MODEL-DRIVEN SOFTWARE DEVELOPMENT AND DISCRETE EVENT SIMULATION

Thomas Sandu, Nicolas Knaak, Bernd Page

[1]University of Hamburg, Department of Informatics
Vogt-Kölln-Strasse 30, 22527 Hamburg, Germany

thomas.sandu@itemis.de (Thomas Sandu)

Abstract

Model-driven software development (MDSD) is a current direction in software engineering that stresses the importance of models in contrast to program code. Since models have always been of large importance in simulation, some aspects of MDSD are especially helpful to support discrete event simulation (DES) studies. In this paper we present a case study concerning the development of an MDSD-compliant domain architecture for DES. This includes code generation facilities for the object oriented simulation framework DESMO-J based on a new UML profile for DES. The approach is supported by techniques and tools from MDSD such as the generator framework openArchitectureWare and the test frameworks JUnit and FIT. On this foundation, a larger teaching example from the domain of harbor logistics has successfully been implemented as a reference model. On the basis of the gained experiences we discuss general prospects and drawbacks of applying MDSD to the development of simulation models as well as interactive simulation tools. In particular, we identify MDSD as an intermediate level between code-centric simulation approaches and domain-specific graphical tools. On the one hand, MDSD can ease the testing and implementation of large simulation programs. On the other hand, it provides techniques and tools that aid the development and prototyping of graphical simulation systems.

Keywords: DES, MDSD, UML, process-oriented simulation, software engineering

Presenting Author's Biography

Dipl.-Inform. Thomas Sandu received a diploma degree in Computer Science from the University of Hamburg in 2007. The subject of his diploma thesis is the model-driven development of discrete event simulation programs. He currently works as a software developer for itemis GmbH, a German software development and IT consulting company, which is specialized on model-driven software development.

1 Introduction

The employment of model-driven software development (MDSD) leads to a change of prevailing software development practice. The application of MDSD promises improvements by stressing the importance of models in relation to pure program code. This leads to a higher abstraction level compared to code-centric approaches. The source code of the application is generated from models. In the ideal case it even does not have to be manually adapted or extended. In order to benefit from MDSD a domain-specific language (DSL) needs to be designed that makes it possible to use elements of a previously crafted metamodel for describing the needed application. In order to achieve a controllable complexity of graphical models and transformations, the assigned domain-specific modeling language must support only a precisely defined domain.

The objective of discrete event simulation (DES) is to illustrate the behavior of real systems in order to understand them and make forecasts about them. Ever since practitioners in this field have employed various kinds of models. Simulation applications are often implemented using graphical simulation environments, but also by employing general purpose programming languages and simulation frameworks.

In this paper, we present a generative infrastructure for the model-driven development of process-oriented discrete event simulation programs. The platform consists of our Java-based simulation framework DESMO-J, the test frameworks JUnit and FIT and some specific helper classes. The DSL is an extension of the Unified Modeling Language (UML 2) that is specialized by defining a simulation-specific profile. The DSL was designed taking in account modeling techniques presented in [1].

Based on these practical experiences, we discuss general prospects that the use of MDSD can provide for developing simulation programs and graphical simulation tools. In both cases the use of MDSD changes the way simulation software is constructed. In particular, we show how to adapt the simulation modeling cycle to fit MDSD. Using MDSD and an iterative approach to software development can save valuable time by speeding up the implementation phase. The role of conceptual models in this context is also discussed. Since MDSD has some similarities to graphical simulation tools, it can possibly combine the user-friendly modeling facilities of graphical simulation tools with the power and flexibility of general purpose programming languages and simulation frameworks. Another important aspect is the support for simulation software testing by means of MDSD techniques.

The paper is organized as follows: In Section 2 we introduce foundations of MDSD and the chosen tools. Section 3 reviews related work. Section 4 presents our domain architecture for discrete event simulation including the UML profile and a basic reference model. In Section 5 we extend the scope towards a general discussion of prospects and drawbacks concerning the use of MDSD in the DES domain. Section 6 concludes the paper and provides an outlook to our future work.

