On Formalizing UML and OCL Features and Their Employment to Runtime Verification

Lars Hamann, M. Sc.

27.02.2015

Kumulative Dissertation
zur Erlangung des Grades eines Doktors der
Ingenieurswissenschaften
– Dr.-Ing. –

Vorgelegt im Fachbereich 3 (Mathematik und Informatik)
Universität Bremen

Gutachter
Prof. Dr. Martin Gogolla (Universität Bremen)
Prof. Dr. Hans-Jörg Kreowski (Universität Bremen)
Prof. Antonio Vallecillo Moreno (Universidad de Malaga)

Datum des Promotionskolloquiums: 25. Februar 2015

In Memorial of

Wilfried Nübel
⋆15. Mai 1927
†6. August 2013

Hans-Jürgen Hamann
⋆20. Februar 1953
†23. März 1996

Abstract

Model-driven development (MDD) has been identified as a promising approach for developing software. By using abstract models of a system and by generating parts of the system out of these models, one tries to improve the efficiency of the overall development process and the quality of the resulting software. In the context of MDD the Unified Modeling Language (UML) and its related textual Object Constraint Language (OCL) have gained a high recognition. To be able to generate systems of high quality and to allow for interoperability between modeling tools, a well-defined semantics for these languages is required.

This thesis summarizes published work in this context that employs an endogenous metamodeling approach to define the semantics of newer elements of the UML. While the covered elements are exhaustively used to define relations between elements of the metamodel of the UML, the UML specification leaves out a precise definition of their semantics. Our proposed approach uses models, not only to define the abstract syntax, but also to define the semantics of UML. By using UML and OCL for this, existing modeling tools can be used to validate the definition.

The second part of this thesis covers work on the usage of UML and OCL models for runtime verification. It is shown how models can still be used at the end of a software development process, i. e., after an implementation has manually been added to generated parts, even though they are not used as central parts of the development process. This work also influenced the integration of protocol state machines into a modeling tool, which lead to publications about the runtime semantics of state machines and the capabilities to declaratively specify behavior using state machines.

Zusammenfassung

Die modellgetriebene Entwicklung wird als ein vielversprechender Ansatz für die Softwareentwicklung angesehen. Durch die Verwendung von abstrakten Modellen als Grundlage für die Codegenerierung wird versucht die Produktivität des Entwicklungsprozesses und die Qualität der erzeugten Produkte zu steigern. Im Kontext der modellgetriebenen Entwicklung haben sich die Unified Modeling Language (UML) als grafische Modellierungssprache und die dazugehörige textuelle Object Constraint Language (OCL) als de-facto Standard etabliert. Damit qualitativ hochwertige Systeme aus Modellen dieser Sprachen erzeugt werden können und um die Interoperabilität von Modellierungswerkzeugen zu erhöhen ist eine wohldefinierte Semantik dieser Sprachen notwendig.

Im ersten Teil dieser Arbeit werden Veröffentlichungen zusammmengefasst, die sich mit der modellbasierten Definition der Semantik von neueren UML Sprachelementen beschäftigen. Dieses semantische Modell der UML wird dabei mit Hilfe der UML selbst und zusätzlichen Regeln in OCL beschrieben. Ein Vorteil dieses endogenen Ansatzes ist, dass keine zusätzlichen Sprachen benötigt werden. Weiterhin können vorhandene UML und OCL Werkzeuge verwendet werden, um das erstellte semantische Modell zu validieren.

Der zweite Teil dieser Arbeit beschreibt einen UML und OCL basierten Ansatz zur Laufzeitverifikation von Systemen. Die als UML und OCL Modell festgelegte Spezifikation eines Systems wird dabei während der Ausführung verifiziert. Dieses Vorgehen erhöht den Wert von Modellen innerhalb des Entwicklungsprozesses, da diese auch dann noch Verwendung finden, sollte das System nicht komplett aus dem Modell generiert werden. Die Entwicklung dieses Ansatzes führte darüber hinaus zu einer Integration von Protokollzustandsautomaten in ein Modellierungswerkzeug, da diese die Verifikationsmöglichkeiten erhöhen. Dieses führte zu weiteren, in dieser Arbeit zusammengefassten Beiträgen über die Laufzeitsemantik dieser Automaten und über die zusätzlichen Modellierungsmöglichkeiten.

Acknowledgments

I am very grateful to my supervisor Prof. Dr. Martin Gogolla for giving me the opportunity to work in his research group. Not only the fruitful discussions about our work, but also his support in areas not covering our research made him a special person in my life. I also would like to thank Prof. Dr. Hans-Jörg Kreowski for taking the task as my co-supervisor and for examining this theses. Additional thanks go to Prof. Antonio Vallecillo for examining this theses as the third referee. I really enjoyed working in the database systems group where I met many people who influenced this work in several directions. I would like to thank my colleagues (in order of appearance) Dr. Fabian Büttner, Mirco Kuhlmann, Oliver Hofrichter, Frank Hilken and Matthias Sedlmeier for their support and inspiring working atmosphere. A special thanks goes to Frank and Matthias for taking the burden of reviewing this thesis and giving me constructive feedback on how to improve it. I am glad that I had the possibility to discuss my research with people around several conferences, especially the vital community at the MODELS conferences. Finally, I would like to thank my mother Dagmar for supporting me in so many ways during my life, the Loosers for deepening my understanding about rules of games, and last but not least Inger who stayed with me even though she had the burden of getting into my life while I was finishing this thesis.

Contents

List of Figures

1 Introduction

Developing software has always been a complex and expensive task. Different shifts of paradigm in the area of software engineering improved the productivity of the development process and the quality of the resulting products. Most of these shifts raised the level of abstraction a developer can use to build systems, e. g., by using more powerful 3rd generation programing languages and their compilers instead of using assembler, which allows a developer to focus on a problem while hiding details of the concrete hardware. However, in [38] the author argues that using a more and more elaborated high-level language will at some point become a burden that increases the intellectual task of programming instead of reducing it, since such a language would include many constructs that are rarely used. In the same essay, the author is very skeptical about the use of visual representations of programs to increase productivity or quality. This was put into perspective in [39]. In particular, he agrees that using multiple types of diagrams, each providing a different view on a system, can support the design of systems, but for some elements, like for instance algorithms, a textual representation is still the most suitable one.

The Unified Modeling Language (UML) [78, 67] maintained by the Object Management Group (OMG) follows this view by providing a rich set of different kinds of diagrams. Moreover, the idea of Model Driven Architecture (MDA) [26, 43, 65], as proposed by the OMG, puts models into the heart of the development process. In MDA, a model can be specified using any well-defined language. The requirements for such a well-defined language are [42], that it has a well-defined form (syntax) and meaning (semantics). Further, it must be suitable for automated interpretation by a computer. This definition allows the usage of many kinds of languages, e. g., very specific ones like Java or more abstract ones like the UML. To gain benefits of applying MDA, one starts by using an abstract language to define a so-called platform independent model (PIM). During the phases of development, this PIM is transformed into more platform specific models (PSM). Consequently applied, this leads to a higher productivity, since different PSMs can be generated from a single PIM. An improvement of the quality can be achieved by the fact that transformations can be reused for different PIMs. In the literature, many different terms, like for instance, model-driven development (MDD) and model-driven engineering (MDE) are used to refer to a development process that uses models as central parts [85].

To what extend the usage of models improve aspects of a development process has been studied in [85]. In a survey among 155 Italian software professionals taken in 2011,

the authors conclude that "modelling [can be classified] as a *highly relevant* technique in the Italian industrial context while MD*[1] can be considered as *relevant*" [85]. The authors separate between modeling as a supporting technique and a model-driven process. The former uses some simple models that are not used exhaustively during the complete development process, whereas the latter implies that models play an important role during the whole development process. The survey also revealed that benefits like "*Support in design definition*, *Improved documentation*, easier *Maintenance*, and higher *Quality* seem to be obtained when simple models are used and no further improvement is observed with MD* adoption" [85]. However, "MD* plays a significant role for *Productivity*, *Platform independence* and *Standardization*" [85]. One main problem in practice identified in this survey is the *lack of supporting tools*, i. e., tools cannot be easily replaced by alternatives and building heterogeneous tool-chains is hard.

The work presented in this thesis addresses two of these issues. First, the problem of tool interchangeability is addressed by strengthening the semantics of modeling concepts used in the de-facto standard modeling language UML. Providing a clear semantics reduces incompatibilities between different modeling tools used in a tool chain or when replacing tools. For this, work on less well-defined concepts in the UML has been published in [A18C, A22C, A16C]. Further, it has been shown in [A15C], how to define a runtime semantics for UML constructs using an endogenous metamodeling approach like the one presented in [42] for the Object Constraint Language (OCL) [92, 63]. Second, an approach is presented, where models can still be useful at the end of a non complete model-driven development process [A19W, A24C, A21C, A17W]. This approach allows a developer to verify assumptions made in a model against concrete implementations. Therefore, models designed in an early phase can still improve the quality of the product even without applying code generation or model-based testing.

Thesis Structure

The foundations for this work are presented in Chap. 2. First, the UML and its history is described. OCL, which is widely used in combination with UML models is presented afterward. Based on these two languages, the modeling tool USE, which is based on both languages is introduced, since much of the work presented in this thesis is validated by an implementation in this tool. Next, the concept of runtime verification is explained. In Chap. 3, contributions to formalize and strengthen the specification of UML and OCL are presented. This work improves the usability of the work presented in the following Chap. 4, which explains a runtime verification approach using UML and OCL and its implementation in the USE tool. The chapter ends with example applications of the approach. Chapter 5 summarized additional publications of the author that were not covered before. The thesis ends with a conclusion and an outlook to future work.

[1] The authors use the notion of MD* to cover all model-driven approaches like MDE, MDD or MDA.

Citations

All references prefixed with an `A` are publications the author was involved in, either as author or co-author. These publications are further categorized by a suffix to distinguish between articles in proceedings of conferences (`C`) or workshops (`W`) and journal articles or chapters in edited volumes (`J`).

The chapters 3 and 4 include parts of published papers that were reworked to fit into the overall structure of this thesis. For readability reasons, these included parts are not highlighted as citations. However, each chapter refers to the used publications.

2 Background

This chapter introduces concepts and artifacts important to this thesis. First of all, the modeling languages used in this thesis: the Unified Modeling Language (UML) and the related textual Object Constraint Language (OCL) are introduced. Next, the modeling tool USE and its approach to validate and verify UML and OCL models is explained. The last section provides an overview about the concept of runtime verification.

2.1 The Unified Modeling Language (UML)

With the paradigm shift from procedural to object-oriented programming in the 1980s, several scientists and practitioners recognized a need for visualizing object-oriented designs. This graphical representation was at first used to ease the communication between software developers. Three different methods, out of many others, each using its own notation to represent object-oriented concepts gained a high recognition and were invented nearly at the same time in the 1990s ([78, p. 5], [21, p. 52]): the Object-Oriented Analysis and Design method (OOAD) by Grady Booch [10], the Object-Oriented Software Engineering method (OOSE) by Ivar Jacobsen [35], and the Object Modeling Technique (OMT) by James Rumbaugh [77]. Following [78, p. 4ff.], these three amigos, as they were called later, joined the same company between the years 1994 and 1995. They merged their work, which they called the Unified Modeling Language (UML). This unified language was later standardized by the Object Management Group (OMG) in 1997. After some years of development, UML 2 was released in the year 2005[1]. UML 2 addressed issues that arose during the application of UML 1.X. Further, the UML metamodel that defines the abstract syntax of the language using the same notation as UML was unified with the Meta-Object Facility (MOF) [78, p. 7].

While UML started as a language to represent object-oriented structures in an abstract way using graphical elements, at least two other application areas arose for it: the idea of model-driven development, which tries to improve the efficiency of software development processes by generating most of the parts of a software system out of a model [65] and the definition of abstract syntax descriptions for languages as it is done for the UML itself.

UML defines diagrams for different aspects of a system. For example, an abstract view to the different tasks that can be done with a system is provided by means of

[1] All release dates of the various UML versions can be found on the OMG website [67].

