthermore, it has a collection of supporting tools (e.g. GEF and GMF) thanks to the Eclipse open source architecture. Recently, some researches also have identified the need of domain specific metamodeling to avoid the general metamodeling facilities like UML and EMF [26].

Figure 3 shows the metamodel of GFCCS. This metamodel consists of a basic diagram node named GroupFireControlSystem and two mutually related nodes named Node and Connection respectively. The Node derives a set of domain concepts such as Group, GroupNode, Weapon, Target, and Channel, which are connected by specific relationships. For example, two groups can share common information by the relation tagged as COPShare that represents common operation picture (COP) [27]. A group can have one or multiple members and zero or multiple weapons which can also be equipped by a group member. A group member may be disjoint with or affiliated by itself, and can control zero or multiple fire control channels. Each channel may be mutually exclusive with itself. It embraces two dynamic entity lists, i.e. weaponList and targetList. These two lists are used to manage weapons and targets that are alive or may be already ruined. Only one target can be assigned to one weapon for building a running fire control channel.

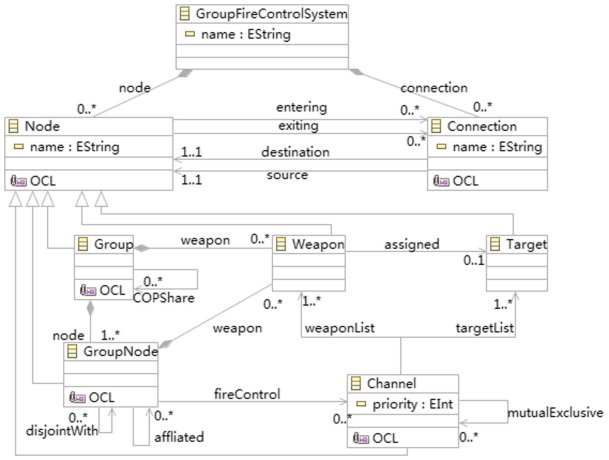

Fig. 3. The GFCCS metamodel.

In fact, except the abstract syntax as described above, there are other aspects need to be detailed for a well-defined metamodel [28]. Table 1 defines the static semantics of GFCCS metamodel, written by OCL [29], explaining how those elements of the abstract syntax model can be organized as a valid GFCCS metamodel.

Table 1. GFCCS domain specific constraints using OCL (part).

OCL static semantics	Descriptions
context *Group* inv*: hasNotDisjointGroupNodes* self*.node->forAll(n1,n2|n1.disjointWith->select(dis|dis.name=n2.name)->isEmpty() and n2.disjointWith->select(dis|dis.name=n1.name->isEmpty())*	A group can never own two disjoint members.
context *GroupNode* inv*: hasNotDisjointChannels* self*.fireControl->forAll(c1,c2|c1.mutualExclusive->select(dis.name=c2.name)->isEmpty() and c2.mutualExclusive->select(dis|dis.name=c1.name)->isEmpty())*	A group node can never own two mutual exclusive fire control channels.
context *GroupNode* inv*: notDisjointWithItself* self*.disjointWith->select(dis|dis.name=self.name)->isEmpty()*	No group node can be disjoint with itself.
context *GroupNode* inv*: notDisjointWithItsDownLevelNode* self*.affliated->forAll(n|self.disjointWith->select(dis|dis.name=n.name)->isEmpty())*	No group node can be disjoint with its senior node.

3.2 Graphical Definitions of GFCCS Using GMF

Figure 4 shows a guidance of the definition, mapping, and generation of a graphical editor for GFCCS using GMF. According to the GMF dashboard, one can define the domain model, domain gen model, tooling model, graphical model, mapping model, and gmf gen model step by step, then generate the diagram editor.

