Developing User and Recording Interfaces for Design Time and Runtime Models

Martin Gogolla, Nisha Desai, Khan-Hoang Doan

Computer Science Department, University of Bremen, Bremen, Germany
{gogolla|nisha|doankh}@uni-bremen.de

Abstract. Design time and runtime models may be uniformly described by UML and OCL models connected through correspondence models. We propose to develop a common user interface for both model layers in order to provide systematic and uniform access through operation calls to design time and runtime items. Operation calls can be systematically recorded in filmstrip models so that complete design time and runtime development steps are accessible. From recorded steps, various standard software models (like class, object and sequence diagrams) may be derived for documentation and comprehension purposes.

1 Introduction

Recently, design time and runtime models of software as well as their connection have attracted attention in research and development [3]. Utilizing models has many advantages over considering mainly code as first class software development artifacts, as models are able to abstract away from unneeded technical details. However, a common agreement about essentials of design time and runtime models and their relationship is still under discussion. This contribution discusses one way of defining and using such design time and runtime models.

This paper applies modelling languages as UML (Unified Modeling Language) [11, 12], which includes the OCL (Object Constraint Language) [10, 14]. UML and OCL support a wide range of systems. Our underlying assumption is that UML and OCL are suitable also for specifying design time and runtime models connected through a correspondence model [6]. In simple cases, the correspondence consists of associations between design time and runtime items. Roughly speaking, in comparison to the OMG 4 level MDA architecture, we only use two levels and map the OMG M2 level to an M1 level (details in Sect. 4).

Assuming a completely specified design time and runtime model has already been achieved, we propose here to introduce a user interface model and a model to record design time and runtime development steps within a so-called filmstrip model [4]. We then derive other standard software models from the recorded steps and from the recorded model. These models serve for documentation and comprehension. All examples have been implemented in our tool USE [5].

The rest of this contribution is structured as follows. Section 2 motivates our approach by studying an overview example. Technical details about the approach