2 Model-Driven Software Development

This section provides a brief introduction to model-driven software development. Following the principles of MDSD, we describe the characteristics and main roles of a typical software development process in this field.

2.1 Principle of MDSD

The principles of MDSD can be described with the aid of Figure 1. The source code of every software that is bound to a particular domain can be partitioned into three parts. Following [2] these are:

- generic source code
- schematical, repetitive source code
- individual, application specific source code

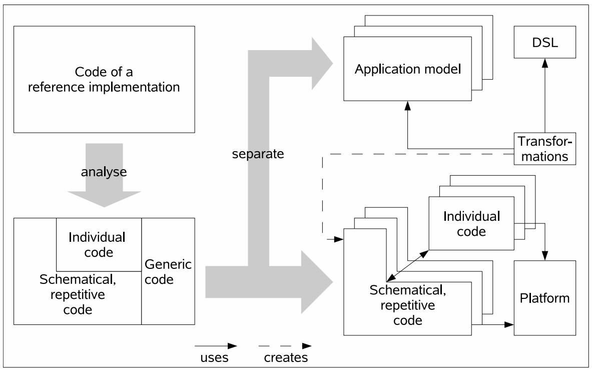

Fig. 1 Identification of repetitive code adopted from [2]

The generic code remains the same for all applications that belong to the particular domain. It becomes a part of the MDSD platform. The platform supports the generated parts of an application constructed in a model-driven fashion. The individual code is specific for every application of the domain. It has to be written manually and cannot be generated.

The objective of MDSD is to find a generative approach for the construction of the schematical, repetitive source code by employing models. This kind of code is not identical for all applications, but has a common structure or follows the same design patterns. In order to generate this part of the application code, an application model is built by means of a domain-specific language (DSL). Transformations designed for this DSL translate the elements of the application model to code that can be run on the MDSD platform. Being separated from the generated code, the generic and individual code is not overwritten in case of re-generation.

2.2 Characteristics of the Development Cycle

MDSD allows to separate the implementation of the business logic from the technical infrastructure. As a result, the development consists of two parallel phases illustrated in Figure 2. During the domain engineering phase the DSL, the transformations from model to code and the platform are built. They constitute the domain architecture and the technical infrastructure.

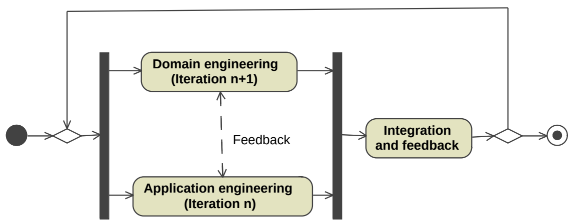

Fig. 2 Domain and application engineering with MDSD adopted from [3]

The application engineers use the DSL to model the needed functionality and business logic of the software product. They cannot merely rely on the generated artifacts but have to extend them by manually written code. Frequent feedback between the two phases leads to a continuous improvement of the domain architecture. Since the application engineering phase in the context of MDSD is not accomplishable without a first version of the DSL, it is necessary that the domain engineering phase starts one iteration in advance.

Due to the use of graphical models and code generation, MDSD heavily relies on tool support. In our case study described in the following, we chose the UML editor MagicDraw 11.6 to create the graphical models and the Eclipse platform as an extensible development environment. The Eclipse subproject UML2, an EMF-based implementation of the UML 2.x metamodel, was adopted as an implementation of the UML metamodel. The MDSD framework openArchitectureWare 4.1.2 (oAW) was used for creating the generator and for other MDSD specific tasks. Details about these tools and their application in our study are provided in [4].

3 Related Work

Traditionally there is a close link between object oriented modeling and the domain of (see also [1]). Current approaches are frequently based on UML and partly include code generation facilities. [5] e.g. present a UML tool that is able to generate simulation code for the process-oriented DES library JavaSim from class and sequence diagrams. Other simulation world views or diagram types are not supported, but the tool incorporates random variables and simulation statistics.