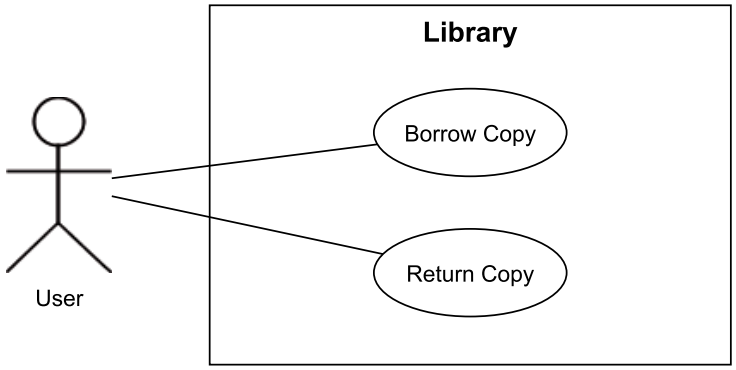


Figure 2.1: Example Use Case Diagram

use case diagrams, or the structure of a system can be defined and revealed using class diagrams. Modeling the dynamics of a system is supported by behavioral diagrams, like for example sequence and activity diagrams.

2.1.1 Language Description

In this section, we informally introduce UML diagrams required for this thesis using a small but adequate example. A formal description of most of the described language elements can be found in [73]. Our example model shall represent a simplified library. Following the idea of use case driven development, we start to introduce our example by such use cases.

Use Case Diagram

Use case diagrams provide a graphically overview about the actors of a system and the tasks, i. e., use cases, they can perform. In such a diagram, the system boundary is depicted by a rectangle. Actors are outside of this boundary and are linked to the use cases they participate in by solid lines. The actor `User` in Fig. 2.1, for example, is related to the use cases `Borrow Copy` and `Return Copy`.

The term *use case* is often defined as an interaction, i. e., the exchange of messages, an actor can perform with a system to gain some benefit. It describes the usage of a system from an external view ignoring internals [78, p. 78]. A use case itself can be defined using different notations. In an early stage of a system design this might be natural language [21, p. 73ff.]. Afterwards, while the model of the system evolves, the informally defined use cases can be specified more precisely by using other UML notations, like activity diagrams. In Fig. 2.2 on the next page an informal description of the use case `Borrow Copy` is given. It describes the normal behavior of the system together with

Use case#:	**UC001**	**Name:**	**Borrow Copy**	**Version:**	**1.0**
Author:	Lars Hamann	Created:	18.08.2014	Modified:	-
Participants:	User				
Short Desc.:	A user borrows an existing copy				
Preconditions:	• The user is registered • The copy is available • The user has not exceeded the maximum number of currently borrowed copies				
Postconditions:	• The copy is marked as borrowed by the user				
Included UC:	none				
Extended UC:	none				
Activity:	• The user grasps a copy she wants to borrow • She scans her identity card • She scans the signature of the copy • She gets an receipt that includes information about the return date				

Figure 2.2: Use Case Described Using Natural Language

conditions that need to hold before and afterward. Using the provided example, one can infer that the entities `User` and `Copy` are central parts of the system. The entity `User` is required to be able to identify a user by its identity card. Whereas, `Copy` is required to store information about available books in the library. Since a library can have more than one copy of the same book, the use case description talks about copies instead of books. It is further implicitly stated that the system needs to keep track of the currently borrowed copies by a user, since there is a maximum number of copies a user can borrow at the same time.

Beside the shown relation of actors and use cases, the UML allows further information inside a use case diagram to model more complex situations. For example, use cases can inherit from others to reuse and specialize generic activities. If a use case includes another one, this can be expressed by adding the stereotype «*includes*» to the relation between the including and the included use case. An extension of a use case can be defined in the same manner except that the stereotype «*extends*» is used. A possible extension of the example use case could be to allow a user to borrow multiple copies after she identified herself.

Class Diagram

Class diagrams define the static structure of a system. They show how the different classifiers[2] relate to each other. Classes are drawn as rectangles that have different

[2]Classifier is a more general concept in UML that covers, for example, associations, enumerations, and classes. During this introduction we do not distinguish between those kinds.

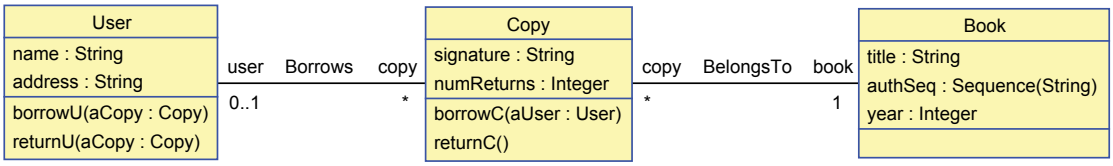


Figure 2.3: Class Diagram of the Running Example [A22C]

compartments. The first compartment is used for the name of the class and possibly applied stereotypes. The second one lists the attributes a class owns. The last one lists the operation signatures of a class. Classes can participate in different relations to each other. One relation type is generalization that defines a general class (the parent) and a specialization (the child). In object-oriented programming languages, these are sometimes called super- and subclass. A subclass can add new features or can redefine existing ones. In a class diagram a generalization is depicted as an edge between the related classes that has a large hollow triangle directing to the more general class.

Another important relation between classes are associations. A simple distinction between generalization and association is commonly explained by the phrases "is a" for generalizations and "has a" for associations. For example, a user *is a* person, meaning the entity user is a specialization of the entity person. Whereas, a user *has a* copy borrowed, which means a user can be related to at least one copy. Associations are drawn as a solid line between classes. One end can be marked as an aggregation using an empty diamond or as a composition using a solid diamond. Both kinds mark an association as a whole-part relationship, with the class at the diamond end acting as the *whole*. A composition strengthens this relation by stating that the instances of the class acting as the whole is responsible for creating and destroying its parts.

Figure 2.3 shows a class diagram, which covers most of the requirements that were previously defined by the use case in Fig. 2.2 on the previous page. The library system keeps care of copies of books. A `Book` has a title, an ordered collection of authors and a publishing year. Copies are related to their corresponding book by an association `BelongsTo`. A `Copy` itself has a unique signature and an attribute `numReturns` that counts how often a copy was borrowed and returned. Copies can be borrowed and returned by users. A copy is borrowed, if a copy instance is linked to a `User` instance. Operations to borrow or return a copy are defined on the class `User` and on the class `Copy`. To easily distinguish the operations they are suffixed with a `C` for the copy class and a `U` for the user class.

Object Diagram

An object diagram exemplifies the structure of a system by containing concrete instances of classes, also called objects, and of associations, called links. Objects are drawn like

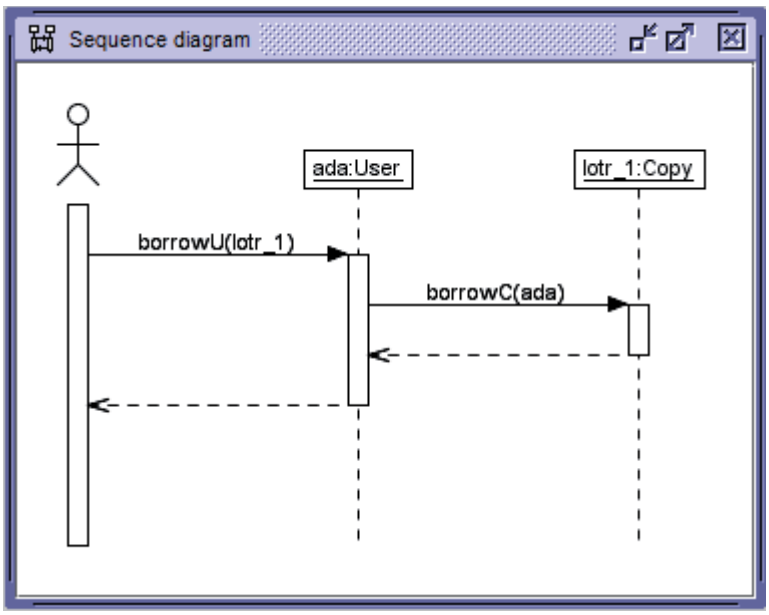


Figure 2.4: Sequence Diagram for a Borrow Scenario

classes but differ in small details. First, object rectangles do not have a compartment for operations, since they are specific to classes. Second, they can be named to give a more precise meaning to the provided example state. To differentiate objects from classes, the name of the object and the name of its class are printed underlined inside the name compartment separated by a colon. The attributes of an object are pictured together with their concrete values in the second compartment. Links are shown like associations inside a class diagram.

Sequence Diagram

While object diagrams show a concrete system state at a single point in time, sequence diagrams highlight the order of messages sent between instances, as it can be seen in Fig. 2.4. The order in time is given by the vertical position of a message in a diagram. Where time is proceeding down the page. While the UML also allows to include control structures like loops and conditional branches inside a sequence diagram, we focus on concrete execution scenarios, which do not include such constructs. Each participating instance in a sequence diagram is represented by an object rectangle showing the name and the class of the instance (like for example the object `ada` in the sequence diagram in Fig. 2.4). Each object rectangle is shown directly before it is created. Since both participating objects shown in the example diagram are already alive, they are shown at the very top of diagram. Connected to the rectangles are the lifelines of their instances.

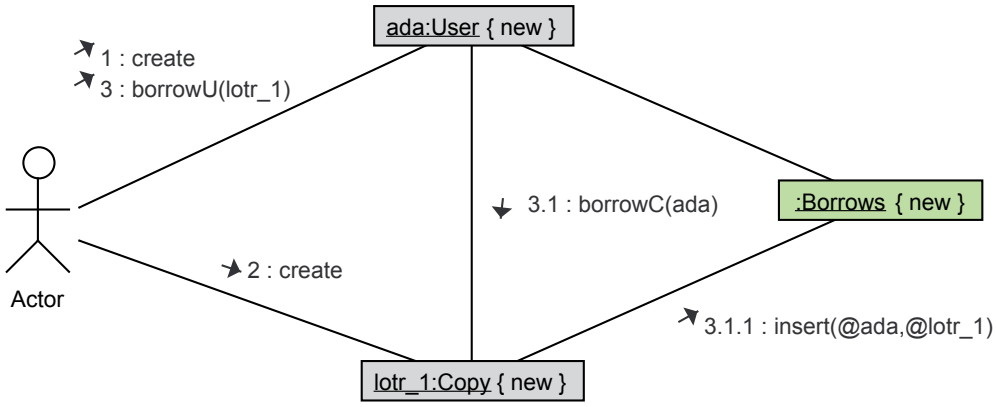


Figure 2.5: Communication Diagram of a Borrow Scenario

If an object is destroyed, a cross at the end of the lifeline is shown. The vertical rectangles on the lifelines highlight the time an execution specification of a procedure on an object is active [78].

Communication Diagram

The focus of communication diagrams is to identify instances communicating with each other. In contrast to sequence diagrams the occurrence of messages is not ordered by their vertical appearance. Instead messages are numbered labels of the links between communicating instances. Figure 2.5 shows the same execution trace as the sequence diagram in Fig. 2.4 on the previous page including some more details. As it can be seen, both messages (`borrowU(lotr_1)` sent to the object `ada` and `borrowC(ada)` sent to the object `lotr_1`) are also present in the communication diagram. The order of their execution is determined by the numbers 3 and 3.1. Instances participating in a communication diagram can be labeled with additional information about their lifetime as it is done in the example diagram by labeling all instances with `{new}`, which states that these instances are created during the communication and are still alive after it has finished. Other labels are `{transient}` for instances that are created and destroyed during the shown communication, `{destroyed}` for instances that were created before and deleted during the communication, and `{persistent}` for instances that were created before the shown scenario and are not deleted during it.