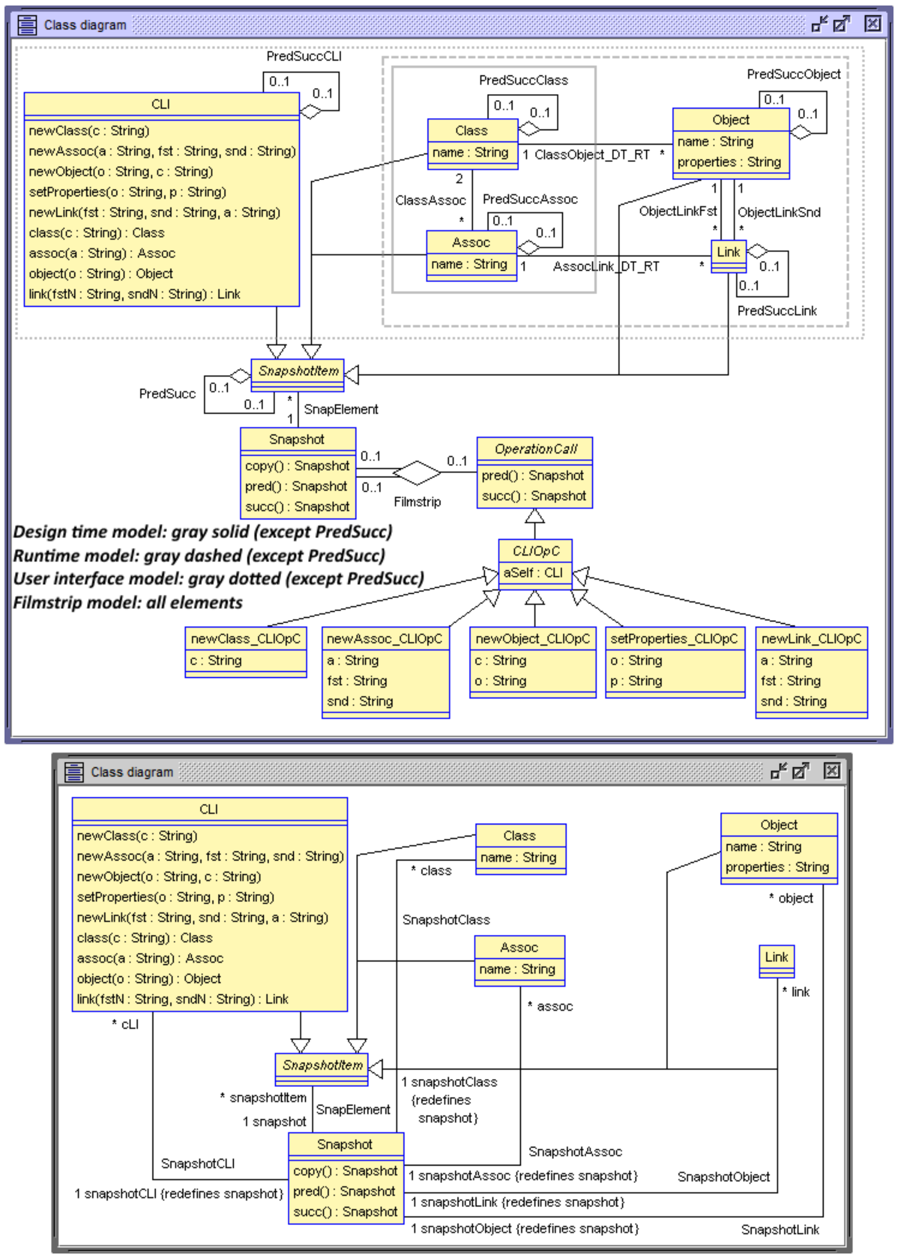


Fig. 1. Overview Class Models: Design time, Runtime, User interface, Filmstrip model.

are discussed in Sect. 3. Section 4 shows related work. Section 5 closes the paper with concluding remarks and future work.

2 Basic Idea: Design Time, Runtime, User Interface, Filmstripping, Derived Models

The class diagram in Fig. 1 determines the structures of our approach. It contains submodels for particular purposes: a design time model, a runtime model, a user interface model, and a filmstrip model for recording operation calls from the user interface model.