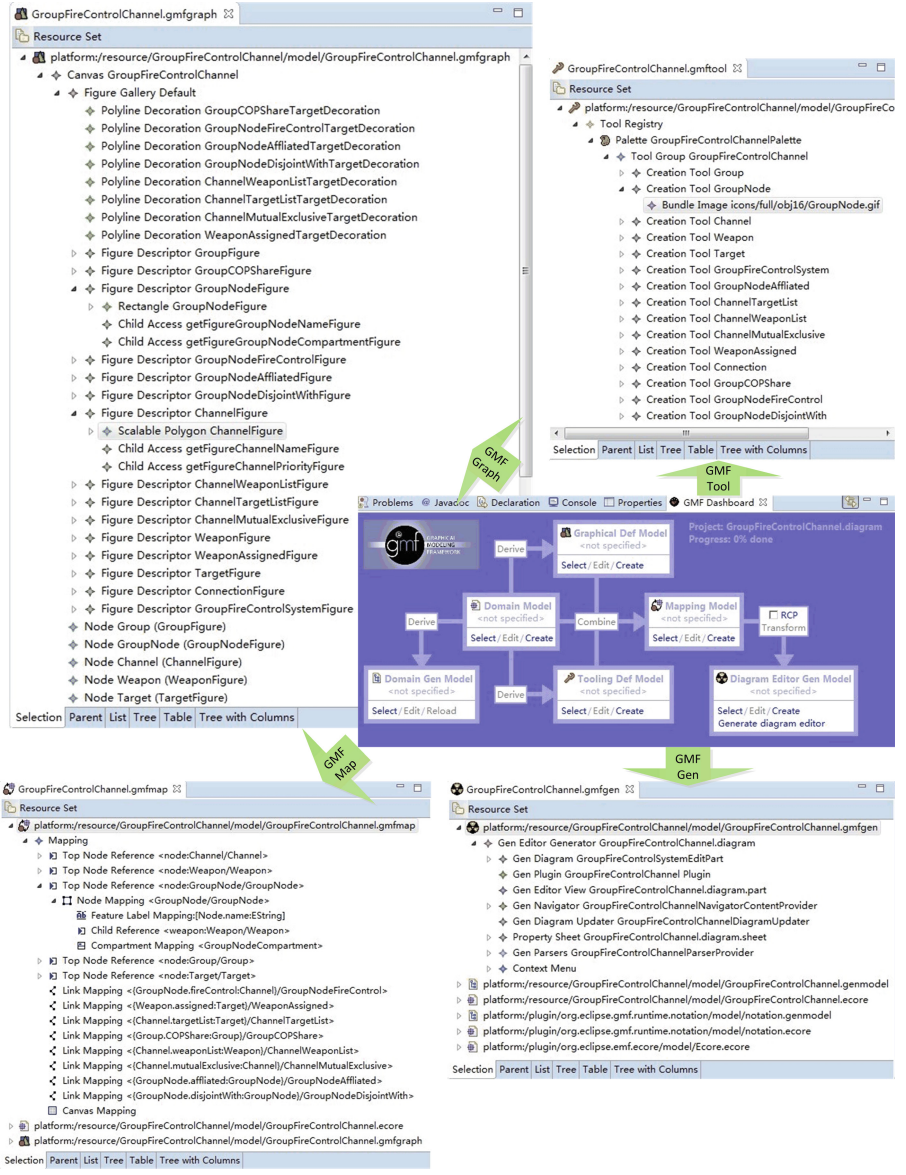

Fig. 4. Graphical definitions of GFCCS using GMF.

1. Tooling model. In usual, the tooling model definition provides six ways to define a tool palette of a graphical editor, including creation tool, standard tool, generic tool, tool group, palette separator, and image. In GFCCS, we created a tool for each element of the domain model except the abstract element Node, and bundled a representative image for each element.
2. Graphical model. It defines the concrete display of modeling elements that will be used in the graphical editing environment. In general, GMF provides default display based on the domain model, but one usually needs to define the figure gallery, figure descriptor, and polyline decoration in practice. In GFCCS, we set the Group and GroupNode as compartments to be able to contain other elements, for example, Group can contain GroupNode and Weapon, and GroupNode can contain Weapon. Additionally, we set the Channel as a scalable polygon, adding template points (0, 0), (40, 0), (40, 30), (30, 30), (30, 40), (40, 30), (30, 40), (0, 40).
3. Mapping model. When the domain model, tooling model, and graphical model are ready, it is necessary to map them into a whole. Usually, we need to select the corresponding tooling nodes and diagram nodes for each node mapping, and provide the correct compartment figure for each compartment node. In the properties of Channel node, for example, we select the Node Channel (Channel Figure) for the diagram node, and the Creation Tool Channel for the tooling node.
4. GMF gen model. If the mapping model is defined correctly, it can generate correct gmf gen model without much modifications. In many cases, it is possible to modify some parameters, such as the fixed background, the list layout, and the suffix of a diagram project.