[6] apply modified UML 1.x activity diagrams to agent-based simulation modeling. They incorporate a large number of modeling elements like object nodes and send-/receive-signal actions and define their own extensions for timed states and “emergency-rules” anticipating some UML 2.0 elements. However, the extended notation can be handled and executed exclusively by their graphical simulation tool SeSAm. [7] employ class, statechart, collaboration and so called story diagrams to model and simulate production systems with their UML case tool Fujaba. However, none of the above approaches explicitly references MDSD processes and techniques.

A more ’MDSD-like’ approach is the work of [8] who use UML 2.0 component and statechart models for the performance analysis of network systems. The UML 2.0 compatible case-tool Tau Telelogic is used as an editor. There is a code generator based on the Velocity template engine that generates simulation programs for the process-oriented SimmCast framework.

[9] describes the advantages of combining the OMG standard *Model-Driven Architecture* (MDA) and DES. MDA can be seen as a specialization of MDSD with a strong focus on platform independence. This can be achieved by using platform independent models (PIM) that are transformed to platform specific models (PSM). The authors use the proprietary tool SIMplicity for modeling and transformation that can generate code for the High Level Architecture (HLA). Unlike the open source framework oAW, SIMplicity binds the user to a HLA-compliant platform (the predecessor Distributed Interactive Simulation (DIS) is also supported). The transformations cannot be manipulated by the user. A major advantage of our interpretation of MDSD is the possibility to alter the domain architecture at any time.

In [10] MDSD and simulation are used to predict the quality of service of models based on their architectural design. A DSL for modeling component-based architectures allows not only the specification of structural features but also of performance related information. The system supports the use of random variables, so uncertainty and nonpredictable behavior can be modeled. After building and parameterizing all required models, these are evaluated with a simulation program based on DESMO-J. In contrast to our work, the scope of this evaluation is to identify models with better quality of service and not to construct arbitrary simulation programs. In our approach, the constructed programs can be used for any simulation specific task depending on the constructed models and the manually implemented behavior.

4 Domain Architecture for Discrete Event Simulation

In the following, we present an example of a domain architecture for process-oriented DES. As a basis, we refine a typical simulation modeling cycle for the use with MDSD. We then present a DSL consisting of a new UML profile for DES and show how to use this for implementing a reference simulation model.

4.1 A Simulation Modeling Cycle Including MDSD

The MDSD development phases can be merged with a typical simulation modeling cycle such as that presented in [11]. If the domain architecture has not been implemented yet, the phases of domain engineering and application engineering are both necessary. In this case, the simulation-specific activities, like problem definition and data collection, become a part of the application engineering phase. If the domain architecture is already built and does not need improvement, the domain engineering phase can be omitted. The application engineering phase can then be incorporated in the simulation development process. In both cases the implementation phase of the simulation development cycle needs to be refined. A possible refinement is shown in figure 3.

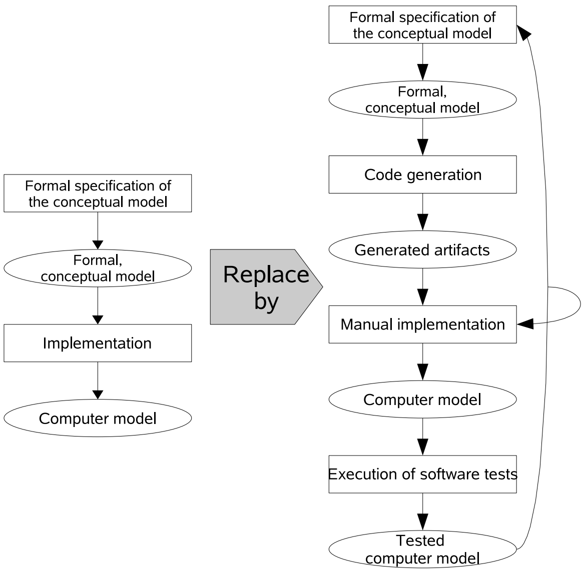

Fig. 3 Refinement of the simulation cycle from [11]