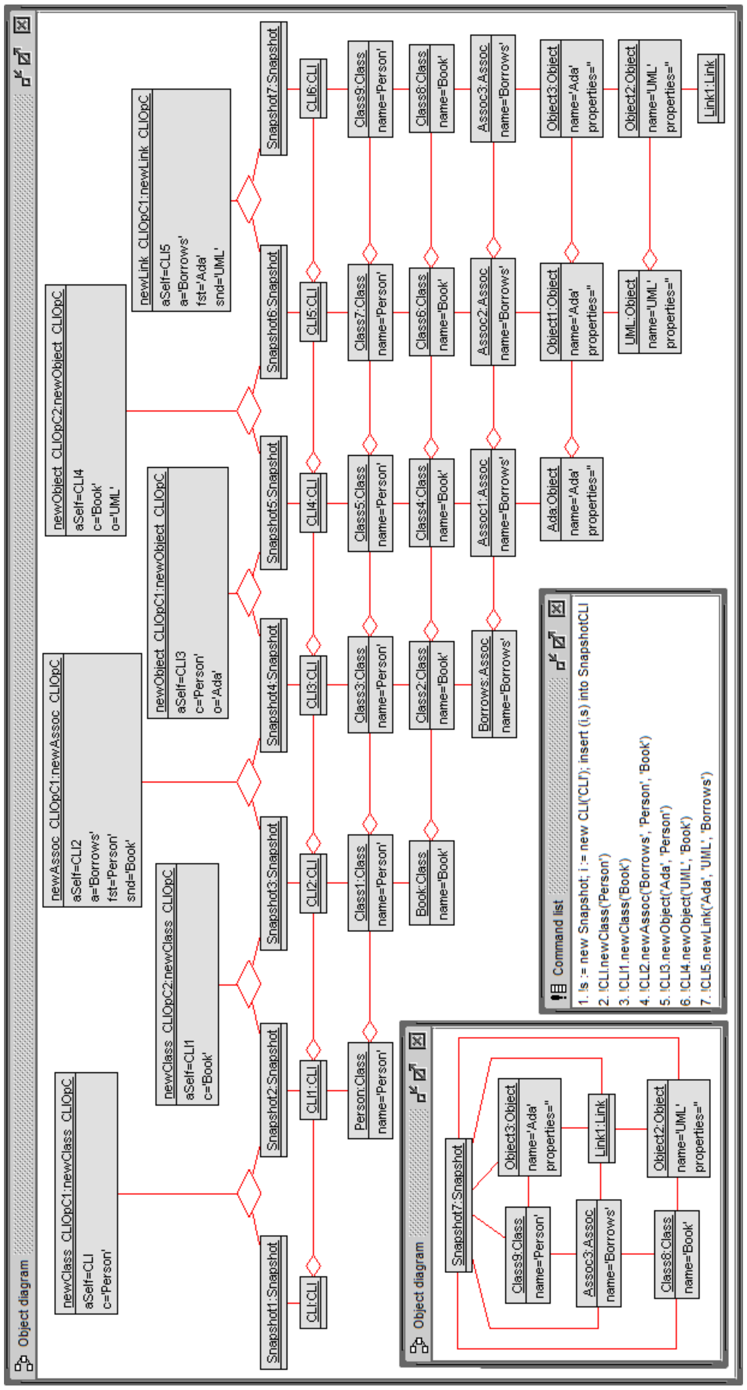
Design time model: In the upper center, the figure contains the classes Class and Assoc. Its purpose is to enable the specification of classes and binary associations, not general associations. This can be regarded as a dramatically simplified version of the UML metamodel for UML class diagrams (within the OMG 4 level architecture).

Runtime model: In the upper right, the figure shows the classes Object and Link. Its purpose is to make it possible to build objects and links for previously defines classes and associations. This can be seen as a simplified version of the UML metamodel for UML object diagrams (within the OMG 4 level architecture).
In our view, the design time items constitute typical elements needed during system design, and the runtime items are the building blocks occurring during system execution (or system runtime).

User interface model: In the upper left, the figure defines the class CLI (Command Line Interface). The operations of the class are designed for building up a (simple) class diagram and a (simple) object diagram. The operation parameters are all String-valued and depend on each other. For example, in the operation newAssoc the parameters fst and snd refer to the names of the classes between the new association that is going to be defined.

Filmstrip model: In the lower part, the figure defined a filmstrip model for recording operation calls from the command line interface. Every CLI operation becomes an OperationCall class so that operation calls are represented as OperationCall objects.

Figure 2 displays an example execution for the discussed class diagram in Fig. 1. In the middle bottom part, the USE commands and operation calls that lead to the main object diagram in the upper part are shown. After three initialization commands, two classes, one association, two objects and one link are constructed. In our view, the design time items are Person, Book, Borrows, and the runtime items are Ada, UML and the Borrows link between Ada and UML. The filmstrip model is responsible for recording the CLI operation calls and its effects in a chain of snapshots. Different Snapshot objects represent the system state at different points in time. The snapshots reflect that, in this case, classes, associations, objects and links are added. The last snapshot Snapshot7 is displayed in more detail in the lower left of the figure and shows the final state



```
1. !s := new Snapshot; i := new CLI('CLI'); insert (i,s) into SnapshotCLI
2. !CLI.newClass('Person')
3. !CLI1.newClass('Book')
4. !CLI2.newAssoc('Borrows', 'Person', 'Book')
5. !CLI3.newObject('Ada', 'Person')
6. !CLI4.newObject('UML', 'Book')
7. !CLI5.newLink('Ada', 'UML', 'Borrows')
```

Fig. 2. Example Filmstrip Object Model with Design-Runtime Model Development.