2.1.2 The UML Metamodel

The valid structures of UML models are defined by the UML metamodel, which itself is modeled using MOF (Meta Object Facility) [62]. In general, the number of these metamodel levels is not restricted, but for the UML the term *four-level metamodel hi-*

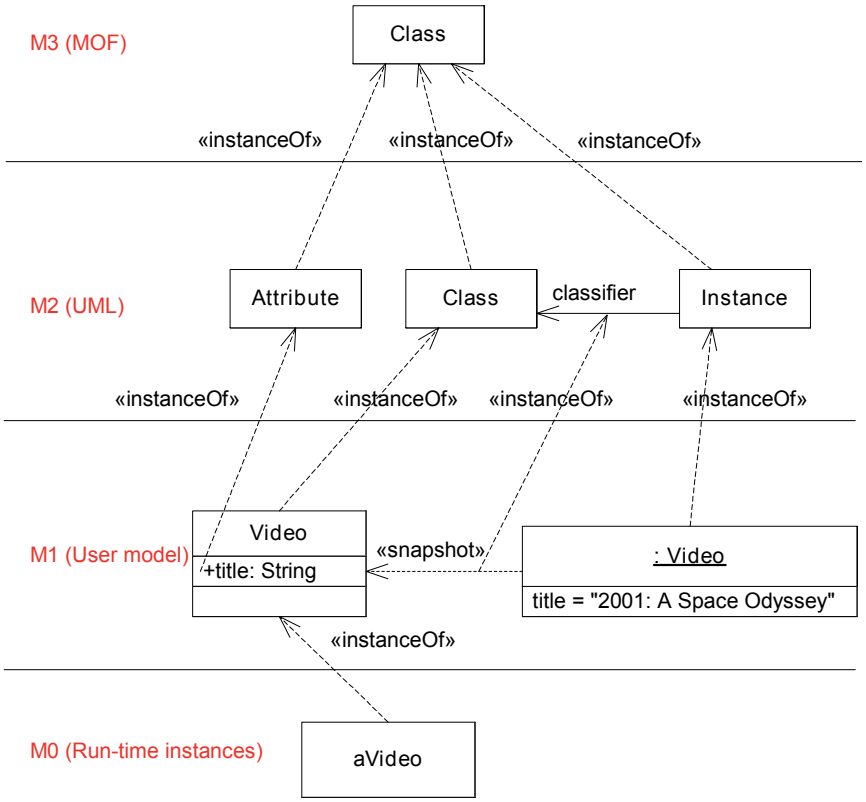


Figure 2.6: UML Four-Level Metamodel Hierarchy [68, p. 20]

erarchy [68, p. 19] is often used. In Fig. 2.6 this hierarchy is shown. Starting with level M0 that represents runtime instances of elements defined by a UML model, each model element of the following levels is an instantiation of its metamodel element.

For example, the attribute `title` of the class `Video` defined in level M1 (User model), i. e., the UML model, is an instantiation of the UML-metaclass `Attribute`[3]. This metaclass has an attribute `name` to be able to define the name `title` for the modeled attribute. The metaclass `Attribute` is an instance of the meta-metaclass `Class` of the level M3 (MOF). Important for this work is the metaclass `Instance`. Strictly spoken, instances of these metaclass are examples of possible snapshots of the model defined in level M1 and do also reside in this level. However, in this work we are going to use them on the level M0 by means of object diagrams.

[3]Note, that the metaclass `Attribute` was replaced by the metaclass `Property` in newer versions of the metamodel of UML, as we will see later.

2.2 The Object Constraint Language (OCL)

Representing the design of a software system using UML diagrams can improve the understanding of a system. When used as a specification language one soon gets to the point that not all relevant details of a system or business process can be expressed using the provided modeling elements of a graphical language. For instance, a simple business rule for the library example could be that a user is not allowed to borrow two copies of the same book for fairness reasons. This rule cannot be expressed by only using a class diagram and its syntactic elements. This is one example, why a need for additional specification possibilities was discovered in the very beginning of UML. The Object Constraint Language (OCL) was designed to overcome this lack of specification possibilities and is now the de-facto standard for defining constraints in the context of UML. Three kinds of constraints are very common in the context of OCL and are also well known from *Design by Contract* (DbC) [56] as it is supported, e. g., by Eiffel:

Preconditions Preconditions define the client part of a contract between an operation, i. e., the provider of a functionality, and its user, i. e., the calling client. The client needs to ensure the preconditions to expect the correct execution of the provider. The provider itself does not need to recheck its defined preconditions. In [56] it is argued, that a defensive programming style, that rechecks preconditions that could already be validated by a client, leads to more complex software systems and therefore introduces more faults.

Postconditions A provider of a functionality defines its part of the contract by providing postconditions. If a caller provides valid input – or in other words the preconditions were satisfied – the caller can expect the postconditions to hold. Special to postconditons is the possibility to access the system state before an operation was called. In OCL this is done by the keyword `@pre`. This allows to define changes that are made by an operation. For example, the following postcondition ensures, that the attribute `numReturns` of the context class `Copy` is incremented by one, during the execution of the operation `borrowC`:

```
1 context Copy::borrowC(aUser:User)
2  post: self.numReturns = self.numReturns@pre + 1
```

Invariants While pre- and postconditions define a specific contract between a client and a provider, like it is done by contracts in business, invariants can be seen like laws that must always be respected [56]. In other words, a contract between a client and a provider is not allowed to violate existing law. The same holds for operation calls. However, invariants inside of an object-oriented system need some special treatment, because while executing a method it is sometimes necessary to temporary violate an invariant.

While the OCL specification makes no assumptions about when an invariant must be satisfied, in [56] it is described, how it is done in Eiffel:

- An invariant mus be satisfied after the creation of an instance of the class for which the invariant is defined.
- An invariant must be preserved by every public routine of the class. By preserved, the author means that a routine must guarantee that the invariant is satisfied on exit if it was satisfied on entry.

In addition to the above usages for DbC, OCL is now used in a much broader context [92, p. 13]. For example, the Query/View/Transformation (QVT) approach of the OMG [61] uses it as a query language for model transformations. QVT further uses a derivative language of OCL called Imperative OCL[4].

2.2.1 Language Description

This section briefly introduces the concepts of OCL needed in this thesis. Readers who are familiar with OCL might skip this section. A more detailed work on OCL and in particular about the semantics of OCL can be found in [73], which laid a profound basis for the definition of OCL.

OCL itself is based on first order predicate logic and set theory. Further, it is a strongly typed language. Each valid expression in OCL has a type, that can be computed during the static analysis of the expression. This allows an OCL tool to report type errors before evaluating an expression. Because of this, type errors, except casting errors, cannot happen during the evaluation of an expression. Consider the following OCL expression that iterates over a set of values and checks if each value is greater than 5:

```
1 Set{1,'a'}->forAll(e | e > 5)
```

During type-checking, an OCL tool should report a type error, since the set to iterate over is of type `Set(OclAny)` and the operator `>` is undefined for the iterated type `OclAny`. Without starting a discussion about the benefits and drawbacks of strongly typed, typed, and untyped languages, this helps to discover erroneous expressions before instantiating a model.

Since OCL is primarily a constraint language, all OCL expressions are side-effect free, i. e., their evaluation does not change the current system state. For example, an expression cannot create new links or objects. An important requirement for defining expressions on models is the possibility to navigate over associations to be able to access connected objects and its attributes. Therefore, navigation is one of the most important features in OCL. Since association ends having an upper bound greater than one are

[4] For a detailed discussion about the integration of OCL into Imperative OCL and resulting issues, see [13].

very common and a navigation to such ends results in a collection of connected objects the OCL standard library provides powerful operations on collections, e. g., operations for projection, selection, and quantification. A complete list of predefined operations can be found in the OCL specification documents published by the OMG [63].

While an invariant is defined for all instances of a classifier, access to all instances inside an expression is sometimes needed, too. For this, OCL provides another powerful feature to access all instances of a classifier using the operation `allInstances()` on a given classifier, which evaluates to a set of values of the given classifier. For example, a constraint defining uniqueness of an attribute must access all other instances of the same classifier to be able to validate the value of the attribute of the context instance against all other values:

```
1 context User inv nameIsUnique:
2  User.allInstances()->forAll(u:User|
3          self <> u implies self.name <> u.name)
```

2.3 The UML-based Specification Environment (USE)

The modeling tool USE (UML-based Specification Environment) allows to specify, validate, and to a certain degree verify software models based on UML and OCL. Started as a project to formalize OCL [73] in the late 90s, the first public version of USE was available in 1998. Since that, USE was continuously extended and improved. By taking a look at the number of times USE was downloaded, we can state, that USE is widely known in the area of UML/OCL tools. In 2011 USE was downloaded some 1,500 times. The number of downloads increased to around 3,100 times in 2012 and to some 4,700 times in 2013.

After the first version of USE, several improvements and extensions have been integrated. Some of them were developed by students for their diploma theses other were integrated by members of the database systems working group. Some notable extensions that are not discussed in this thesis, are the integration of an imperative programming language based on OCL [13], the development of a plug-in architecture, to support easy extensions to USE without the need to change the USE source code [4], the extension of object diagrams with complex selection features in [36], and adding support for communication diagrams [58]. In the next section, the workflow to validate models as it is supported by USE is explained.

2.3.1 The USE Approach to Validation

In [73] the USE approach to validate models was introduced. Later, [13] extended this approach by adding use cases related to the integrated action language. In Fig. 2.7 on page 16 an extended use case diagram based on the one presented in [13, p. 127] is

depicted. The diagram provides an overview of the different tasks a modeler can perform with USE. Bold use cases highlight tasks that are going to be extended or introduced in this thesis. Next, the different tasks of the USE validation approach are described. These descriptions are only slightly different from the ones presented in [73, 13].

Specification

The preceding step for every other use case of USE is to specify a model. This is done by applying a textual USE-specific syntax, which is similar to the UML Human-Usable Textual Notation [60] published by the OMG. The top-level elements in each model are classes, associations and enumerations. Together with attributes and OCL-invariants, these elements allow to specify the structure of a model. To be able to specify behavior in a declarative way, like it is done in Design by Contract, pre- and postconditions can be defined for operations. As we will see in Chap. 3, the recently added protocol state machines can be used to specify behavior in a declarative way, too. With the integration of SOIL [13] in USE 3.0, an imperative way to specify behavior was added as an alternative for simulating models. USE can validate the static soundness of a given model. For this, basic validation of the structure is done. For example, the correct definition of classes, including attribute types and cycle-free inheritance can be checked. Further, the syntactical and type correctness of OCL-expressions can be validated.

Instantiation

After a valid model was loaded in USE, one can start to instantiate concrete system states. In USE a system state consists of

- instances of classes, i. e., objects,
- instances of associations, i. e., links, and
- values assigned to attributes of instances.

Depending on the users needs, a system state in USE can be created in different ways:

Shell USE always provides a shell to invoke commands. These include commands for creating and deleting objects and links and for setting attribute values. One benefit of using the shell is the possibility to use command files to restore a given system state.

GUI The easiest way to create system states is to invoke given commands by using the graphical user interface (GUI) of USE. The complete instantiation task can be done either using views or context menus. Especially, while using large models the GUI can help a user to get a more profound understanding of the model. For example, after selecting two objects, USE automatically shows up the possible associations these objects can participate in.

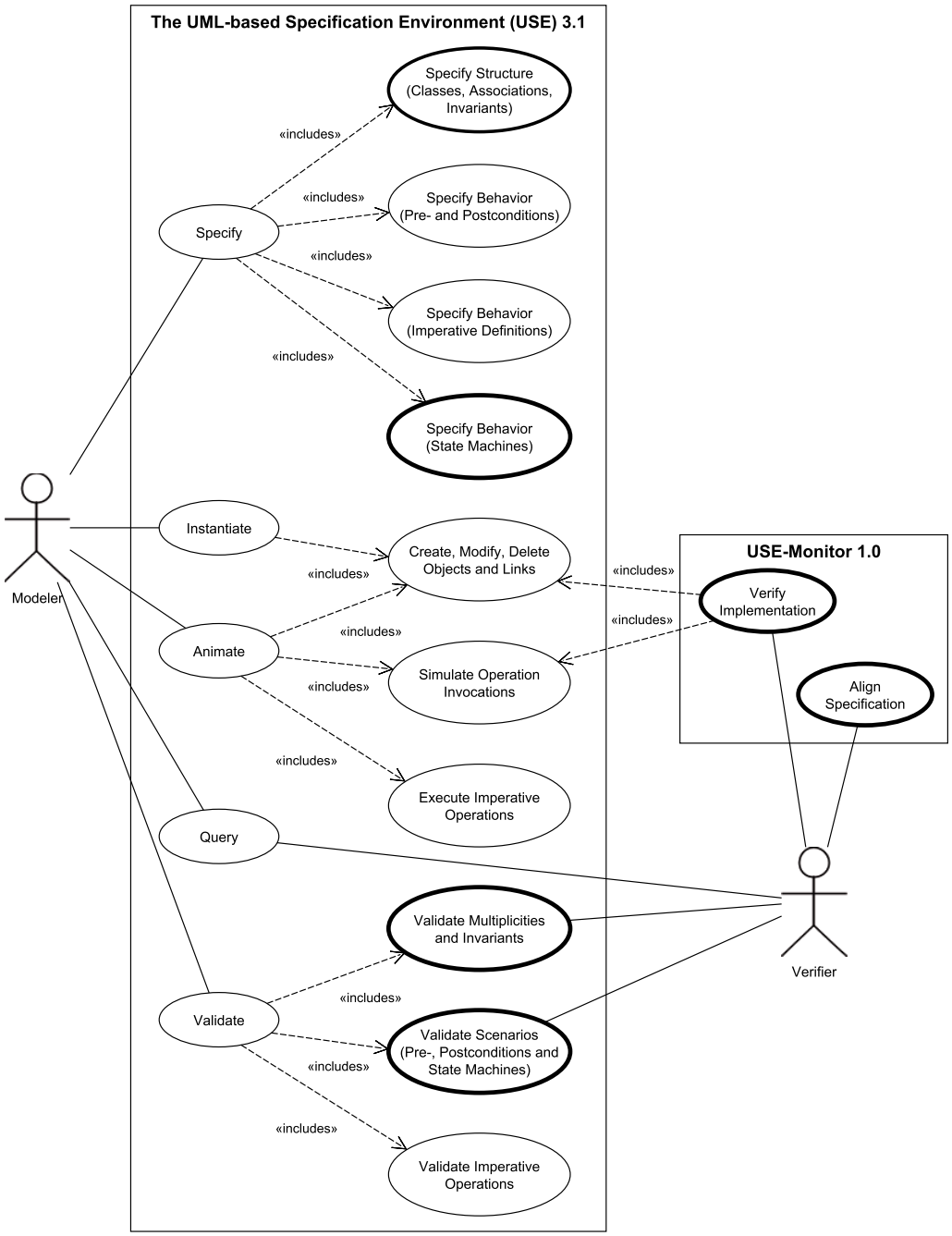


Figure 2.7: Use Case Diagram of the USE Approach to Validation

Validator The USE model validator (see for example [A27C] and [45]) is a black box approach of model finding. A modeler specifies certain properties of a model instance to search for, like the number of instances for each class or the number of links for each association. Using such a configuration, the model validator automatically searches for a valid model instance w. r. t. the supported model constraints. The validator itself is based on relational logic provided by the constraint solver Kodkod[5] [86].