3.3 The GFCCS DSME

Figure 5 shows a simple example of building an engagement scenario using the GFCCS DSML. On the tool palette, this domain specific modeling environment (DSME) contains a set of buttons decorated with professional denotations, including the basic language elements such as GroupFireControlSystem, Group, GroupNode, Channel, Weapon, and Target as well as various relationships. Using this environment, it is possible for domain experts to use these language elements intuitionally and friendly. For example, one can draw an arrow from Channel List 1 to Enemy Fighter 1 BLUE only by ChannelTargetList, disabling the use of other relationship buttons.

In the editor, we create a scenario of many to many combat between two opposite sides RED and BLUE. The RED side consists of two defense groups, i.e. Defense System RED and Remote Surveillance System RED, which refers to the local defense system (e.g. air defense base) and the remote surveillance system (e.g. satellite). The local system is composed of a warship platform, a ground-to-air missile base, and a helicopter platform, which are armed with two surface-to-air missiles, a ground-to-air missile, and a homing torpedo, respectively. The BLUE side consists of three coming threats which are denoted by Enemy Fighter 1 BLUE, Enemy Fighter 2 BLUE, and Enemy Submarine BLUE. In addition, there exist two lists of fire control channel which are denoted by Channel List 1 and Channel List 2, respectively. The former list is

managed by the warship platform and has a weapon-target pair, i.e. Surface2AirMissile1-Enemy Fighter 1 BLUE. The latter list is managed by the helicopter platform and has two weapon-target pairs.

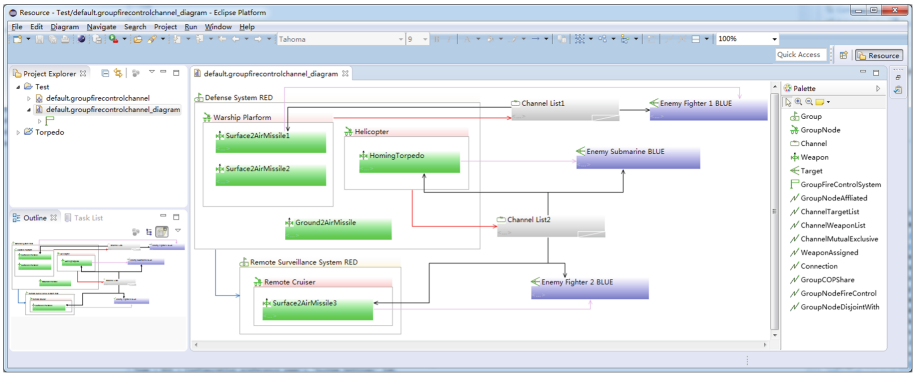

Fig. 5. The GFCCS DSME. (Color figure online)

4 Model Transformations: GFCCS Implementations

4.1 The Process of GFCCS Implementation

Assume we need to build the process of GFCCS implementation as illustrated in the conceptual sample of Fig. 6. Our aim is therefore to define a set of transformation rules facilitating the transformation from CIM to PIM to PSM, then to final source code. The transformation has two kinds across three levels.

In the first kind, all CIM_GFCCS model elements are expected to be transformed into specific PIM_P-DEVS model elements, and all connections are transformed into internal couplings from an output port in the source component to an input port in the target component. Ports are also generated. But in most cases, this does not apply because the connections in the source model do not always connect the elements of the same layer [30]. Therefore, it is necessary to refine such connections which cross more than one layer, and define the external input couplings (EICs) and external output couplings (EOCs) for the compartmental components. For example, while the group named Defense System RED and the group nodes named Warship Platform and Helicopter are transformed into coupled components, the weapon named Surface2AirMissile1 is transformed into an atomic component.

In the second kind, a transformation from PIM_P-DEVS to PSM_JAVA is defined according to a set of predefined transformation rules. In this transformation, all P-DEVS components are transformed into JAVA classes. Coupled components are transformed into files that include the package imports, class, constructor, port, contained component, and couplings. Atomic components are transformed into files that include package imports, class, port, and constructor.