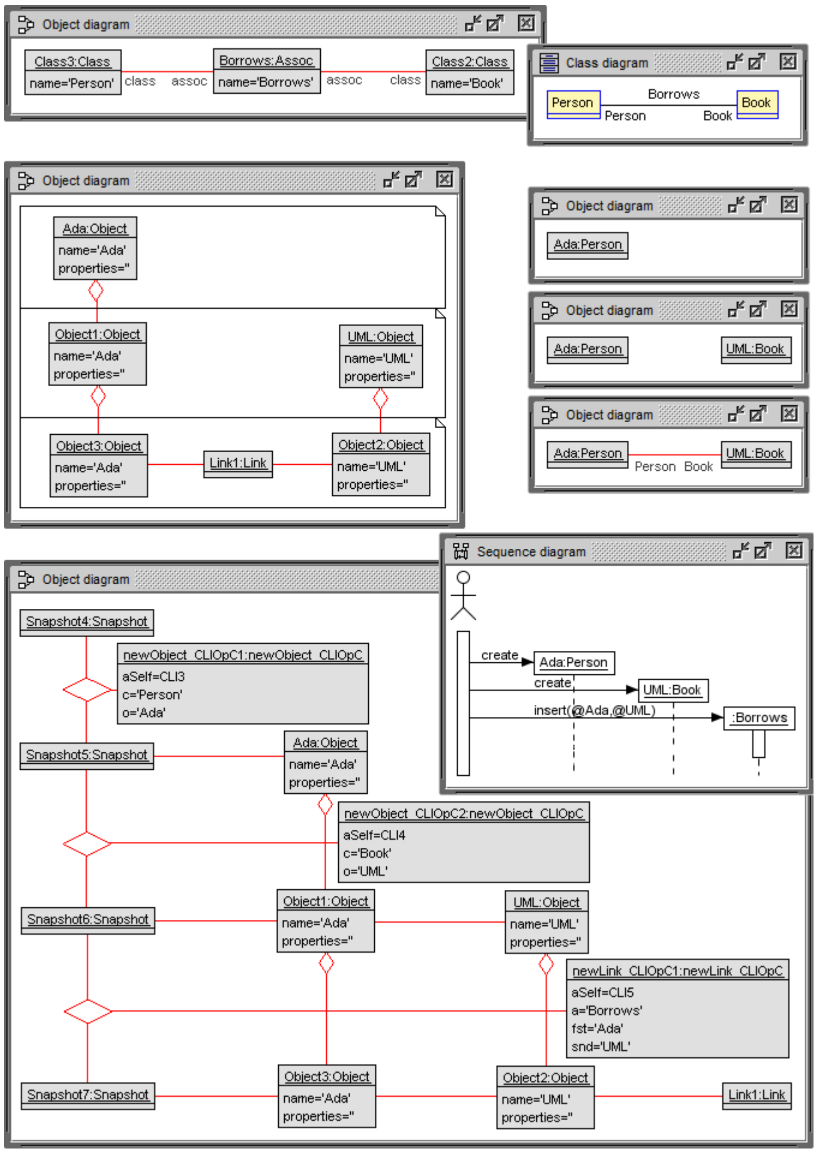


Fig. 3. Derived Example Models: Class model, Object model, Interaction model

after execution of the command and operation call sequence. In the example execution, design time elements are handled before runtime elements. However, this must not necessarily be the case. In general, calls handling runtime items may also be executed before design time calls.

Figure 3 pictures in the left three derived models that can be regarded simply as views on the larger object diagram in Fig. 2. All items from Fig. 3 occur in Fig. 2, however, they are differently arranged.

Class model view: In the upper part, a class diagram view is presented. The two classes Person and Book and the association Borrows are shown. These three items are available for the first time in Snapshot4.

Object model view: In the middle part, three evolving object diagrams are pictured, namely the Person object Ada and the Book object UML. The shown items are taken from the snapshots Snapshot5, Snapshot6 and Snapshot7.

