Prerequisites for Simulation-Based Software Design and Deployment

Radek Kočí and Vladimír Janoušek

Brno University of Technology, Faculty of Information Technology
Bozetechova 2, 612 66 Brno, Czech Republic
emails: koci@fit.vut.cz, janousek@fit.vut.cz

***Abstract*—The fundamental problem associated with software development is correctly identifying, specifying, and realizing the software system requirements. Many methodologies are not formally defined and rely on intuitive use. In contrast, the formal description techniques clearly describe the user requirements and their specific solutions. We are involved in modeling the requirements and behavior of software systems using formal models used in a specific manner. The approach combines intuitive modeling with the precise expression of specified requirements and a detailed implementation description. Models serve for analysis, system design, validation, and simulation. Models can also be directly deployed in real environments of developed systems. This paper summarizes the current state of the approach to system development, which is being developed by our team.**

***Keywords*—Modeling; simulation-based design; model-driven engineering; model-continuity.**

I. Introduction

Software Engineering deals with the issues of efficient development of correct and reliable systems. Correctness means that the system fits perfectly with the intentions and goals of deploying this system. Reliability means the system does not contain errors or provide for damage caused by unexpected and wrong behavior. The primary development cycle of each software product is divided into several phases that are continuously linked to each other. The first phases are mainly analysis and specification of requirements, system design, implementation and testing, and finally, system deployment. Many software development methodologies work with phases in different ways. It is possible to follow the phases one by one accurately, to overlap or iterate them. In any case, they are part of every development process. One of the fundamental problems is the correct specification and validation of the requirements for the system [1]. A use case diagram from the Unified Modeling Language (UML) is often used to specify the requirements, which is then developed with other UML diagrams [2]. The disadvantage of this approach is the difficulty in validating the specification models. In response, methods for working with modified UML models having executable form have been developed, such as the Model Driven Architecture (MDA) methodology [3], the Executable UML language (xUML) [4] or the Foundational Subset for xUML [5]. However, these approaches still need to solve the problem of model transformations as it is difficult to transfer back to the model all the changes that result from the validation process. Another approach, for example, uses a modified subset of the UML, called fUML, with the formal language Alf [6][7]. This approach is supported by modeling and analysis tools [8].

The fundamental prerequisite for achieving the correct and reliable system is continuous verification or validation of specification documents, design documents, and implementation [9]. Another area for improvement is the transition between different development process phases, from one document type to another. An example may be the transition from an informal specification to the model or from a design model to the implementation. In these cases, mistakes often occur due to misinterpretation of the outgoing model or by simply overlooking any model element. Two main reasons for these mistakes are the complexity and informal semantics. Many elements of the used modeling means need a clearly defined syntax and semantics, and their use is relatively intuitive. In this paper, we summarize the concepts of software product development and deployment using a combination of formal and informal models, programming languages, and simulation.

The paper is structured as follows. We discuss related work in Section II. In Section III, we specify basic requirements for reliable software development and deployment. Section IV introduces models that may appear during the development life cycle. Section V addresses techniques needed for exploiting the potential of visual and formal languages in the simulation-based design.

II. Related Word

The approach that combines formal models, simulations, and their deployment or transformation is mainly applied in control software. Many of these approaches [10]–[12] propose to generate models in a particular language (e.g., System Modeling Language—SysML) from UML models, usually from a class diagram. Other work, such as [13], transforms different levels of diagrams. Some approaches attempt to transform conceptual models, described, i.e., in SysML, into simulation models [14]. The approach most closely resembles ours is based on the network-within-networks (NwN) formalism, with which the Renew tool is associated [15]. In the design of more general software systems, an example is already mentioned xUML or fUML. The resulting system can often be generated from the designed models [16][17]. However, freely available tools only allow partial output (often, only a skeleton in the chosen language is generated). However, these approaches also do not allow formal models to implement the system but only for simulation runs. Our proposed approach retains the generated models throughout the software development and deployment. We aim to create more efficient representations of models and their simulators for deployment purposes on commonly used platforms and languages (Java, C++).

III. Prerequisites for a Simulation-Based Approach

This section briefly summarizes the basic requirements for reliable software design and defines the prerequisites for meeting these requirements.

A. Basic requirements

First, we define points that have to be met to create the correct and reliable software system.