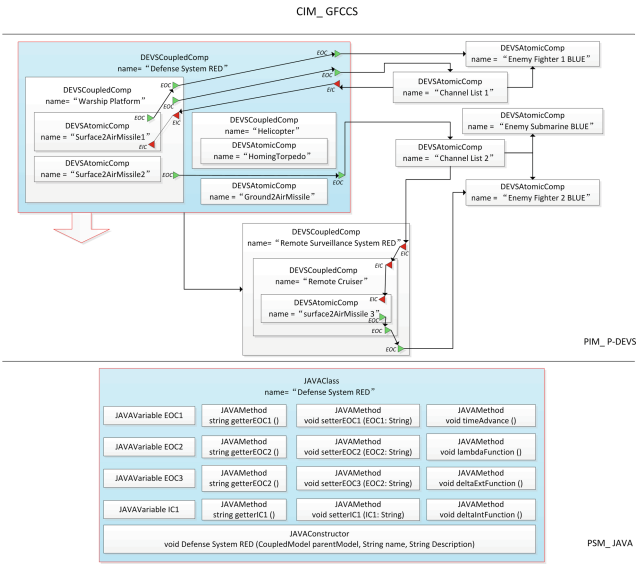

Fig. 6. A sample of GFCCS to P-DEVS to JAVA transformations based on MDA.

4.2 Detailed GFCCS to P-DEVS to JAVA Transformations

The matching rules of GFCCS to P-DEVS transformation is defined by using the GFCCS metamodel and P-DEVS metamodel, as detailed in Table 2. The basic diagram node GourpFireControlSystem matches with the DEVSModel element. The compartmental nodes like Group and GroupNode match with coupled components named DEVSCoupledComp. The connections that connect the modeling elements of the same layer are transformed into internal transitions named DEVSOutToIn_ICConnection, while those cross different layers are transformed into external transitions and output functions with a set of ports.

Table 2. The matching rules of GFCCS to P-DEVS transformation (part).

GFCCS metamodel	P-DEVS metamodel
GroupFireControlSystem	DEVSModel
Group	DEVSCoupledComp
Weapon	DEVSAtomicComp
Target	DEVSAtomicComp
COPShare	DEVSOutToIn_ICConnection
fireControl	DEVSOutToOut_ICConnection +Source.out: DEVSOutputPort +Target.in: DEVSOutputPort +SourceParents.EOCPorts: DEVSOutputPort
mutualExclusive	DEVSOutToIn_ICConnection
taretList	DEVSOutToIn_ICConnection

The matching rules of P-DEVS to JAVA transformation is defined by using the P-DEVS metamodel and JAVA metamodel, as detailed in Table 3. The basic element DEVSModel matches with a JAVA package named JAVAPackage which includes javaClasses, javaConstructors, and javaExpressions. Both the coupled component DEVSCoupledComp and the atomic component DEVSAtomicComp match with JAVA classes which include JAVAVariables, javaConstructors, and javaExpressions. DEVSInputPort, DEVSOutputPort, and StateVariable are all transformed into JAVAVariables. The connections like DEVSOutToIn_ICConnection, DEVSInToIn_EICConnection, DEVSOutToOut_EOCConnection as well as Expression are transformed into JAVAExpressions. The remaining functions like DeltaIntFunction, DeltaExtFunction, LambdaFunction, TimeAdvanceFunction, DeltaConFunction are all transformed into JAVAMethods.

Such transformation rules are written in ATL, as proposed in Definition 2. ATL, the Atlas Transformation Language, is a model transformation language specified as both a metamodel and a textual concrete syntax. In the MDE field, ATL provides developers with a means to specify the way to produce a number of target models from a set of source models. An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models. Besides, ATL Integrated Development Environment (IDE) provides a number of standard development tools (syntax highlighting, debugger, etc.) that aim to ease the design of ATL transformations. The ATL development environment also offers a number of additional facilities dedicated to models and metamodels handling. These features include a simple textual notation dedicated to the specification of metamodels, but also a number of standard bridges between common textual syntaxes and their corresponding model representations.

Table 3. The matching rules of P-DEVS to JAVA transformation (part).

P-DEVS metamodel	JAVA metamodel
DEVSModel	JAVAPackage +javaClasses: JAVAClass +javaConstructors: JAVAConstructor +javaExpressions: JAVAExpression
DEVSInputPort	JAVAVariable
DEVSOutputPort	JAVAVariable
StateVariable	JAVAVariable
DEVSOutToIn_ICConnection	JAVAExpression
DEVSInToIn_EICConnection	JAVAExpression
DEVSOutToOut_EOCConnection	JAVAExpression
Expression	JAVAExpression
DeltaIntFunction	JAVAMethod
DeltaExtFunction	JAVAMethod
LambdaFunction	JAVAMethod
DeltaConFunction	JAVAMethod

Figure 7 shows the transformations of GFCCS to P-DEVS to JAVA instances based on ATL. On the left contains two projects named GFCCS2P-DEVS and P-DEVS2JAVA, each of which includes three packages, i.e. Metamodels, Models, and TransoformationEngine. On the upper right includes three instances for the GFCCS, P-DEVS, and JAVA metamodel respectively, while the lower right is two ATL files for the GFCCS to P-DEVS and to JAVA transformations.