Sequence diagram view: In the lower part, the considered items are arranged in a sequence diagram-like style. Four imaginary sequence diagram lifelines are present: (a) the vertically arranged Snapshot objects can be imagined as an actor in the sequence diagram; (b) three lifelines for the object Ada, the object UML and the link between Ada and UML can be imagined. The OperationCall objects represent the messages from the imaginary sequence diagram. Here, the snapshot objects Snapshot4 to Snapshot7 together with their connecting OperationCall objects have been selected and displayed.

When displaying the models in the left of Fig. 3, we have taken the currently available options in USE. Instead, one could choose a different visual syntax that is closer to the purpose of the considered fraction of the object diagram, as indicated in the right of the figure.

3 Technical Realization

In our view, the architecture and operation mode of our approach can be visualized as shown in Fig. 4. It captures the four different layers and makes clear that: (a) the design time model can be used on its own; (b) the design time model and the runtime model can be used together on their own; (c) this analogously holds for the remaining models. This section goes through the four models in Fig. 1 and Fig. 4 and discusses relevant details.

Design time model: We have kept this model rather simple, because our aim in this contribution was to explain our overall approach with a manageable example. The two classes Class and Assoc with their connecting association define that an Assoc object (an association) is binary with exactly two implicit association ends. Naturally, also more involved models can be used as design time models. Please note that the PredSucc associations are part of the filmstrip model, not part of the design time model. An analogous statement holds for the runtime model and user interface model: PredSucc associations are part of the filmstrip model.

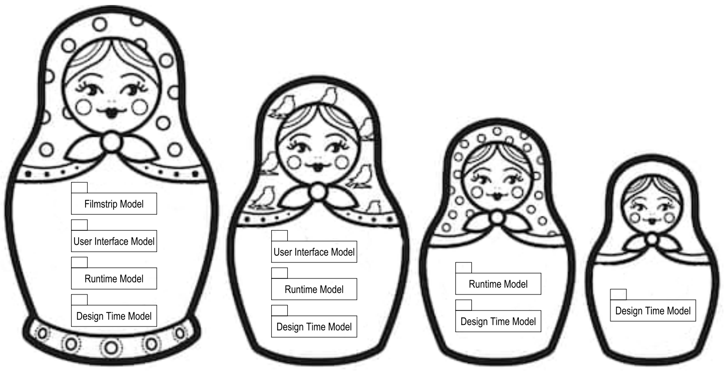


Fig. 4. Russian Doll Architecture Showing Model Usage.

Runtime model: The runtime model has counterparts for the items occurring in the design time model. The two different parts of a link are modelled by two different associations. The correspondence model connecting the design time model and the runtime model consists of the two 'typing' associations ClassObject_DT_RT and AssocLink_DT_RT. These associations specify for an Object its Class and for a Link its Assoc. The runtime model brings the design time model into life by allowing to instantiate classes and associations.

User interface model: This model consists of the single class CLI only (Comand Line Interface). It has operations to construct a new class, a new association, a new object and a new link. Classes, associations and objects have unique names. Additional query operations retrieve items when provided names as parameters. For example, the query operation CLI::class(c:String):Class returns the Class object cls for which cls.name=c holds. In Fig. 2, we have for instance CLI6.class('Person') = Class9. The following listing shows how essential operations are implemented in the imperative language SOIL (Simple Ocl-like Imperative Language) that is part of USE.

```
class CLI
operations
newClass(c:String)
  begin
  declare v:Class; v:=new Class(c); v.name:=c
  end
newAssoc(a:String,fst:String,snd:String)
  begin
  declare v:Assoc; v:=new Assoc(a); v.name:=a;
  insert (self.class(fst),self.assoc(a)) into ClassAssoc;
  insert (self.class(snd),self.assoc(a)) into ClassAssoc;
  end
newObject(o:String,c:String)
```

```
    begin
    declare v:Object;
    v:=new Object(o); v.name:=o; v.properties:='';
    insert (self.class(c),self.object(o)) into ClassObject_DT_RT
    end
  ...
  class(c:String):Class = Class.allInstances->any(e|e.name=c)
  assoc(a:String):Assoc = Assoc.allInstances->any(e|e.name=a)
  object(o:String):Object = Object.allInstances->any(e|e.name=o)
  ...
  end
```