1) understand the goals of the software project and precisely specify the specific requirements whose implementation meets the declared objectives,
2) verify that the requirements specification is in line with the objectives,
3) based on a verified specification, create a system design that reflects the conditions of a particular implementation environment,
4) verify that the system design complies with the requirements,
5) implement the verified design,
6) verify that implementation is consistent with the design,
7) verify accuracy and reliability of implementation under real conditions

In the following sections, we explain the basic principles of our approach and how they fulfill the above points.

B. Model Continuity

The primary means for specifying requirements is plain text in the native language. In this form, the specification is also part of the contract between the developer and the customer. Validation of the text description of requirements specification is, however, difficult and very often impossible. This validation can not be performed in an automated manner but by a person. Nevertheless, someone is limited by his/her memory and cannot analyze multi-page text in all its dependencies.

Visual models with a clear formal foundation make it possible to capture a particular aspect of requirements unequivocally, helping to understand the developed system better and detect errors. For these reasons, the ideal state is to use one type of model that captures everything. However, such a model would be too extensive to get into the same problem as the text description.

A more appropriate approach is to combine models capturing the system at different levels of abstraction so that it is possible to view and analyze system models as a whole and their details. This approach is complemented by other models or text descriptions that include those features or requirements that can not be captured in the existing models; eventually, their capture would be very complicated. *It satisfies the point 1 from the list in Section III-A.*

An important feature that extends the capability of validating specification models is the ability to simulate these models. It allows live testing of models in simulated conditions instead of simply passing through a document. Another aspect that affects validation capabilities is the environment or context in which the simulation is performed. If models are integrated into a realistic environment, the credibility of simulation results increases. *It satisfies the point 2 from the list in Section III-A.*

After the validation of specification models, the question of a correct transition to the design models and the subsequent implementation in the chosen environment remains. We aim to work with the same models in all development phases, especially the specification, design, and implementation, with no transformation and minimization of errors. The models are only complemented with further details while preserving the possibility of previewing models at different levels of abstraction from the specification to the implementation view. *It satisfies points 3 and 4 from the list in Section III-A.*

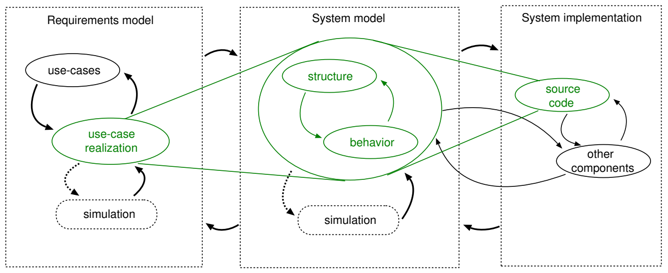

Figure 1. Model Continuity: Basic principle.

At the end of the development process, we have functional models that fully reflect the system requirements. In certain situations, especially concerning performance, these models can serve as implementation models, i.e., become part of the target system. If this is inappropriate or impossible for the above reasons, we must implement or exploit the ability to generate code from such models. Consistency with the design does not need to be checked, as the same set of models is still being developed. Verification accuracy and reliability under real conditions are proved in the same or partially modified manner. *It satisfies points 5, 6, and 7 from the list in Section III-A.*

The prior text presents the basic principle of the continuity model, which is depicted in Figure 1. Design models complement and extend each other in the development process, and there is no need to transform or create new models based on existing ones. If the nature of the resulting application permits, it is possible to maintain the models in the target system.

IV. Models in the Simulation-Based Development Life-Cycle

In this section, we will introduce the types of models that may appear during the life cycle of simulation development of software systems. One of the basic principles of simulation development is the continuity of models over the entire development process until deployment in the application environment. We define categories of models and typical representatives and evaluate their applicability in simulation development.

The process of modeling software systems consists of several phases that follow and interleave [18]. Various modeling tools may be used in each phase, but there must be a way of interconnecting them. We distinguish between *Domain Model*, *Behavioral Model*, and *Design Model* that are supplemented by *Architecture Model*.

In the following text, we will explain and analyze the importance of each model in more detail. We will proceed from a simple example of a robotic system, which we now briefly specify. The example, which is based on the case study presented in [19], presents a robot control system whose motion is controlled by a pre-specified algorithm.

A. Domain Model

Domain Model captures concepts of the domain system as identified and understood during the requirements analysis process. The class diagram modeling conceptual classes and their links is usually used. The domain model is the initial model for modeling functional requirements and creation of design models and is one of the first models to use when designing the software.