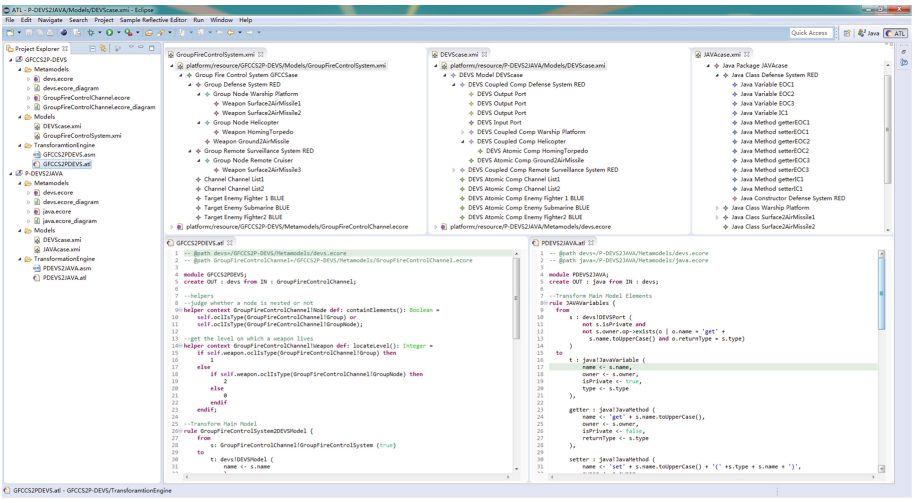

Fig. 7. GFCCS to P-DEVS to JAVA transformations based on ATL.

4.3 Source Code Generation

Figure 8 shows a screenshot of the M2T transformation model as well as its source model and the generated code framework for the node Defense_System_RED. The transformation model design should incorporate all the source model required information to satisfy completeness. The target code framework is automatically generated, but the concrete logic details must be manually implemented. In practice, not every concrete detail should be considered when designing M2T transformation models, because they may heavily burden the design phase.

Following the M2T transformation principles, the general source model is some instance models that must conform to a certain metamodel, and the target model can be text, e.g. java, C++, python, etc. Transformation model design is vital to implement the M2T transformation. This paper performed M2T transformation using Acceleo, a template based code generator incorporating a code generation editor with syntax highlighting, completion, real time error detection, and refactoring. The source model was a collection of JAVA instance models, represented by an instance file named JAVAcase.xmi, and the target model was described in Java programming text.

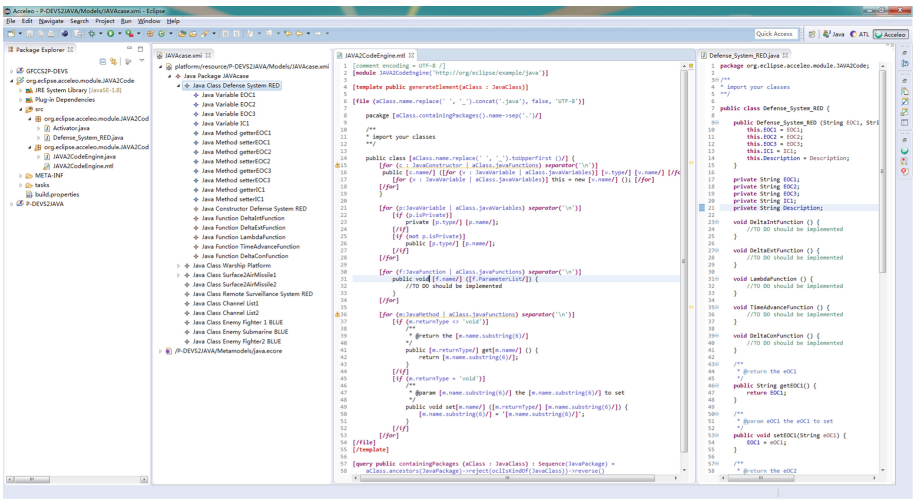

Fig. 8. JAVA xmi to code transformation based on Acceleo.