Generator A white box approach of model finding is also present in USE. For this, a language called *ASSL* (A Snapshot Sequence Language, c. f. [28]) is applied to guide the search for a valid model instance. Although the model validator uses more advanced and faster search algorithms than the generator, ASSL and the generator still have their value. First, because the model validator does not support all features provided by USE. For instance, the possibility to define recursive operations, which cannot be expressed in relational logic used by the model validator. Further, since ASSL delegates the validation of model instances during the traversal of the search space completely to the USE core system, newly added OCL features are automatically available. In contrast to this, the model validator needs to be aligned to new USE versions, because it needs to translate new features into concepts of relational logic. For example, constrains defining relations between associations, as discussed in Sec. 3.2.1 starting on page 28, are automatically supported by the generator whereas the model validator needs to be extended to support them.

Animation

Two approaches of animating models exist in USE: simulation and execution. The former can be used to validate given scenarios, i. e., test cases, using concrete values and sequences of operation calls. The latter is based on imperative implementations of operations and can be used to test a concrete method implementation.

During simulation, a sequence of commands with given values is executed[6]. These commands represent a single trace of execution. Therefore, only atomic commands like `create` to create instances of classes or `set` to set attribute values can be used. Operation calls are simulated by the commands `openter` and `opexit`. Imperative constructs like loops or conditional checks are not supported during simulation. Listing 2.1 on the following page exemplifies this. Lines 1–7 define the class `User` with a single operation `borrowU(aCopy:Copy)`. Here, only the operation signature is defined. During the simulation of a borrow process (lines 9–16), this operation is entered by the command `openter` including concrete values for the arguments (here: `ada` borrows the copy `lotr_1`). The simulation continues by calling the operation `borrowC` of the class `Book`.

[5] `http://alloy.mit.edu/kodkod/`

[6] In fact, *given values* is not quite right in a formal way, since values can be computed by OCL expression.

Listing 2.1: USE Simulation Example

```
1  model LibrarySim
2
3  class User
4    ...
5   operations
6    borrowU(aCopy:Copy)
7  end
8
9  !create ada:User
10 !create lotr:Book
11 !create lotr_1:Copy
12 !openter ada borrowU(lotr_1)
13 !openter lotr_1 borrowC(ada)
14 !insert (ada,lotr_1) into Borrows
15 !opexit
16 !opexit
```

Listing 2.2: USE Execution Example

```
1  model LibraryExec
2
3  class User
4    ...
5   operations
6    borrowU(aCopy:Copy)
7    begin
8     aCopy.borrowC(self)
9    end
10 end
11
12
13 !aUser := new User('ada')
14 !aBook := new Book('lotr')
15 !aCopy := new Copy('lotr_1')
16 !aUser.borrowU(lotr_1)
```

This operation is not shown, but is defined in a similar way as the operation of the class `User`. By using the command `opexit` to simulate the return of an operation, the shown command sequence of operation calls leads to the sequence diagram shown in Fig. 2.4 on page 9. Another possible visualization of the executed commands in USE is provided by means of a communication diagram as shown in Fig. 2.5 on page 10. While atomic commands like `create` are hidden in the sequence diagram (this is an optional behavior), they are shown in the communication diagram.

For execution, in contrast to the previously described simulation approach, the operations are enriched with an implementation in the USE specific action language SOIL [13]. All of the previously described statements, like object creation, are also supported by SOIL. Furthermore, imperative control structures like conditional execution (`if`) and loops (`while` and `for`) can be used. An imperative implementation of the previously described borrow scenario is shown in Listing 2.2. After executing the command sequence (lines 13–16), the same execution trace as shown in the sequence diagram in Fig. 2.4 on page 9 and in the communication diagram in Fig. 2.5 on page 10 is recorded by USE.

Validation

The main task of USE is to validate model constraints. First, structural constraints, like multiplicities of association ends, can be checked. For this, no simulation or execution is necessary. One can create a system state and validate the structure of it. If multiplicities are violated, USE reports the violating association and the connected objects to guide the designer to the erroneous part of the system state. In addition, invariants can also be validated without animating a model. Again, USE reports violations to the

Listing 2.3: Violation of a Precondition in USE

```
1  use> !create ada:User
2  use> !create lotr:Book
3  use> !create lotr_1:Copy
4  use> !insert (ada,lotr_1) into Borrows
5  use> !ada.borrowU(lotr_1)
6  [Error] 1 precondition in operation call
7    'User::borrowU(self:ada, aCopy:lotr_1)' does not hold:
8    copyNotBorrowed: aCopy.user.isUndefined
9      aCopy : Copy = lotr_1
10     aCopy.user : User = ada
11     aCopy.user.isUndefined : Boolean = false
12   call stack at the time of evaluation:
13     1. User::borrowU(self:ada, aCopy:lotr_1)
14        [caller: ada.borrowU(lotr_1)@<input>:1:0]
15
16 +-------------------------------------------------------------------+
17 | Evaluation is paused. You may inspect, but not modify the state. |
18 +-------------------------------------------------------------------+
19
20 Currently only commands starting with '?', ':', 'help' or 'info'
21 are allowed.
22 'c' continues the evaluation (i.e. unwinds the stack).
23 > ?aCopy.user
24 -> ada : User
```

user and provides features to investigate violating instances. For invariants, a powerful evaluation browser can be used, which allows for a detailed evaluation of the values of sub-expressions. A more extensive description of the evaluation browser can be found in [A1C].

During animation of a model, USE evaluates pre- and postconditions defined for operations. If such a condition fails during the execution of a model (see Sec. 2.3.1 on page 17), USE stops the execution an provides a special shell to examine the given situation. Listing 2.3 shows an example violation of a precondition for the library example extended with the following precondition `copyNotBorrowed`:

```
1 context User::borrowU(aCopy:Copy) pre copyNotBorrowed:
2  aCopy.user.oclIsUndefined()
```

The precondition ensures that a given copy is not already borrowed by a user. The Listing 2.3 shows a scenario which violates the precondition `copyNotBorrowed`. The copy `lotr_1` is already borrowed by the user `ada`. When trying to borrow it again (line 5), USE detects the violation of the precondition and opens the evaluation shell. A user can now examine the current state, as it is done in line 23 to query the current user that

borrowed the copy. The same feature is used, if a postcondition is violated. Therefore, we do not provide an example for this situation.

Query

While modeling a system, during animation, or validation, one often needs to be able to evaluate ad-hoc queries. For example, to identify invalid instances or just to test an OCL expression. For this, USE allows a modeler to enter OCL queries at any time using the query command on the shell or the evaluation view. A detailed evaluation command is also provided. It prints the complete evaluation tree of an expression. Using the graphical user interface, the already mentioned evaluation browser can be used to examine the sub-expressions. In contrast to the shell command, the evaluation browser provides additional interactive features to highlight and suppress different sub-expressions. This allows a modeler to identify relevant parts of an examined expression more easily. As an example for such ad-hoc queries consider line 23 in Listing 2.3 on the previous page, where the system state is queried for the current user who already borrowed the copy.

2.4 Runtime Verification

In this section, we show the overall goal of runtime verification and examine different approaches in this research area. Verifying a systems means to show that it conforms to its specification (*Do we build the product right?*) [7, p. 101f.]. This is in contrast to validation where it is checked that a system provides the intended functionality (*Do we build the right product?*) [7, p. 101f.]. In general, the latter question needs to be answered by experts of the domain the system is intended for together with software engineers. Verification relies on the, ideally validated, requirements defined for the product (its specification). The verification itself can be done in many different ways, e. g., by code reviews or using tests (c. f. [91]). Of high interest for practical usage are approaches that can be automated. On a first sight, the verification of a system by proving its correctness would lead to the highest certainty of correctness. However, such a verification is generally hard to achieve, because of its complexity [72, p. 901]. To reduce the complexity, several possibilities exist. For example, the system can be abstracted by using models of it that focus on the properties to verify. These models can then be analyzed by a verification tool either statically or dynamically.

Static verification techniques, like model checking [18, 31], try to verify requirements off-line by analyzing the execution paths of a program. An overview of the complete model checking process is shown in Fig. 2.8 on page 22. First, the design to be verified is modeled in an appropriate modeling language. The requirements, i. e., the specification, is formalized using temporal properties. A (bounded) model checker tries to traverse the reachable states and verifies the given temporal properties. If a property is not fulfilled,

a counterexample in form of a sequence of states is generated [8]. A lot of research in this area tries to reduce the state space that has to be examined, to overcome the so called "state space explosion" [40, p. 15][72, p. 902].

In contrast to model checking, the goal of runtime verification (also called runtime monitoring) is not to verify *all* possible execution paths, but to verify requirements while the program is executed [19], thus providing more rigor in testing [16]. A detailed comparison of model checking and runtime verification can be found in [50]. Using runtime verification as an integral part in the development process combines the specification and the implementation to together form the complete system. In [16] this is called *Monitor Oriented Programming* (MOP). By combining the specification and the implementation, see Fig. 2.9 on the next page, it is obvious, that a runtime verification approach is more coupled to the end product, since the more abstract specification must be aligned to the executed system in order to monitor it (see the directed edge between the system node and the Runtime Verification node in Fig. 2.9 on the following page). If the abstract specification would be inconsistent with the monitored part of the system, this would lead to an error during execution. This is in contrast to model checking, where the abstract model of the verified system is more decoupled from the concrete implementation, requiring complementary techniques to verify the correctness of an implementation [40, p. 15]. If a model checker finds a counter example, i. e., the specification does not hold, an additional simulation step is required to locate the error inside of the system, whereas during runtime verification the application can stop at the location where the error occurred. The main drawback of runtime verification, compared to model checking, is that the properties to verify must be executed either manually or by using specialized test drivers, whereas a model checker automatically evaluates all possible execution paths, leading to a higher degree of automation.