B. Behavioral Model

Behavior Model captures an external view of the system's functionality, specific behavior, and system interaction with its surroundings. The behavior model can be divided into two complementary types:

- *User Requirements Model* captures an external view of the system functionality. Use case models are used.
- *Scenario Model (Model of Functional Requirements)* captures specific behavior and interaction of individual use cases. Different descriptions are used, e.g., structured text, activity diagrams, or state charts. Generally speaking, it is possible to use such models that describe the workflow of the use case supplemented by communication mechanisms.

Use case diagrams are used to model user requirements. The goal of modeling is to identify system users, user requirements, and how the user works with these requirements. The essential elements are *users* of the system, their *role*, and *activities*. Roles are modeled through *actors* and activity through individual *use cases* in the use case diagram. Interconnected scenarios (activity nets or role nets) then specify the behavior of the individual elements (see Figure 2) and can be described by different formalisms.

Activity diagrams, state diagrams, or interaction diagrams can model case scenarios (activities). However, formal models or formal languages, such as *Petri nets*, can also be used with advantage. An important feature is an interconnection between use case diagrams and scenarios modeled by specific diagrams since both models represent a different view of the system under development.

Figure 3 shows an example of the scenario model for the elements Robot (role) and Execute Scenario (activity). The formalism of Petri nets, its variant Object-Oriented Petri nets, models the scenarios and uses its inherent synchronous port mechanism (e.g., isCloseToObstacle) to synchronize with each other [20].

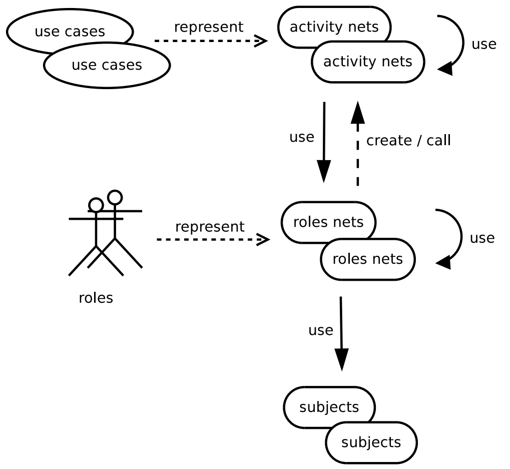

Figure 2. Interrelation of elements and their descriptions.

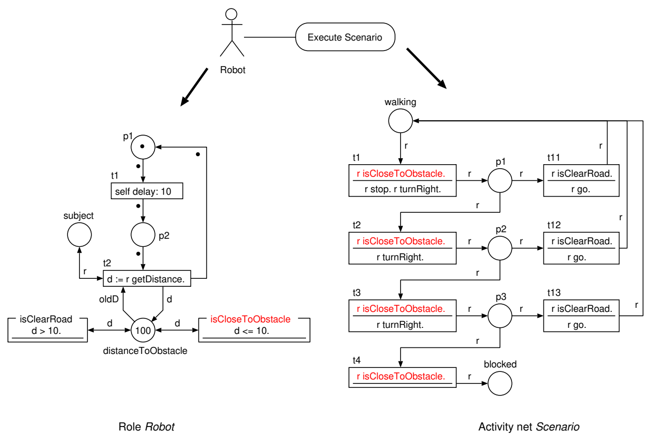

Figure 3. Sample scenario model for role and use case.

C. Design and Architecture Model

Design Model is based on the domain and behavioral models. Generally speaking, these are elaborate models of the domain, requirements, and behavior that can be directly implemented. Class diagrams, activity diagrams, or state diagrams are used. The *Architecture Model* captures the organization of the design classes. Class diagrams and grouping diagrams or deployment diagrams are used. Usually, the architecture model merges with the design models.

D. Interrelation of Models

As indicated in Figure 4, the scenario models at the level of behavioral and design models merge into a single concept. Therefore, the class diagram is also included in this concept. The scenarios associated with the diagram of use cases correspond to classes from the design model. A specific class type models each element. Thus, we can identify groups of classes modeling *actors*, *use cases*, and other classes based on the domain model. As behavioral models evolve, they become design models that also serve the purpose of requirements specification. The basic view of requirements is conveyed by use case diagrams, class diagrams provide the architectural view, and individual scenarios are represented as workflows specified by Petri nets. Additional objects from the domain environment can be used in the workflow to simulate the system or run it under realistic conditions without displaying these implementation details at the scenario level. Thus, the same model can be used both for requirements documentation and for the developed system's executable version (prototype, implementation).