4.4 Model Continuity in GFCCS

The process of GFCCS implementation showed that model continuity between different development stages is obtained when applying the formal transformation definition successfully. As stated earlier, it is possible to provide model continuity in a development process when transforming the initial and intermediate models, and preserving the modeling relations during the transformations. To evaluate model continuity, we already presented the criteria for model transformations in Sect. 2.3. According to these criteria, we describe how the model continuity is obtained for the process of GFCCS implementation when the formal transformation rules are applied. In fact, except these criteria as listed, there are some other non-functional requirements such as termination and readability satisfied. Also note that maintainability, scalability, reusability, evolvability, efficiency, etc. are partially supported since these requirements need more experiments for a better evaluation.

The process of GFCCS implementation has two kinds of model transformation, with different expressions and output types. As the source input or target output of model transformations, the formal definition gives three model types: independent of computing details, independent of the computing platform, and specific to a particular computing platform, and two model transformation categories.

In the M2M category, the transformation focuses on the design of a set of formal rules to ensure model continuity when transforming CIM to PIM and to PSM. The transformation usually incorporates three steps.

1. All source concepts, relationships, and domain specific rules are transformed into particular target elements, connections, and domain specific constraints, respectively.
2. Compare all target elements, connections, and constraints to delete identical expressions.

3. Check the target model conforms to the target metamodel.

Completeness can be ensured in step (1) since all source elements are transformed, and a corresponding target element can be found for each source element. Step (2) is helpful and necessary to reduce target element redundancy, thus uniqueness is guaranteed. Syntactical correctness can be satisfied in step (3) since the target model will be expressed in a given formalism, and its semantic correctness will be evaluated in later stages of model transformation. Determinism is guaranteed implicitly in the model transformation editors, such as ATL IDE, which eases development and execution of ATL transformations [31].

In the M2T category, the transformation converts a source model into a text file, i.e. PSM to source code. If the text is in source code form, then the transformation is also called code generation, and the transformer is also called a code generator. The process of a M2T transformation is similar to that of a M2M transformation. The only difference is that step (ii) in the M2M transformation can be skipped in a M2T transformation, since uniqueness has already been checked. Therefore, the three criteria listed above are achieved according steps (i) and (iii). Similarly, determinism is satisfied because model transformation editors, such as Acceleo [32], implicitly guarantee a unique output for each particular input.

5 Conclusions

Abstraction is now widely admitted as an effective means to reduce the complexity of system specification. It is also generally agreed that ontology as an important form of abstraction can be employed in MDE to describe the existing world, the environment, and the domain of system. However, this consensus has not lead to a coherent research on how to enhance the semantic composability of simulation models yet. Hence, this study attempts to adopt an ontological metamodeling method for engineering the semantic composability of simulation models within MDE. We believe that the experience collected from this study can bring some new visions of simulation models development and of the state of art that relate to the semantic composability. A benefit of this study is that the formal definition of model transformation can be viewed as a referenced experience to guide other practices that have the needs of formal analysis. However, as a drawback some effort for further evaluations are required.

References

1. Bryant, B.R., Gray, J., Mernik, M., Clarke, P.J., France, R.B., Karsal, G.: Challenges and directions in formalizing the semantics of modeling languages. ComSIS **8**(2), 225–253 (2011). Special Issue
2. Abdulah, M.S.: A UML profile for conceptual modeling of knowledge-based systems. Ph.D. thesis. University of York, York, England (2006)
3. Langer, P., Wieland, K., Wimmer, M., Cabot, J.: From UML profiles to EMF profiles and beyond. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 52–67. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21952-8_6

4. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., et al.: The generic modeling environment. In: IEEE International Workshop on Intelligent Signal Processing, Budapest, Hungary, pp. 1–6 (2001)
5. Getir, S., Challenger, M., Kardas, G.: The formal semantics of a domain-specific modeling language for semantic web enabled multi-agent systems. Int. J. Coop. Inf. Syst. **23**(3), 1–53 (2014)
6. Meyers, B.: A multi-paradigm modeling approach to design and evolution of domain-specific modeling languages. Ph.D. thesis. University of Antwerpen, Antwerpen, Belgium (2016)
7. Schmidt, D.C.: Guest editor's introduction: model-driven engineering. IEEE Comput. **39**(2), 25–31 (2006)
8. Sarjoughian, H.S.: Model composability. In: 38th WSC Proceedings, Monterey, CA, pp. 149–158 (2006)
9. Hardebolle, C., Boulanger, F.: Exploring multi-paradigm modeling techniques. Simulation **85**(11–12), 688–708 (2009)
10. Mosterman, P.J., Vangheluwe, H.: Computer automated multi-paradigm modeling: an introduction. Simulation **80**(9), 433–450 (2004)
11. Lei, Y.L., Li, Q., Yang, F., Wang, W.P., Zhu, Y.F.: A composable modeling framework for weapon systems effectiveness simulation. Syst. Eng.-Theory Pract. **33**(11), 2954–2966 (2013)
12. Çetinkaya, D.: Model driven development of simulation models: defining and transforming conceptual models into simulation models by using metamodels and model transformations. M.S. thesis. Middle East Technical University, geboren te Konya, Turkije (2013)
13. Hu, X., Zeigler, B.P.: Model continuity in the design of dynamic distributed real-time systems. IEEE Trans. Syst. Man Cybern. - Part A: Syst. Hum. **35**(6), 867–878 (2005)
14. Balci, O.: A life cycle for modeling and simulation. Simulation **8**(7), 870–883 (2012)
15. Zhu, Z., Lei, Y.L., Zhu, N., Zhu, Y.F.: Composable modeling frameworks for networked air & missile defense systems. J. Natl. Univ. Defense Technol. **36**(5), 186–190 (2014)
16. Strembeck, M., Zdun, U.: An approach for the systematic development of domain-specific languages. Softw. Pract. Exper. **39**(15), 1253–1292 (2010)
17. Ehrig, H., Ermel, C.: Semantical correctness and completeness of model transformations using graph and rule transformation. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 194–210. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87405-8_14
18. Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electron. Notes Theoret. Comput. Sci. **152**(1–2), 125–142 (2005)
19. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture™: Practice and Promise. Addison-Wesley, Boston (2003)
20. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, 2nd edn. Academic Press, San Diego (2000)
21. Szabo, C., Teo, Y.M.: On syntactic composability and model reuse. In: 1st Asia International Proceedings on Modeling and Simulation, Phuket, Thailand, pp. 230–237 (2007)
22. Estanol, M., Sancho, M.R., Teniente, E.: Ensuring the semantic correctness of a BAUML artifact-centric BPM. Inf. Softw. Technol. **93**, 147–162 (2018)
23. Atkinson, C., Kuhne, T.: Model-driven development: a metamodeling foundtion. IEEE Softw. **20**(5), 36–41 (2003)
24. Nordstrom, G., Sztipanovits, J., Karsai, G., Ledeczi, A.: Metamodeling - rapid design and evolution of domain-specific modeling environments. In: IEEE Proceedings on Engineering of Computer-Based Systems, Nashville, TN, pp. 68–74 (1999)

25. Zhu, Z., Lei, Y.L., Zhu, Y.F., Sarjoughian, H.S.: Cognitive behaviors modeling using UML profile. IEEE Access **5**, 21694–21708 (2017)
26. De, L.J., Guerra, E., Cuadrado, J.S.: Model-driven engineering with domain-specific meta-modeling languages. Softw. Syst. Model **14**(1), 429–459 (2015)
27. Seo, K.M., Choi, C., Kim, T.G., Kim, J.H.: DEVS-based combat modeling for engagement-level simulation. Simulation **90**(7), 759–781 (2014)
28. Selic, B.: A systematic approach to domain-specific language design using UML. In: 10th IEEE International Proceedings on Object and Component-Oriented Real-Time Distributed Computing, Santorini Island, Greece, pp. 2–9 (2007)
29. Warmer, J., Kleppe, A.: The Object Constraint Language-Precise Modeling with UML. Addison-Wesley, Boston (1999)
30. Álvarez, J.M., Evans, A., Sammut, P.: Mapping between levels in the metamodel architecture. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 34–46. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45441-1_4
31. Jouault, F., Allilaire, F., Bezivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008)
32. Benouda, H., Essbai, R., Azizi, M., Moussaoui, M.: Modeling and code generation of Android applications using acceleo. Int. J. Softw. Eng. Appl. **10**(3), 83–94 (2013)