2.4.1 Specifying Properties

Essential to runtime verification are the properties to verify, since they define the specification of a system. To be able to evaluate these properties, a formal specification language that can be computed must be used. Most of these languages include elements to express temporal properties, i. e., relations between different (sequential) states of a program. This is in contrast to an invariant, which only needs access to a single state [70]. Examples of such formalisms, which are described next, are *Linear Temporal Logic* (LTL) [70] and *Extended Regular Expressions* (ERE) [46]. LTL introduces operators to express temporal relations between program states. For example, the operator **X** applied to a formula ϕ is used to express the fact that ϕ must hold in the ne**X**t sate. Similar to invariants is the operator **G**, which requires a property ϕ to be valid **G**lobally, i. e., in all states. Some authors further separate between future time linear logic (FTLTL) and past time linear temporal logic (PTLTL). The former states properties about the future execution of a system, whereas the latter goes back in time. In [53]

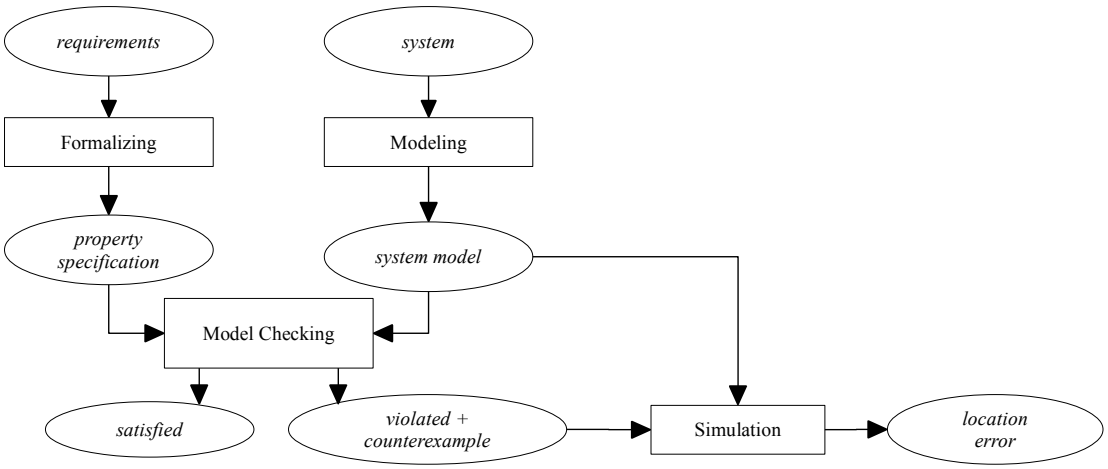


Figure 2.8: Overview of Tasks and Artifacts for Model Checking (taken from [40])

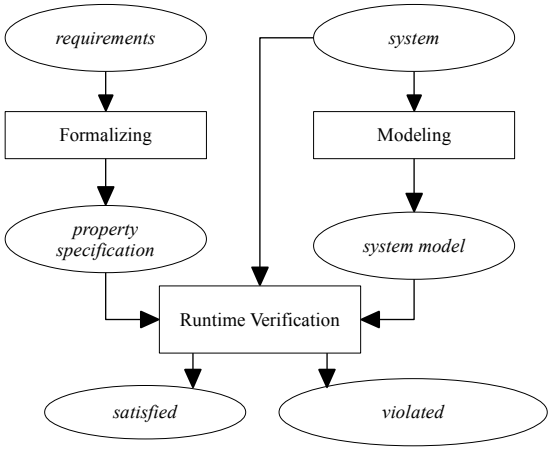


Figure 2.9: Overview of Tasks and Artifacts for Runtime Verification (based on [40])

it has been shown that PTLTL has the same expressiveness as FTLTL.

The high popularity of regular expressions inside the developer community, lead to the usage of *Extended Regular Expressions* for runtime verification. Since a program execution trace can be seen as a string of states, regular expressions fit well for defining properties on them. EREs extend regular expressions by adding operators for complement ($\neg$) and intersection ($\wedge$) to the three regular expression operators union ($\vee$), concatenation ($\cdot$), and repetition ($*$) [46]. These two additional operators, especially the complement operator, ease the specification of patterns for traces that are not allowed [80]. As an example consider the safety property "it should not be the case that in any trace of a traffic light we see green and then immediately red at any point" [80]. Following [80] this can be expressed in "the natural and intuitive way" by the ERE "$\neg((\neg\emptyset) \cdot \text{green} \cdot \text{red} \cdot (\neg\emptyset))$, where $\emptyset$ is the empty ERE (no words), so $\neg\emptyset$ means "anything"".

2.4.2 Classification of Runtime Verification Techniques

Runtime verification can be done using different techniques. For example, it can be done by directly reacting on changes inside the running system or it can be done "post mortem" by analyzing recorded execution traces. In the following, we provide an excerpt of the relevant categories for our work, as they are described in [76] and [16]. Common to all techniques is that they need to inject some kind of monitoring code inside the running application. Though, they have different consequences for the runtime behavior of the monitored application, e. g., the introduced overhead, and differ in the possibilities to react on encountered violations of the specification.

On-line and Off-line

On-line runtime verification approaches verify properties while the system that is verified is executed, whereas off-line approaches analyze the recorded traces "post mortem" after the system has finished execution.

If runtime verification is done on-line, the user gets the benefit of immediate feedback. The monitor can force the executed system to pause and can provide detailed information about the current state. The major drawback of on-line verification is the introduced overhead, since at relevant points during execution, the verification engine needs to evaluate the properties to validate. Depending on the size of the system and the specification, this can introduce noticeable delays.

To overcome possible delays in the execution of a system under test, off-line approaches minimize the introduced overhead by recording traces of the execution. Using an adequate logging technique, this leads to much shorter delays. These recorded execution traces are used after the system under test has executed to rebuild the program execution and to verify the specified properties. If a property is violated, only the trace

information is present to the user, which makes the reconstruction of the failing situation more complicated than using an on-line approach where the system under test stops in the failing state.

Note that sometimes, e. g., in [76] on-line approaches are called *synchronous* (the validation engine reacts on a received event and returns control to the monitored application after validation) and off-line ones *asynchronous* (the monitored system sends events and immediately continues execution afterwards).

Trace-Storing vs. non Storing

Obviously, two possibilities to work on execution traces exist. First, the complete trace can be stored to allow for a verification algorithm to compute on it. And second, a monitor can change the snapshot of the running system if it receives an event and discard this event after an incremental update of snapshot and required validation tasks have been done. The former approach is useful, if a validation algorithm has a reduced complexity when it has access to the complete trace. One example are algorithms to efficiently evaluate extended regular expressions [76]. The latter approach, which is not storing the complete trace but keeps a synchronized snapshot, reduces the overall consumption of memory and further allows to compute values without the need to loop through the trace.

Predictive vs. Exact Analysis

While the exact analysis of recorded traces can only detect concrete violations of properties, algorithms that try to predict possible anomalies by analyzing valid execution traces exist also. Use cases for them are, e. g., the prediction of possible deadlock or datarace situations [76]. In this thesis, we focus on exact algorithms, therefore we leave out a more detailed discussion.

3 Formalizing and Applying UML and OCL

In this chapter, work in the context of formally defining newer UML modeling constructs and its application inside of the USE tool is summarized. The main contributions are related to the runtime semantics of UML in combination with OCL, since little is stated about this semantics in the UML specification.

The first part of this chapter deals with structure modeling, e. g., constraints and extended modeling concepts in class diagrams. In [A18C], [A15C], and [A16C] we published work on how to define a more precise semantics for property relations defined in class diagrams. Instead of using a translation to another language with a well-defined semantics, the presented approach uses UML and OCL itself to define the semantics, i. e., the approach stays inside of the same technology space, which makes it unnecessary to introduce further languages. In addition, existing UML and OCL validation tools can be used to validate the semantics developed this way. While [A18C] discusses possible realizations in USE, [A15C] applies an approach based on [42] to define the runtime semantics of these constraints using UML and OCL. The work in [A12W] applies the property relations to connect multiple metamodels inside a single model, which allows for a uniform handling of all metamodel levels.

In the second part, work on UML Protocol State Machines (PSM[1]) as published in [A22C] is presented. It is shown, how this kind of state machines can strengthen the design of the behavior of a system. Further, the runtime behavior of PSMs is discussed. The main motivation of this work was to increase the possibilities of the runtime verification approach explained in Chap. 4.

3.1 Endogenous Metamodeling Semantics

This section introduces the basic concepts of our approach, which is described in the forthcoming sections. The overall idea of metamodeling semantics is to define the semantics of languages by using models. *Endogenous* metamodeling semantics further stays inside of the same technology space, i. e., the semantics of a language are defined by using itself instead of using another formalism, like it is done, e. g., in [74] for OCL. In [42], which is the base for our work, metamodeling semantics is defined as follows:

[1] Please note that PSM is also used as an acronym for Platform Specific Model as stated earlier. In the following we always use PSM as an acronym for Protocol State Machine.

> "Metamodeling semantics is a way to describe semantics that is similar to the way in which popular languages like UML are defined. In metamodeling semantics, not only the abstract syntax of the language, but also the semantic domain, is specified using a model." [42]

Metamodeling a language by defining the abstract syntax using a graphical modeling language combined with a formal textual language to express well-formedness rules is a well-known technique. The UML specification, for example, uses UML (or MOF, which itself uses UML) in combination with OCL to define its abstract syntax. In [42] this is called the *Abstract Syntax Model* (ASM), which defines the valid structures in a model. The same technique is rarely used to define the semantics of a language, i. e., to specify a *Semantic Domain Model* (SDM) of a modeling language. A *semantic domain* defines the meaning of a particular language feature, whereas a semantic domain model describes this meaning by modeling the runtime behavior of a (syntactically) valid model using its runtime values and applying meaning to them. For example, later we will see that in the UML there is the class `Class` in the abstract syntax part, and there is the class `InstancesSpecification` in the semantic domain part which together can describe (through an appropriate association) that a class (introduced at design time) is interpreted (at runtime) by a set of objects, formally captured as instance specifications. Another publicly available example for metamodeling semantics can be found in Section 10 of the OCL specification [63]. It defines constraints on values, i. e., runtime instances, which are part of the SDM. For example, the runtime value of a set is constrained as follows [63, p. 113]:

```
1 context SetTypeValue inv: self.element->isUnique(e : Element | e.value)
```

The central idea behind the approach in [63] is to describe the runtime behavior of OCL using OCL, which is similar to the UML metamodel described by UML models. While this is done in the UML to constrain the metamodel level M1, i. e., the valid structure of models, very little formal information is given for the level M0. Nearly only the structure for the runtime snapshots is specified, but little use is made of defining runtime constraints in a formal language like OCL. An excerpt of the UML metamodel which shows important elements for our work is shown in Fig. 3.1 on the next page. The diagram combines elements from roughly six syntax diagrams of the UML metamodel. On the left side, the ASM (syntax) of the UML is shown. On the right, the SDM (semantics) elements are given as they are present in the current specification. In the next sections we define runtime constraints on the semantic domain model for several modeling constructs which are frequently used in the definition of the UML metamodel, but are only defined in an informal way using verbal descriptions.

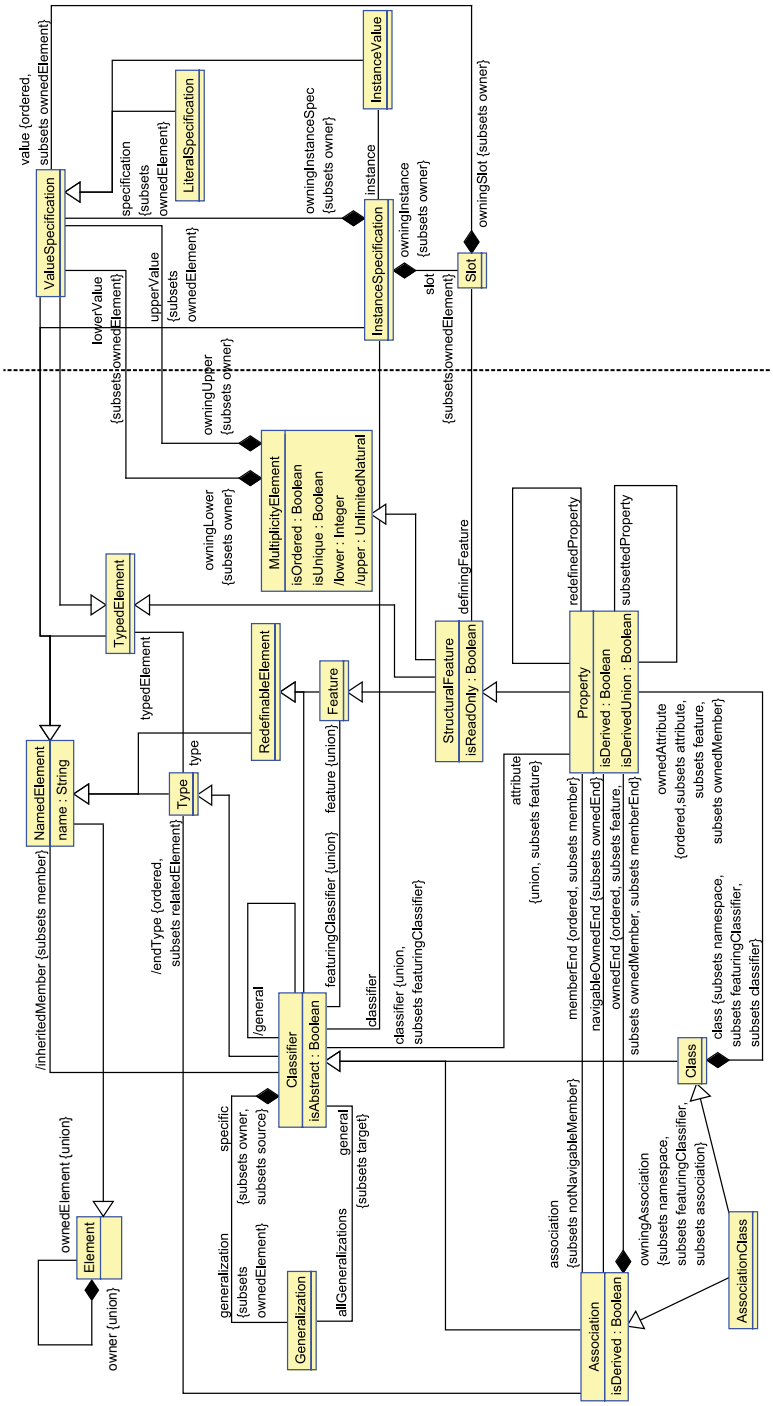


Figure 3.1: Combined View of UML Metamodel Elements[A15C]