Filmstrip model: The user interface model is automatically transformed into a so-called filmstrip model. A filmstrip model consists of snapshots that make system states from the user interface model (and the design time and runtime model) explicitly accessible. The purpose of the filmstrip model is to record a complete sequence of operation calls from the user interface model, so that one can trace the system development in terms of the snapshots and the transitions between the snapshots in form of operation calls. The filmstrip model includes all PredSucc associations that are used to describe the incarnations of a particular object in later snapshots. For example, in Fig. 2 for the object Ada in Snapshot5 its later incarnations are Object1 and Object3 with PredSuccObject links between them.

Object attributes could be handled by the property properties. In our examples we currently do not make use of attributes. The user interface model already provides the operation setProperties for this pupose.

Essential for our approach is the option allowing the developer to derive from the filmstrip object diagram specialized models that help to document and to better comprehend the design time and the runtime structure and behavior. The example models in Fig. 3 show how structural models like class or object diagrams and behavioral models like sequence diagrams may be derived. As future work, we plan to provide the option to define a domain-specific visual syntax in order to make the diagrams in Fig. 3 look like proper UML class, object and sequence diagrams, as we have already indicated in the right of the figure. We envision that further UML diagram forms like communication or statechart diagrams may be derived from filmstrip object diagrams. For example, from a filmstrip object diagram, one may derive a protocol state machine for books specifying that the operations borrow and return alternate between states available and borrowed.

4 Related Work

In Fig. 5 on the left side (part of the) the well-known OMG 4 level architecture and on the right side essentials of our approach are shown. Roughly speaking, we are mapping the M2 level to the M1 level. This has the advantage that the

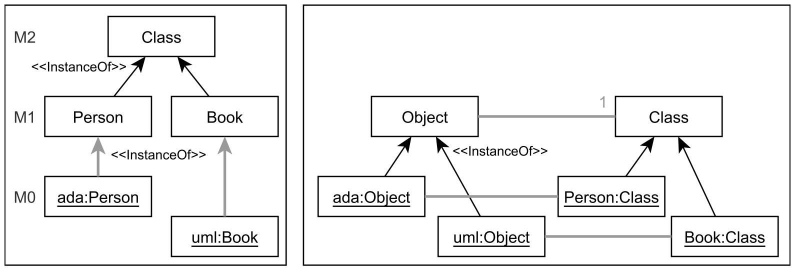


Fig. 5. Comparison of OMG 4 Level Architecture with Our Approach.

InstanceOf relationship between M2 and M1 now becomes explicitly available as an association and is made precise. Bridging the gap between design and runtime aspects is one of the significant challenges when developing complex systems [3], and several works are presented recently in this direction. In [15] the need to shift model-based assurance cases from design time to runtime is discussed. In this paper, the authors conclude that runtime assurance cases could be a potential solution for assuring safety-related Cyber-Physical Systems. A solution introduced in [1] makes executing UML design models directly on embedded microcontrollers possible thanks to a model interpreter. In [13], a set of modeling patterns extracted from the literature is introduced. These patterns are organized in a pattern language and consist of the specification for setting up the models and the environment for the analysis of runtime behavior utilizing design models. The authors in [9] introduce ModelPlex, a method which combines offline verification of Cyber-Physical System (CPS) models with runtime validation of system executions for compliance with the model. In [7], the authors present a design pattern, Aggregate Callback, for building DSL-based models in a robust and flexible way by enforcing constraints in the model so that the consistency of the output is guaranteed. The approach in [2] discusses the Requirements Modeling Languages (RML) and proposes a conceptual distinction between design time and run time requirements models. Run time models extend design time models with additional information about execution of system tasks.