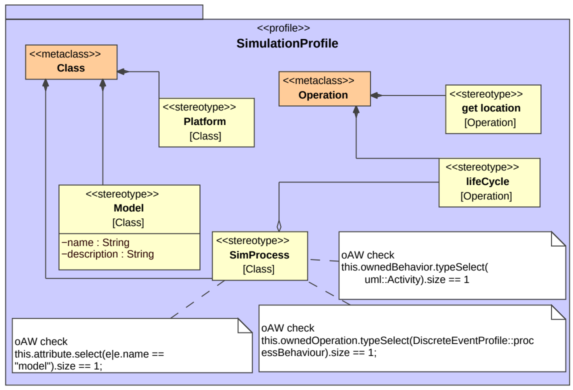

Fig. 4 First part of the DSL

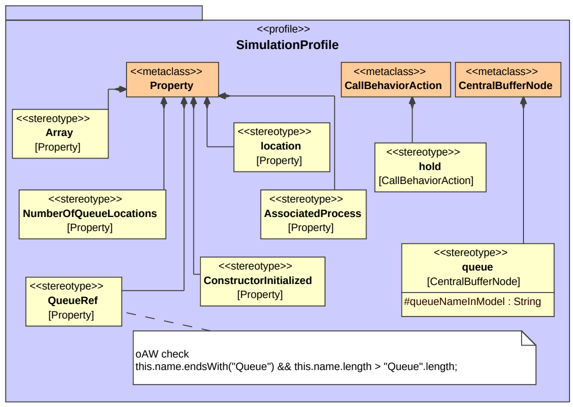

Fig. 5 Second part of the DSL

A more detailed discussion of the combination of the MDSD and simulation development cycles can be found in [4].

4.2 A DSL for Process-Oriented Simulation

In [4] we have developed a domain architecture for process-oriented discrete event simulation. The elements of the DSL are displayed in Figures 4 and 5. The DSL was created by defining a UML profile that partly implements the simulation-specific extension stereotypes proposed in [1].

The DSL supports two UML diagram types that we deem most important for DES, i.e. class and activity diagrams. In a class diagram the classes representing the simulation model and the processes can be marked with the stereotypes `<<Model>>` and `<<SimProcess>>`. The `<<Platform>>` stereotype indicates that a class is part of the platform or manually implemented, so nothing is generated from it. Operations can also be marked by stereotypes. `<<lifeCycle>>` indicates that an operation describes the behavior of a simulation process. Stereotypes of attributes are shown in figure 5. The stereotype `<<location>>`, for instance, marks an attribute that describes the location of a simulation entity within the model's environment.

Activity diagrams are employed to describe the lifecycle of simulation processes. Elements of activity diagrams such as object nodes or send and receive signal actions have been specialized with stereotypes according to the terminology of process-oriented simulation. The stereotype `<<hold>>` e.g. marks an action that passivates a simulation process for a certain period of time. An object node with the stereotype `<<queue>>` represents a waiting queue.

To validate the well-formedness of models created with the DSL, the UML profile includes constraints that have to resolve to true, before the code is generated. One example constraint shown in Figure 4 ensures that every simulation process class is connected to the simulation model class. Another constraint checks that every class with the `<<SimProcess>>` stereotype has only one operation marked with `<<lifeCycle>>`.

4.3 A Simulation Study Implemented with MDSD

As a reference model, we have chosen a teaching example from [11] that was previously implemented in the process- and event-oriented modeling styles. The model was re-implemented by means of MDSD (see [4]) to illustrate the possibilities of the constructed domain architecture. According to the principles of MDSD described in [2], we chose a straightforward, yet typical example and implemented the reference model in parallel to the domain architecture in order to ensure the appropriateness of the architecture and the quality of the generated code.