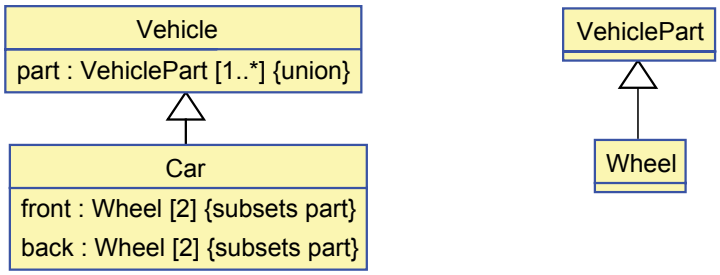


Figure 3.2: Class Diagram Using `subsets` and `union` on Attributes [A15C]

3.2 Static Structure Modeling

In the next sections we show, how the previously explained approach can be used to define a precise meaning of informally defined elements of the UML specification and how this definition can be automatically validated by using instances of the specified ASM and SDM in the USE tool. In particular, we are going to define the runtime semantics of property relations by using UML and OCL. For this, we first introduce these relations by an example. Afterwards, we use this example to demonstrate our approach.

Like classes, associations in the UML can be related by specifying an inheritance relation between them. This is expressed in the UML metamodel by the class `Generalization` connecting classifiers. Since the metaclass `Association` is derived from the metaclass `Classifier`, generalizations can be defined for them. Another kind of relation for association ends and also for properties not linked to an association, i. e., attributes, are so-called property relations. The UML metamodel defines two kinds of property relations [69, p. 37]:

1. subsets and union (extensional semantics)
2. redefines (intentional semantics)

In the area of database systems, the distinction between intentional and extensional semantics is sometimes made by defining the schema of a database as the intentional layer and the state of a database as the extensional layer [41]. In the following, we concentrate on the definition of `subsets` and `union`. The usage of `redefines` was discussed in [A18C].

3.2.1 Subsetting and Derived Unions

We explain our proposal by starting with a basic class diagram, which uses subsetting and union constraints on attributes of classes. Later on, we extend this diagram by using

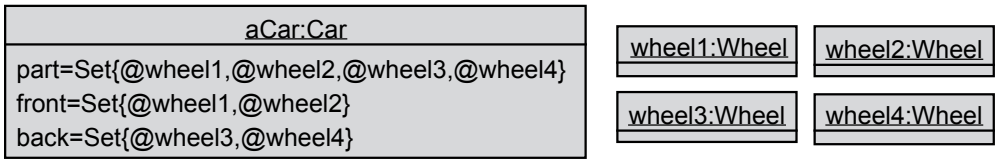


Figure 3.3: Valid Object Diagram Using `subsets` and `union` on Attributes [A15C]

Listing 3.1: Property Constraints Translated to Invariants

```
1  context Vehicle inv partIsUnion:
2   let selfCar = self.oclAsType(Car) in
3    selfCar <> null implies
4     self.part = selfCar.front->union(selfCar.back)
5
6  context Car inv frontIsSubset:
7   self.part->includesAll(self.front)
8
9  context Car inv backIsSubset:
10  self.part->includesAll(self.back)
```

subsetting and union on associations. Subsetting and union constraints on properties (a property can take the role of an attribute or an association end) define a relation between two or more properties. The values of a subsetting property must be a subset of the values for the subsetted property. Union can be used on a single property. Its usage defines that the values of a property are the union of all its subsetting properties.

Figure 3.2 on the facing page shows a simple model of vehicles (c. f. [A15C] and [9]). A vehicle consists of vehicle parts. For a car, information about the front and back wheels is added to the class `Car`. Because these wheels are part of the overall vehicle, the properties `front` and `back` are marked as subsets of the general property `part`. The property `part` itself is marked as a derived `union` of all of its subsets. Furthermore, the subsetting properties restrict the lower and upper bounds of the wheels to the common number of wheels for a car (2 is equivalent to 2..2). A valid object diagram w. r. t. the given class diagram is shown in Fig. 3.3. For this simple diagram, one can see directly that the intended constraints are fulfilled. However, for more complicated models, an automatic validation is required. If the used modeling language would not provide `subsets` and `union` constraints, a modeler could still specify invariants on the classes `Vehicle` and `Car` as shown in Listing 3.1. However, these constraints would strongly couple the abstract class `Vehicle` and its subclass `Car`, because `Vehicle` needs information about its subclasses to validate the union constraint. This would break well-known design guidelines, e. g., [21, p. 94], because it leads to reduced reusability of

Listing 3.2: Invariant For the Runtime Semantics of Subsets

```
context Slot inv subsettingIsValid:
 let prop = self.definingFeature.oclAsType(Property) in
  (prop <> null and prop.owner.oclIsKindOf(Class)) implies
   prop.subsettedProperty->forAll(subsettedProp |
    let subsettedValues = self.owningInstance.slot->
     any(definingFeature=subsettedProp).value.getValue()->asSet() in
      let currentValues = self.value.getValue()->asSet() in
        subsettedValues->includesAll(currentValues))
```

the abstract class `Vehicle`.

To allow a generic usage of these constraints the UML provides the ability to specify subset relations between properties using a reflexive association on `Property` and to mark a property as a derived union (see Fig. 3.1 on page 27). Further, several well-formedness OCL rules are given, to ensure the syntactical correctness of the usage. For example, the type of the subsetting property must conform to the type of the subsetted end [69, p. 126]. However, information about the semantics of the UML language element `subsets` is only provided textually, not in a formal way. We propose to add (what we call) runtime semantics by means of OCL constraints to the already present elements describing runtime elements. For the previously explained example, which uses property constraints on attributes, the constraint shown in Listing 3.2 specified on the UML metaclass `Slot` (a slot allows, for example, to assign an attribute value to an attribute) describes the runtime semantics of subsets. The constraint checks for each slot that defines a value or values for an attribute of a class, if it is a subset of the values defined by the slots of the subsetted properties. Because this constraint only considers attributes of classes, the navigation to the slots of the owning instance of the context slot is enough. For associations, and especially for associations with more than two ends, the calculation of the values to be considered is more complicated.

A class diagram which makes use of `subsets` and `union` on association ends is given in Fig. 3.4 on the next page. The previously specified attributes `part` and `front` are changed to association ends, while the attribute `back` is left out in order to keep the following examples at a moderate size. Figure 3.5 on the facing page shows an example instantiation of the class diagram. The links shown as a solid line are inserted by the user, while the dashed links are automatically calculated by USE, because they are part of a derived union. In USE, all derived links (either established through a derived union or through an explicit derived association end) are shown as dashed links to be able to separate them from concrete, i. e., non calculated, links.

The object diagram in Fig. 3.6 on page 32 shows an instantiation of the UML metamodel representing the class diagram of Fig. 3.4 on the facing page at the top and the object diagram shown in Fig. 3.5 on the next page at the bottom. This figure inten-

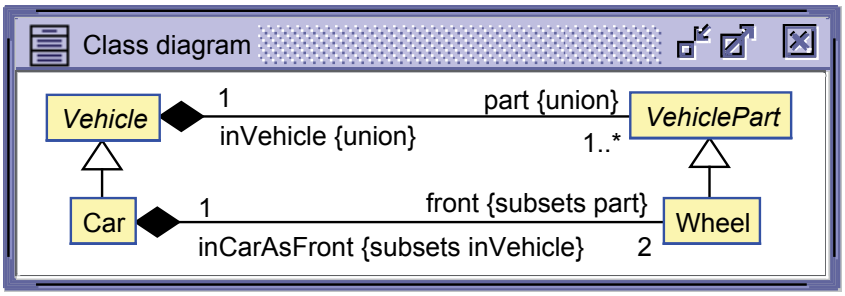


Figure 3.4: Class Diagram Using `subsets` and `union` on Association Ends [A15C]

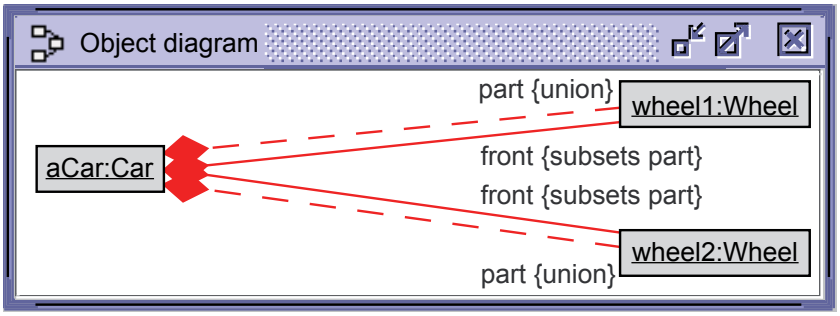


Figure 3.5: Object Diagram Using `subsets` and `union` on Association Ends [A15C]

tionally includes so many dashed lines and compositions, in order to show the inherent complexity of the UML metamodel. This complexity can automatically be revealed by using our tool. In Sect. 3.2.2 on page 34 we are going to explain these so-called virtual links in more detail. On the other side, these virtual links allow us to suppress certain elements in the object diagram to make it easier to be read. For example, the generalization relationships are only shown as derived links between the classes leaving out the generalization instance. To be more concrete, in the left upper part of Fig. 3.6 on the next page the dashed link between `Class3` (Vehicle) and `Class4` (Car) corresponds to the left generalization arrow in Fig. 3.4. We use this diagram in the following to explain an extended runtime semantics which also covers associations.

A runtime semantics for subsetting that covers attributes and association ends must consider all tuples of instances which are linked to a subsetted property and the set of instances linked to this tuple at the subsetting end. For the previously shown example on attributes, this tuple contains only one element, namely the defining instance, whereas for association ends of an association with n ends, this tuple contains $n - 1$ elements. We accomplish this by using a query operation called `getConnectedObjects()` which is similar to the operation `Extent::linkedObjects(...)` defined in the MOF specification [62], but also covers n-ary associations, properties, and derived unions. The query