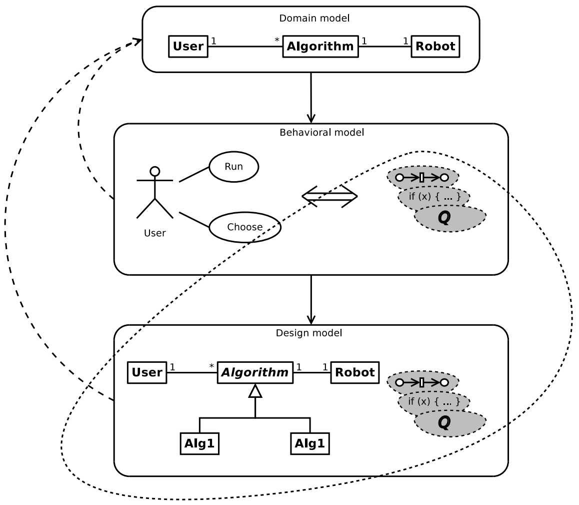

Figure 4. The role of models in the development process.

V. Supporting Techniques

The mentioned prerequisites need to be complemented by supporting techniques that exploit the potential of visual and formal languages in the simulation-based design. Many of these techniques have already been developed and introduced in previous papers.

A. Components

The possibility of exchanging parts of the software to debug and verify the correctness or behavior of the system under different conditions. The exchange should be enabled on the fly (simulation). For this purpose, a component concept based on the Discrete Event System Specification formalism (DEVS) [21][22] was chosen. It makes possible to associate the formal models described by High-level nets with an executable code that is incorporated into DEVS formalism structures. DEVS formalism is abstract concept that can be easily adapted to a particular environment.

B. Debugging and Constraints

An important aspect is, of course, the possibility of debugging and stepping. Simulation stepping is an obvious functionality of the simulation tool. We have also explored tracking and reverting the model run using Petri nets [23]. However, the presented concept still needs to consider all possible applications.

We also introduced the basic concept of requirements validation and its implementation through scenarios described by sequence diagrams [24]. Scenarios can be created manually or generated from running (simulation) models. It allows us to obtain assumed scenarios of the behavior of the use case under study and real scenarios reflecting the design that can be compared.

We introduced the concept of constraints and exceptions over the Petri net formalism, which can be used to verify the consistency of component interfaces or the correctness of the behavior of the modeled system [21].

C. Models Supported by Programming Languages

Models can be combined with programming (or other formal) languages that can be interpreted together with the model. Thus, they can also use concepts (e.g., objects) from another environment or programming languages. Current simulator can work with only Smalltalk objects.

D. Transformations

Transform the model into the chosen programming language for more efficient running. In the case of a transformed model, using some of the above resources is limited. Currently, we have the experimental implementation of transformations to Java and C++ languages done by our master students.

VI. Conclusion

This paper summarized the concept of simulation-based software development in conjunction with model-continuity principles and the current state of the art that our research team has achieved. The simulator has experimentally implemented many of the techniques presented but is only partially suitable for wider use (experimental implementation in a Smalltalk environment). We are, therefore, currently working on a new implementation of the simulator and a comprehensive model editor in Java. The goal is to create a comprehensive tool for modeling, designing, and verifying software systems with the possibility of direct deployment (with a lightweight version of the virtual machine for running models) or direct transformation into a programming language for more efficient running.

Acknowledgment

This work has been supported by the internal BUT project FIT-S-23-8151 – Reliable, Secure, and Intelligent Computer Systems.

References

[1] K. Wiegers and J. Beatty, Software Requirements. Microsoft Press, 2014.
[2] N. Daoust, Requirements Modeling for Bussiness Analysts. Technics Publications, LLC, 2012.
[3] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in International Conference on Software Engineering, FOSE, 2007, pp. 37–54.
[4] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie, Model Driven Architecture with Executable UML. Cambridge University Press, 2004.

[5] S. Mijatov, P. Langer, T. Mayerhofer, and G. Kappel, “A framework for testing uml activities based on fuml,” in Proc. of 10th Int. Workshop on Model Driven Engineering, Verification, and Validation, vol. 1069, 2013, pp. 11–20.