In [11] the model is introduced as follows: Hamburg is Germany's principal seaport and largest overseas trade and transshipment center. In this international seaport, container bridges charge a multitude of so-called feeder ships with containers for overseas transport, e.g. to further ports within the Baltic Sea. The feeder ships successively supply several Baltic Sea ports on different routes. In each of the visited ports, container bridges unload the containers destined to the respective port from the feeder ships. The objective of the simulation study is to gather information about bottlenecks concerning the container bridges as well as about the ships' lay days in each port.

In Figure 6 the classes of the simulation model (`ContainerShipmentModel`) and the processes (`Ship` and `Crane`) are defined. From this class diagram, not only simulation classes are generated. The code generation also comprises helper classes for the realization of the simulation processes' behavior as well as test classes.

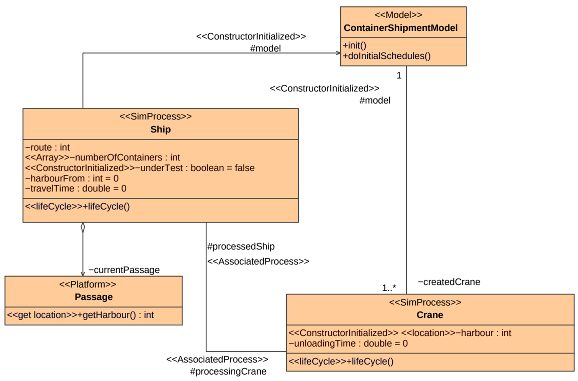

Fig. 6 Definition of the model class and the simulation processes

Figure 7 illustrates the definition of the behavior of a feeder ship process as an activity diagram. Elements from the DSL are the `Navigate to next port` action and the object nodes `Ships` and `Cranes`. The object nodes are marked by `<<queue>>` and have a tagged value named `queueNameInModel`. Tagged values are attributes defined by stereotypes and provide extra information for the correct generation of the code. The value of `queueNameInModel` represents the attribute name of the accessed queue as defined in the model class.

From every action in the activity diagram, an abstract action class is generated. It has to be subclassed to define the actual behavior of the respective action. To realize the complete behavior of the simulation process, instances of the subclasses are executed in the order imposed by the activity diagram. JUnit test classes are also generated for every action class. They use mock objects of their assigned simulation process and most of them need to be subclassed for concrete implementations.

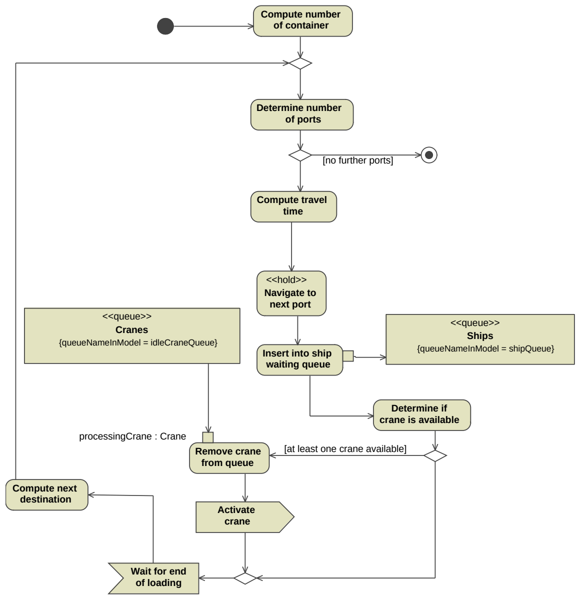

Fig. 7 Lifecycle of a feeder ship process

After the code generation the application engineers are informed which classes need to be subclassed and what names these subclasses must be given. This information is also generated and depends on the constructed UML models. In the oAW framework, these hints are contained in a so-called recipe file that can be interpreted with the aid of a specific Eclipse plug-in.

5 Combination of MDSD and DES

Building DES programs in an MDSD style provides advantages but also drawbacks. In the following, we discuss the combination of both fields, based on the experiences from constructing the above domain architecture. In this discussion, we particularly identify potential benefits for the construction of graphical simulation tools. A complementary discussion with a slightly different focus in the context of MDA can be found in [9].