5 Conclusion

The problem discussed in this contribution was how to provide a generic infrastructure in that design time and runtime models can be applied and properties of these models can be retrieved. The goal was to achieve such properties in a semi-automatic fashion. We started from a design time and runtime model, proposed to develop a common user interface and then automatically generated a filmstrip model that is able to record development steps. From this filmstrip model, we were able to extract other models that represented artifacts occurring in the development process and that help in comprehending the development.

As future work, we want to develop language support in form of a visual domain-specific language for the different layers, namely the design time, the runtime and the user interface layer on the basis of the filmstrip model. Tool support for the different layers must be extended, e.g., layout and view support for differentiating between design time and runtime. Furthermore, larger case studies in particular with more advanced correspondence models and a wide range of derived diagram forms must check the applicability and practicability of the proposed approach.

References

1. Besnard, V., Brun, M., Jouault, F., Teodorov, C., Dhaussy, P.: Embedded UML model execution to bridge the gap between design and runtime. [8] 519–528
2. Borgida, A., Dalpiaz, F., Horkoff, J., Mylopoulos, J.: Requirements models for design- and runtime: a position paper. In Atlee, J.M., Chechik, M., France, R.B., Gray, J., Paige, R.F., Rumpe, B., eds.: Proc. 5th Int. Workshop Modeling in Software Engineering, MiSE 2013, IEEE Computer Society (2013) 62–68
3. Brunelière, H., Eramo, R., Gómez, A., Besnard, V., Bruel, J., Gogolla, M., Kästner, A., Rutle, A.: Model-driven engineering for design-runtime interaction in complex systems: Scientific challenges and roadmap - report on the mde@derun 2018 workshop. [8] 536–543
4. Gogolla, M., Hamann, L., Hilken, F., Kuhlmann, M., France, R.B.: From Application Models to Filmstrip Models: An Approach to Automatic Validation of Model Dynamics. In Fill, H., Karagiannis, D., Reimer, U., eds.: Proc. Modellierung (MODELLIERUNG'2014), GI, LNI 225 (2014) 273–288
5. Gogolla, M., Hilken, F., Doan, K.H.: Achieving Model Quality through Model Validation, Verification and Exploration. Journal on Computer Languages, Systems and Structures, Elsevier, NL (2017) Online 2017-12-02.
6. Kästner, A., Gogolla, M., Doan, K., Desai, N.: Sketching a model-based technique for integrated design and run time description. [8] 529–535
7. Kövesdán, G., Asztalos, M., Lengyel, L.: Aggregate callback - A design pattern for flexible and robust runtime model building. In Hammoudi, S., Pires, L.F., Desfray, P., Filipe, J., eds.: Proc. Modelsward 2015, SciTePress (2015) 149–156
8. Mazzara, M., Ober, I., Salaün, G., eds.: STAF 2018 Collocated Workshops. Volume 11176 of Lecture Notes in Computer Science., Springer (2018)
9. Mitsch, S., Platzer, A.: Modelplex: verified runtime validation of verified cyber-physical system models. Formal Methods in System Design **49**(1-2) (2016) 33–74
10. OMG, ed.: Object Constraint Language, Version 2.4. OMG (2014) OMG Document, `www.omg.org`.
11. OMG, ed.: Unified Modeling Language, Version 2.5. OMG (2015) OMG Document, `www.omg.org`.
12. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language 2.0 Reference Manual. Addison-Wesley, Reading (2003)
13. Szvetits, M., Zdun, U.: A pattern language for manual analysis of runtime events using design models. In: Proc. 23rd European Conf. on Pattern Languages of Programs, EuroPLoP 2018, ACM (2018) 15:1–15:24
14. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley (2003) 2nd Edition.
15. Wei, R., Kelly, T., Reich, J., Gerasimou, S.: On the transition from design time to runtime model-based assurance cases. In Hebig, R., Berger, T., eds.: Proc. MODELS 2018 Workshops. CEUR Proceedings, Vol. 2245 (2018) 56–61