[6] T. Buchmann and A. Rimer, “Unifying modeling and programming with alf,” in SOFTENG 2016: The Second International Conference on Advances and Trends in Software Engineering, 2016, pp. 10–15.

[7] E. Seidewitz, “UML with meaning: executable modeling in foundational UML and the Alf action language,” in HILT ’14 Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, 2014, pp. 61–68.

[8] Z. Micskei and et al., “On open source tools for behavioral modeling and analysis with fuml and alf,” in 1st Workshop on Open Source Software for Model Driven Engineering, MODELS 2014, pp. 31–41, [online; retrieved: September, 2022]. [Online]. Available: http://ceur-ws.org/Vol-1290/paper3.pdf

[9] D. Cetinkaya, A. V. Dai, and M. D. Seck, “Model continuity in discrete event simulation: A framework for model-driven development of simulation models,” ACM Transactions on Modeling and Computer Simulation, vol. 25, no. 3, 2015, pp. 17:1–17:24.

[10] T. Hussain and G. Frey, “UML-based Development Process for IEC 61499 with Automatic Test-case Generation,” in IEEE Conference on Emerging Technologies and Factory Automation. IEEE, 2010.

[11] C. A. Garcia, E. X. Castellanos, C. Rosero, and Carlos, “Designing Automation Distributed Systems Based on IEC-61499 and UML,” in 5th International Conference in Software Engineering Research and Innovation (CONISOFT). IEEE, 2017.

[12] I. A. Batchkova, Y. A. Belev, and D. L. Tzakova, “IEC 61499 Based Control of Cyber-Physical Systems,” Industry 4.0, vol. 5, no. 1, November 2020, pp. 10–13.

[13] S. Panjaitan and G. Frey, “Functional Design for IEC 61499 Distributed Control Systems using UML Activity Diagrams,” in Proceedings of the 2005 International Conference on Instrumentation, Communications and Information Technology ICICI 2005, 2005, pp. 64–70.

[14] G. D. Kapos, V. Dalakas, A. Tsadimas, M. Nikolaidou, and D. Anagnostopoulos, “Model-based system engineering using SysML: Deriving executable simulation models with QVT,” in IEEE International Systems Conference Proceedings, 2014.

[15] L. Cabac, M. Haustermann, and D. Mosteller, “Renew 2.5 - towards a comprehensive integrated development environment for petri net-based applications,” in Application and Theory of Petri Nets and Concurrency - 37th International Conference, PETRI NETS 2016, Toruń, Poland, June 19-24, 2016. Proceedings, 2016, pp. 101–112. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-39086-4_7

[16] F. Ciccozzi, “On the automated translational execution of the action language for foundational uml,” Software and Systems Modeling, vol. 17, no. 4, 2018, doi: 10.1007/s10270-016-0556-7.

[17] E. Seidewitz and J. Tatibouet, “Tool paper: Combining alf and uml in modeling tools an example with papyrus,” in 15th Internation Workshop on OCL and Textual Modeling, MODELS 2015, pp. 105–119, [online; retrieved: September, 2022]. [Online]. Available: http://ceur-ws.org/Vol-1512/paper09.pdf

[18] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall, 2004.

[19] R. Kočí and V. Janoušek, “Formal Models in Software Development and Deployment: A Case Study,” International Journal on Advances in Software, vol. 7, no. 1, 2014, pp. 266–276.

[20] R. Kočí and V. Janoušek, “Specification of Requirements Using Unified Modeling Language and Petri Nets,” International Journal on Advances in Software, vol. 10, no. 12, 2017, pp. 121–131.

[21] R. Kočí and V. Janoušek, “The Object Oriented Petri Net Component Model,” in The Tenth International Conference on Software Engineering Advances. Xpert Publishing Services, 2015, pp. 309–315.

[22] R. Kočí and V. Janoušek, “Incorporating Petri Nets into DEVS Formalism for Precise System Modeling,” in ICSEA 2019, The Fourteenth International Conference on Software Engineering Advances. Xpert Publishing Services, 2019, pp. 184–189.

[23] R. Koci and V. Janousek, “Possibilities of the reverse run of software systems modeled by petri nets,” International Journal on Advances in Software, vol. 12, no. 3, 2019, pp. 191–200.

[24] R. Kočí, “Requirements validation through scenario generation and comparison,” in The Fifteenth International Conference on Software Engineering Advances, ICSEA 2020. Xpert Publishing Services, 2020, pp. 129–134.