5.1 Code Generation and Prototyping Save Time

MDSD makes it possible to generate all entities and other important elements of a simulation program from conceptual models. Once the initial effort of creating the code generation infrastructure is completed, the designer of a simulation application can concentrate on creating a good representation of the real system. Changes in the conceptual models are synchronized to changes in the source code. Only custom behavior has to be implemented manually while common actions such as adding a new server to an existing queuing model are performed automatically.

An argument against MDSD is the large effort in the early stages of the simulation study. Before being able to generate vital parts of the simulation, the DSL and the transformations need to be constructed. In DES understanding and studying the domain is traditionally time-consuming and complex. However, the deep understanding of the real system needed by a simulation developer can help him create a good metamodel and DSL. If a domain architecture can be re-used, the simulation developer can fully concentrate on the system under study. Since many application parts are generated, the implementation phase becomes shorter.

Following [2], an MDSD project needs a manually created reference implementation of important aspects of the domain, i.e. one or two manually implemented simple use cases that should cover all elements of the DSL. The transformations can be derived from this reference implementation and the code generation is based on the manually crafted code. In terms of quality this code should be superior to the code generated by former CASE tools. In simulation the reference implementation consists of a model representation lacking details compared to a productive simulation model. The need of a reference implementation encourages the designer to construct early prototypes, which supports an early elimination of misunderstandings regarding the real system. Summarizing, a reference implementation and an iterative approach to MDSD lead to an improvement of simulation software quality and can save valuable time.

5.2 Larger Importance of Conceptual Models

In code-centric simulation modeling, the formal MDSD models replace the traditional conceptual models. These models are of greater importance than their predecessors, since they do not only illustrate the structure and behavior of the real system, but are directly linked to the simulation program. Without using MDSD the conceptual models and the source code need to be synchronized manually. Changes of the MDSD models result in a generative update of the simulation program. After regenerating the application code, the application developers implement the parts of the program which need to be implemented manually. This workflow guarantees that the MDSD models always represent the latest version of the source code and are not only employed for documentation or for the first steps of the implementation.

5.3 Construction of Graphical Simulation Tools

The employed domain-specific language has to cover the concepts and entities of the analyzed domain. This can be done on a textual but also on a graphical basis. Choosing a graphical DSL has some well-known advantages such as an easier understandability and validation by domain experts and a higher level of abstraction. However, models built with a DSL become more complex than mere conceptual models since the mapping from the elements of the DSL to code has to be unambiguous.

MDSD has many strengths when combined with powerful object oriented frameworks. The code generated from the models constructed with the DSL does not directly implement the behavior of simulation elements, but instantiates the predefined elements from the used frameworks and takes care of the relations between these elements and their parameterization. A DSL covering all aspects of the analyzed domain and the assigned transformations can be regarded as a basic graphical simulation tool for one particular domain. A significant advantage over traditional simulation tools is the result of the code generation: It is an object oriented application, that can be modified and extended manually.

An obvious disadvantage of MDSD is the fact, that it often leads towards hard to use graphical languages supported by rather general tools like UML editors (as opposed to graphical simulation tools such as e.g. Extend [12]). More user friendly ways to design the executable simulation model and to set parameters of its components have to be found. The GMF plug-in for the Eclipse platform simplifies the creation of a special purpose graphical editor for a DSL. The tool allows to generate user friendly graph editors from the data structures describing the DSL metamodel. Thus the rapid prototyping of graphical simulation tools is supported. The resulting editors are Java applications and can be extended manually in order to reach the usability level of graphical simulation tools. Thereby simulation programmers can easily build specialized tools for domain experts without a programming background.