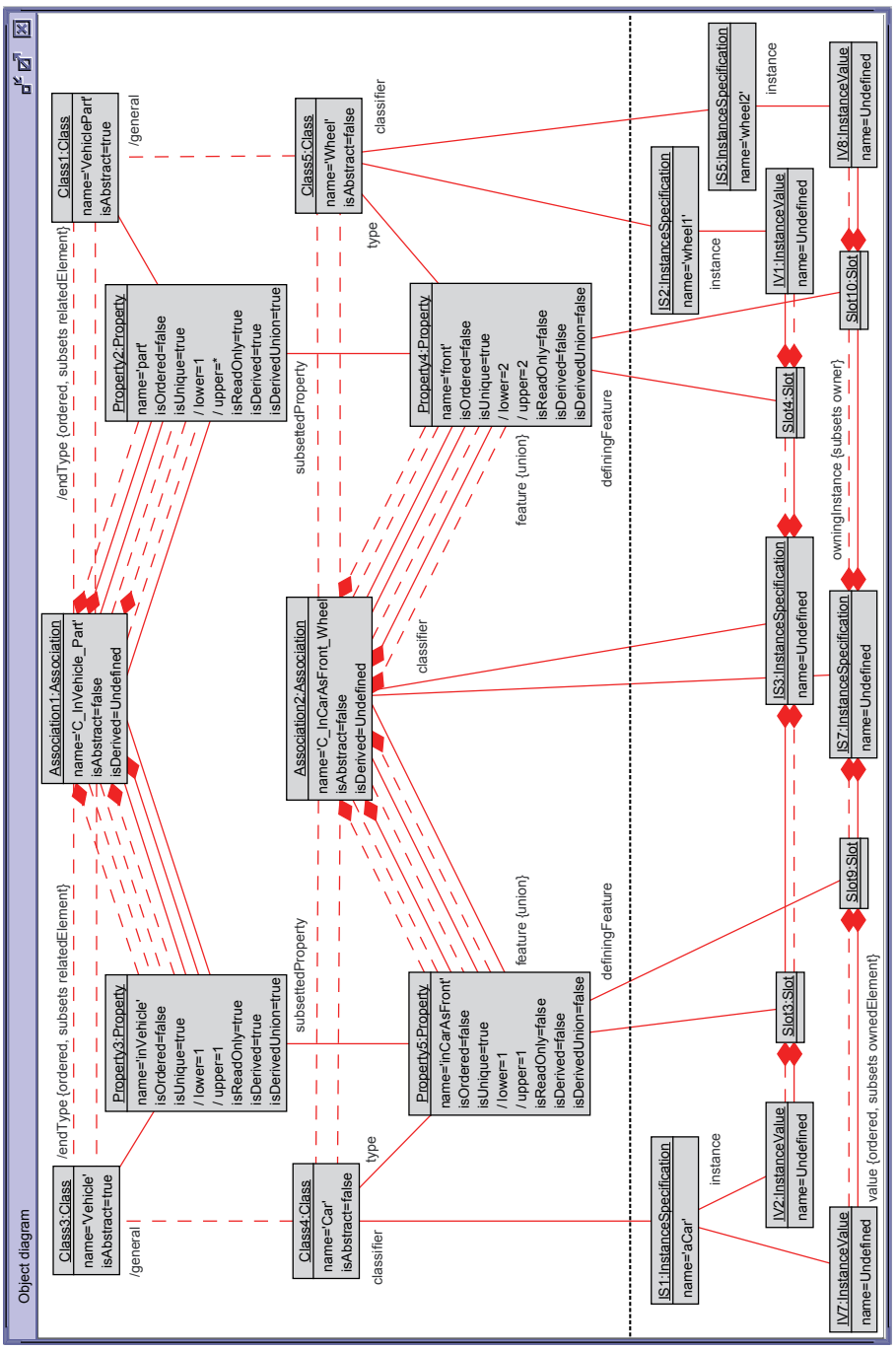


Figure 3.6: Class and Object Diagram as a Metamodel Instance [A15C]

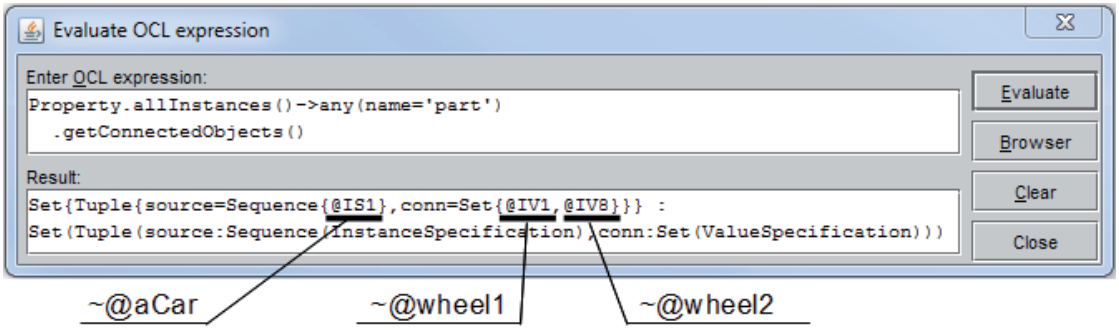


Figure 3.7: Querying Runtime Values Using `getConnectedObjects()` [A15C]

operation uses the metaclasses of the semantic domain model to obtain all connections specified for a property. For this, it navigates to all instance specifications to consider and their owned slots. If a property is defined as a derived union, this operation is recursively invoked on all properties subsetting the derived union property and collects all connected values in a single set, i. e., it builds the union of the values. To give a more detailed view of the usage of this central operation, Fig. 3.7 shows the result of invoking it on the property `part` using the state shown in Fig. 3.6 on the facing page.

The result is a set of tuples with two parts:

1. **source**: The sequence of source objects in the same order as the association ends, if the property is owned by an association. If it is used as an attribute, a sequence including the context object, only.

2. **conn**: The objects connected to the source objects at the property.

The result of the evaluation is the calculated union of the property values for all possible source objects. Because only one vehicle (named `aCar`), is present in the given state, the set contains a single tuple. This tuple consists of the sequence containing the instance specification representing the object `aCar` and a set of values which are linked to this instance via subsetting properties of `part`. Given the previously described operation `getConnectedObjects()`, we can define a constraint which ensures the subsetting semantics:

```
1 context Property inv subsettingIsValid:
2  let subsetLinks = self.getConnectedObjects() in
3   self.subsettedProperty->forAll(supersetProperty |
4    let supersetLinks = supersetProperty.getConnectedObjects() in
5     subsetLinks->forAll(t1 |
6      supersetLinks->one(t2 | t1.source=t2.source and
7       t2.conn.getValue()->asSet()->includesAll(
8        t1.conn.getValue()->asSet())))))
```

The central part of the given invariant can be seen on line 7 where the operation `includesAll` is used, which is the OCL way to validate whether a collection is a superset of another one. Some things need to be explained in more detail. First, the usage of the operation `getValue():OclAny`, which is an extension to the UML metaclass `ValueSpecification`, is required to be able to get the concrete value of a value specification. The UML metamodel defines several operations on this class for retrieving basic types like `stringValue():String` but excludes a generic definition. Second, the collected values need to be converted to a set using `asSet()` (see lines 7 and 8) because values can map to the same specifications. It should be mentioned that, if evaluated at runtime, the invariant only validates the union calculation if subsets is used in the context of a derived union. If subsets is used on a property which is not a derived union, the constraint validates the user defined structure.

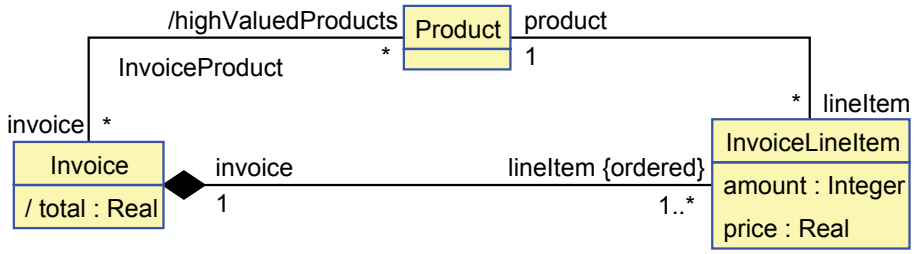
Including the described invariant and similar invariants for other runtime elements adds a precise definition of its semantics to the modeling language. Further, our approach allows for an automatic validation of the defined semantics by using existing tools, because no new UML and OCL features are required for the invariant. For example, instantiations of the metamodel inside the USE tool can be checked or the USE model validator can be applied to verify the semantics.

3.2.2 Derived Properties

As already shown, properties can be marked as derived. First of all, this expresses that these properties are a combination of data already present in a model. If information about how to extract the derived information is present, the name of the derived property further assigns meaning to this combination. By providing a derive expression that can be automatically computed at runtime, derived properties can help to examine runtime phenomena, since they can visualize the relations of the model elements combined by the derive expression. Besides the easier retrieval of information, derived properties can also be used to leave out too detailed information by shortening the navigation to connected objects. In the following we are going to explain the integration and the usage of derived expressions for the two common types of properties in USE: attributes and association ends.

Derived Attributes

Derived attributes are visualized using nearly the same syntax as normal attributes, except that a leading slash (`/`) before the attribute name indicates that this attribute is derived. For an attribute, a derive expression can be defined using OCL. The type of such an expression must conform to the defined type of the attribute. Its evaluation context is given by the instance the attribute value is calculated for and can be accessed by the keyword `self`. Figure 3.8 on the next page shows an example of a derived




```
1 context Invoice::total:Real derive:
2   self.lineItem->collect(amount * price)->sum()
3
4 context Invoice::highValuedProducts derive:
5   self.lineItem->select(item| item.price > 200).product->asSet()
```

Figure 3.8: Class Diagram Including Derived Properties and Their Definitions

attribute named `total` which is owned by the class `Invoice`. The derive expression, that computes the total of an invoice by summing up the products of the amounts and the price, is shown below the figure. This attribute can be used like any other attribute inside of an OCL expression, but moreover, the total amount of an invoice can directly be seen in an object diagram as shown in Fig. 3.9. This is different to the usage of query operations, which cannot be directly displayed in an object diagram, since operations, in contrast to derived attributes, can define parameters and there is no possibility inside an object diagram to provide arguments for these parameters.

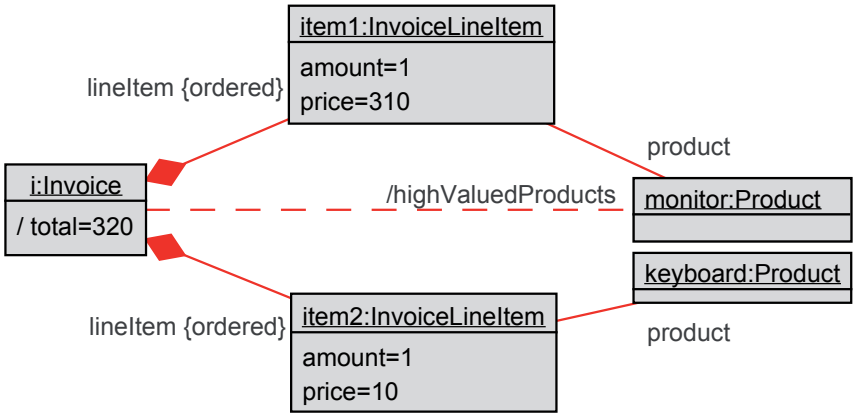


Figure 3.9: Object Diagram Including Derived Properties

Derived Association Ends

The usage of derived association ends can also help to improve views on a system state. Like normal links, derived links are a natural way to show connections between objects. A user can easily identify connected objects without the need to look up object identifiers or to enter a query.

To make these associations visible, they need to be evaluated. In the UML specification, derived associations, as with derived attributes, are defined by means of constraints, i. e., boolean expressions, which define them in a declarative way. This approach precisely describes the content (the links) of a derived association, but cannot be easily computed if a model is evaluated, because the information about the derived part of the boolean expression would need to be extracted. To provide a usable way, our approach requires the use of expressions that directly evaluate to the expected result. For example, the derive expression for the association end `/highValuedProducts` shown in Fig. 3.8 on the previous page on the lines 4 and 5 evaluates to a set of `Products`, because the association end has a multiplicity of more than one. To be able to evaluate a derive expression an evaluation context is needed. As previously described, the context of a derive expression for an attribute is the instance on which it is evaluated. This also holds for binary associations if the result of a navigation to a derived end needs to be calculated, since the instance from which the navigation starts is the context for it (an instance of type `Invoice` in the previously used example).

However, there are situations for which such a context instance is not given. If the result of a navigation expression starting at a derived association end must be computed, there is no implicit context instance available. Instead, the derive expression must be evaluated for all instances which could be connected to the starting instance, i. e., for all instances of the opposite association end. If the result of the evaluation contains the starting instance, the instance for which the expression was evaluated is linked to the starting instance.