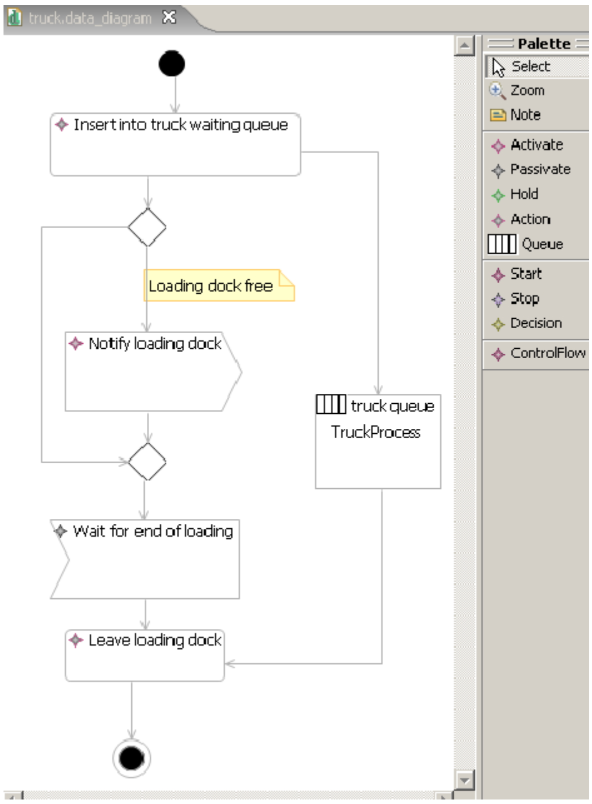

Fig. 8 A simple GMF editor for modeling the behavior of simulation processes

Figure 8 shows an editor generated with GMF. The input data for the generation is a meta-model for describing the behavior of simulation processes. In general GMF supports any kind of meta-model and could therefore be used to build graphical editors for other simulation world-views as well (e.g. transaction-oriented modeling). Without prior knowledge of GMF we managed to generate this editor within three man-days of work. The representation of the diagram is Eclipse EMF which is also supported by the employed oAW framework.

An MDSD domain architecture for discrete event simulation can furthermore be used as a basis for a traditional domain-specific graphical simulation tool. MDSD frameworks like oAW are very flexible and powerful and allow to easily implement importers for any kind of model representation. The generator can create components used by the simulation tool during its execution. Thereby, manufacturers of simulation software can profit from the *best practices* of MDSD and a change of platforms or modeling styles might become easier.

5.4 Assistance for Model Testing

Following [13] model testing is a challenging task in simulation. MDSD can ease the creation of software tests significantly because the generative approach also allows to generate test code. The generation of partially implemented test classes already eases the development of tests. Generated parts of the test classes show the inexperienced user which part of the application should be tested, and what test strategies should be used. Furthermore, the use of MDSD allows to specify the elements to be tested in the conceptual models. The semantics of this marking depends on the design of the transformations. For example certain data collectors can be marked to write debug reports or to be automatically compared to real system data in operational validation.

Besides unit tests, it is also possible to generate parts of integration or acceptance tests. These black box tests ensure that the overall application exhibits the expected behavior. Therefore, they are quite appropriate to test the behavior of complete simulation models. We have made rather positive experiences with the FIT Framework for Integration Tests by Ward Cunningham [14] that are reported in detail in [4].

MDSD also eases the subsequent refactoring of existing applications towards better testability. This is due to the fact, that the architecture of the constructed application is encapsulated in the transformations. Therefore, it can be refined more easily than in conventional applications, since it is only necessary to adapt the transformations. A single change of the domain architecture affects many generated artifacts. If the focus is on better testability, the improvement of the architecture towards a better testable structure can thereby be simplified throughout the whole application.

6 Conclusions

In this paper, we have discussed the benefits and drawbacks of applying model driven software development in the domain of discrete event simulation. We have implemented a domain architecture for DES and an operational generative infrastructure. It comprises a new UML profile for process-oriented simulation as the domain-specific DSL and code generation facilities for the object oriented simulation framework DESMO-J. In this context, we have applied and evaluated several MDSD-specific tools and technologies. We have also successfully implemented an example from harbor logistics as a reference model.

Based on these experiences, we have drawn conclusions on the general applicability of MDSD to DES. As a benefit, the generative approach of MDSD can help saving time during model development and further encourage early prototyping in simulation. Additionally, MDSD stresses the importance of models in code-centric simulation approaches and provides support for model testing.

However, due to its rather technical orientation, MDSD cannot replace traditional domain-specific graphical simulation tools. Instead, it provides an intermediate level between code-centric and graphical model development. On the one hand, MDSD supports the developer of large simulation programs in schematical routine tasks on the basis of models. On the other hand, MDSD-related concepts and tools like GMF or oAW can ease the rapid prototyping of graphical simulation tools for domain experts.

In future work, the presented concepts should be applied in other and larger simulation studies, and the presented DSL should be adapted accordingly. More domain-specific editors for DESMO-J models can be built based on the GMF framework. Another interesting direction for future research is an investigation of the applicability of MDSD to later phases of a simulation study such as experimentation, result analysis, and validation.

7 References

[1] Bernd Page and Nicolas Knaak. Applications and Extensions of the Unified Modelling Language UML 2 for Discrete Event Simulation. In Y. Merkuryev, R. Zobel, and E. Kerckhoffs, editors, *Simulation in Wider Europe - Proceedings of the 19th European Conference on Modelling and Simulation (ECMS 2005)*, Riga (Latvia), 2005.

[2] Thomas Stahl and Markus Völter. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. dpunkt.verlag, Heidelberg, 2005.

[3] Jorn Bettin. Prozessauswirkungen von MDSD. http://www.sigs.de/publications/os/2004/MDD/bettin_MDD_2004.pdf (in April 2007), 2004.

[4] Thomas Sandu. Modellgetriebene Entwicklung von Simulationsprogrammen am Beispiel des DESMO-J-Frameworks. Master's thesis, University of Hamburg, 2007.

[5] L.B. Arief and N.A. Speirs. A UML Tool for an Automatic Generation of Simulation Programs. In *WOSP 2000*, Ontario, Canada, 2000.

[6] Christoph Oechslein, Franziska Klügl, Rainer Herrler, and Frank Puppe. UML for Behaviour-

Oriented Multi-Agent Simulations. In B. Dunin-Keplicz and E. Nawarecki, editors, *Proceedings of the CEEMAS*, number 2296 in Lecture Notes in Artificial Intelligence, pages 217 – 226, Berlin, 2001. Springer.

[7] H.J. Köhler, U. Nickel, Jörg Niere, and Albert Zündorf. Integrating UML Diagrams for Production Control Systems. In *Proc. of the 22nd International Conference on Software Engineering (ICSE)*, pages pp. 241–251, Limerick, Ireland, 2000. ACM Press.

[8] Nico De Wet and Pieter Kritzinger. Using UML Models for the Performance Analysis of Network Systems. In *Proceedings of the Workshop on Integrated-reliability with Telecommunications and UML Languages (WITUL)*, Rennes, Brittany, France, 2004.

[9] Shawn Parr and Russell Keith-Magee. How To Apply MDA To Simulation. In *SimTecT 2004 Simulation Conference*, 2004.

[10] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-Based Performance Prediction with the Palladio Component Model. In *Workshop on Software and Performance (WOSP 2007)*, 2007.

[11] Bernd Page and Wolfgang Kreutzer. *The Java Simulation Handbook. Simulationg Discrete Event Systems with UML and Java*. Shaker Verlag, Aachen, 2005.

[12] Imagine That Inc. Extend. http://www.imaginethatinc.com (in April 2007), 2007.

[13] C. M. Overstreet. Model Testing: Is it only a Special Case of Software Testing. In E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, editors, *Proceedings of the 2002 Winter Simulation Conference*, pages 641-647, 2002.

[14] Ward Cunningham. Framework for Integrated Test. http://fit.c2.com/wiki.cgi (in April 2007), January 2007.