As an example, consider Fig. 3.8 on the preceding page and the navigation from a `Product` to an `Invoice` using the association `InvoiceProduct` that starts at the derived association end `/highValuedProducts`. To get all connected invoices, the derive expression needs to be evaluated for all instances of `Invoice`. If the result contains the source product, the invoice is part of the resulting set of the navigation expression. This informal description can be defined using OCL as follows:

```
1 context Product::invoice derive:
2   Invoice.allInstances()->select(i|i.highValuedProduct->contains(self))
```

Since a navigation, if started from a derived association end, always follows the pattern shown in the previous example, it can be integrated into an evaluator. Another situation, where no implicit context is available, occurs when using derived associations with more than two association ends. To be able to calculate the result, even if evaluated on a given

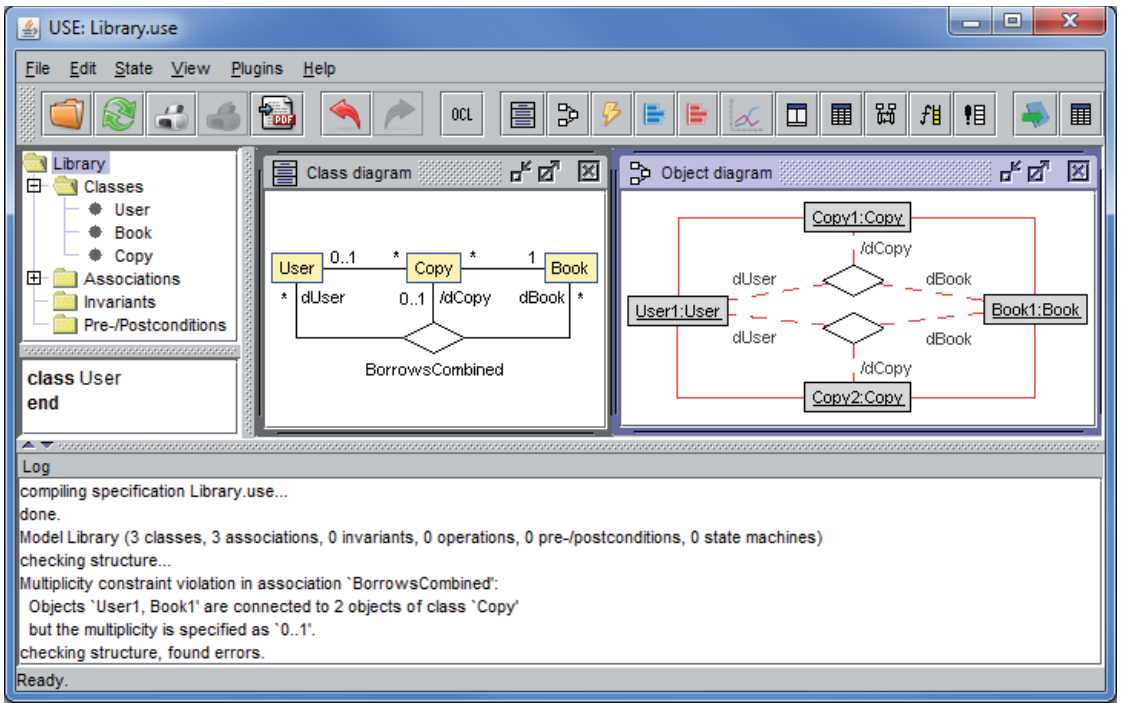


Figure 3.10: Using a Derived Ternary Association to Express Further Rules [A15C]

context instance, all n-tuple (pairs for ternary associations) of possible combinations must be evaluated. In [A15C] we have shown, how n-ary derived associations can be used to visually express additional model constraints. An example using an extended model of the library system introduced earlier is shown in Fig. 3.10. The additional derived association between the three classes together with the defined multiplicity `0..1` for the association end `dCopy` are used to express the constraint, that one user can only borrow a single copy of a book. The definition including the derived expression for the new association is as follows:

```
1 association BorrowsCombined between
2   User[0..*] role dUser
3   Copy[0..1] role dCopy derive(aUser:User,aBook:Book) =
4     aUser.copy->select(c | c.book=aBook)
5   Book[0..*] role dBook
6 end
```

It uses a pair of copy and book objects to calculate the borrowed copies of the given user (`aUser`) and the given book (`aBook`). As it can bee seen in the log at the lower part of Fig. 3.10, the shown system state violates this multiplicity constraint on the derived association end, because the user `User1` has borrowed two copies of the book `Book1`.

Further, one can directly see the instances related to a single lending by looking at the ternary links.

3.3 Behavior Modeling with Protocol State Machines

This section summarizes work on modeling dynamic behavior in a declarative way using UML state machines, which has been published in [A22C] and [A16C]. The benefits of using state machines to define behavior on the static level together with the validation capabilities of USE in this context on the runtime level are shown using an example.

Our approach is similar to Executable UML [54], which is designed to specify a system at a high level of abstraction, independent from specific programming languages and decisions about the implementation. This follows the ideas of the Shlaer-Mellor methodology, which separated concerns about the structure [83] and the behavior [82] of a system to be developed. Executable UML is defined as a profile of the UML [69]. Executable UML models are testable, and can be compiled into less abstract programming languages to target a specific implementation. Executable UML supports model-driven development through specification of platform-independent models.

When using Executable UML, a system is decomposed into multiple modeling sublanguages: A class diagram defines the system structure in terms of the classes and associations; a state machine defines the states, events, and state transitions for a class instance; an action language defines the actions or operations that perform processing on model elements; the system behavior is determined by the state machines and the operations realized in the action language.

The following sections explain our support for state machines in order to complete the description of behavior. Within USE, we integrate class diagram validation with UML protocol state machine validation on the basis of OCL state invariants and OCL guards and postconditions for transitions. In contrast to Executable UML, our approach extends OCL in order to express actions and operation implementations, but does not need to define a separate action language.

3.3.1 Design Time

Based on the exemplary system of a library that we used in Chap. 2 to describe UML and our modeling tool USE, we are going to show the additional modeling capabilities introduced by adding support for state machines into USE. For this, please recap the class diagram of the library system shown in Fig. 2.3 on page 8 consisting of the three classes `User`, `Copy`, and `Book`.

We extend this model by adding behavioral system requirements. Some of them are shown in Fig. 3.11 on the facing page and 3.12 on page 40 as UML protocol state machines with states and transitions. For the class `Copy` the valid object lifecycles are depicted, which restrict the order of creation events and operation calls. Note, that

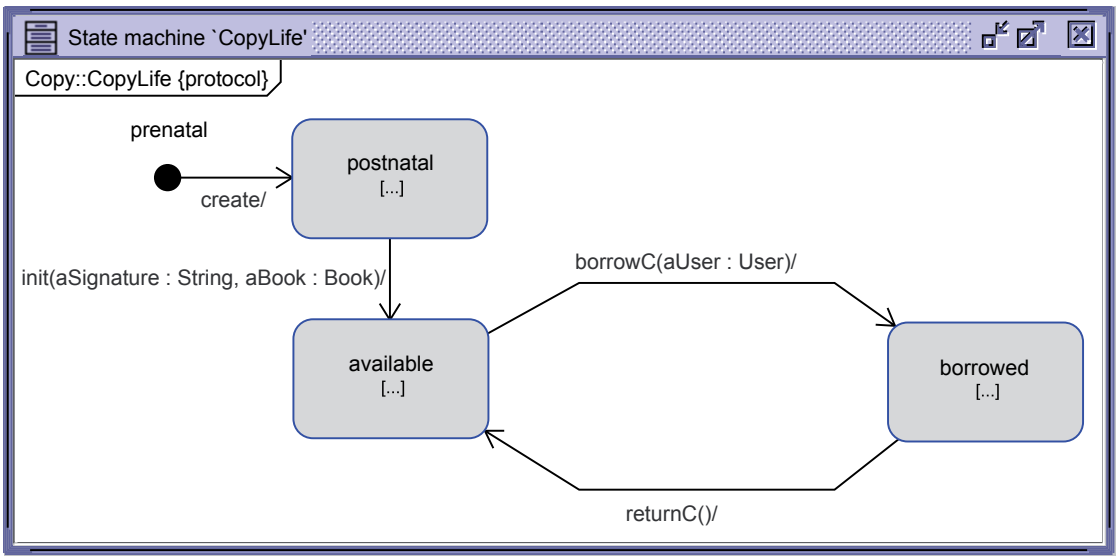


Figure 3.11: Requirements for the Behavior of the Class `Copy`

this figure is a modified version of the state machine presented in [A15C]. For example, the used arrows for transitions are now correctly aligned to the UML syntax and it is highlighted by using `[...]` that state invariants are suppressed.

As a central means to make the model precise, OCL is used in various places: States are described by state names and state invariants in form of boolean OCL expressions; transitions include: (a) the triggering create or call event, (b) a guard in form of a boolean OCL expression asserting that the transition only takes places when the guard holds, and (c) a postcondition in the form of a boolean OCL expression asserting that the transition only takes place in the case that after the transition the postcondition holds. Traditionally, the notion *guard* is used in connection with state machines; however, because of the symmetric behavior of the guard and postcondition, the guard may also be called transition precondition.

The state invariants may optionally be shown in the protocol state machine diagrams, however, we have suppressed them in the diagram to maintain readability. For example, for the class `Copy`, the three proper, non-pseudo states embody the state invariants given in Listing 3.3 on the following page. In the state `postnatal` (after `create`), all attributes must be undefined and the copy must not be linked to any book. In the state `available` (after a call to the initialization operation `init`), all attributes are defined and the copy is linked to a book.

The transitions are either labeled with the `create` event which brings the respective object into life or with an event which calls an operation of the object. The protocol state machine for the class `Copy` (Fig. 3.11) asserts a finite life-cycle demanding that after

Listing 3.3: State Invariants of the Class `Copy`

```
1  postnatal [self.signature.oclIsUndefined()
2             and self.numReturns.oclIsUndefined()
3             and self.user->isEmpty() and self.book->isEmpty()]
4
5  available [not self.signature.oclIsUndefined()
6             and not self.numReturns.oclIsUndefined()
7             and self.user->isEmpty()]
8
9  borrowed  [not self.signature.oclIsUndefined()
10            and not self.numReturns.oclIsUndefined()
11            and self.user->notEmpty()]
```

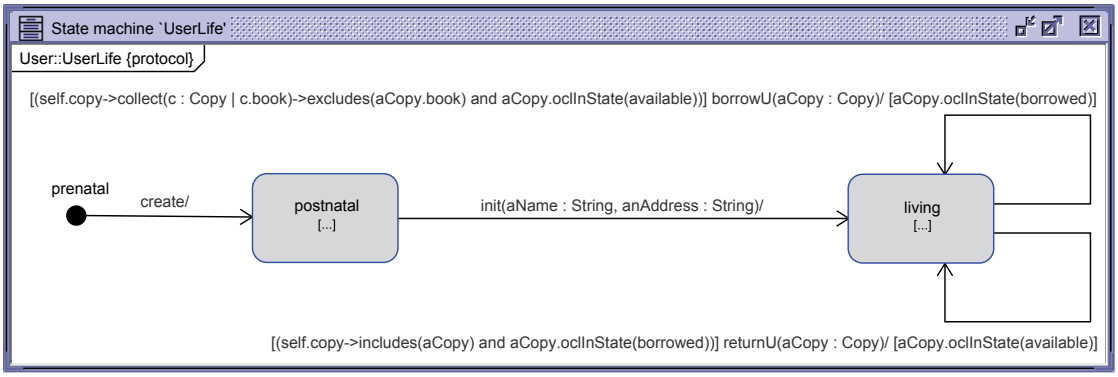


Figure 3.12: Requirements for the Behavior of the Class `User`

object creation only the operation `init` may be called once. It further guarantees that after creation and initialization, the `borrowC` and `returnC` operations switch between the states `available` and `borrowed`.

The state machine for the class `User` shown in Fig. 3.12 additionally employs OCL for transition guards and postconditions. Please be aware of the fact that all states are accompanied by OCL state invariants. Both operations, `borrowU` and `returnU` in class `User` are allowed in state `living`, however, OCL restrictions via transition guards and postconditions apply. The guard (precondition) for `borrowU` guarantees that a user cannot borrow two copies of the same book. And the guard asserts that only available (i. e., not borrowed) copies can be handled with the operation `borrowU`. The postcondition of `borrowU` checks that the copy, which was available before the transition took place, is now unavailable. Conversely, the guard for `returnU` asserts that the copy to be returned belongs to the current user and is indeed a copy in state `borrowed`. The postcondition checks that the parameter copy is indeed `available` after the `returnU` call. Note that these simple example restrictions do not guarantee unproblematic behavior

in all possible implementations. The state invariants, guards, and postconditions have been chosen for demonstration purposes.

An implementation on the modeling level of the operations can be realized in the language SOIL (see Sect. 2.3.1 on page 17). As an example, we show implementations for the operations of the class `User` in Listing 3.4 on the following page. These operation implementations allow the developer to build up simple or complex test states and scenarios with call sequences easily. Consequently, model properties like consistency or the reachability of protocol states can be checked with scenarios constructed with SOIL statements. The SOIL command sequence in the upper right side of Fig. 3.13 on page 43 is an example for such a test scenario. The validity of model properties formulated in OCL as class invariants, operation pre- and postconditions, state invariants, and transition pre- and postconditions is checked against these scenarios and by this also against the SOIL implementation given for the operations. When writing down a particular test scenario, the developer will have expectations on particular (class or state) invariants and (operation and transition) pre- and postconditions. These informal expectations are formally checked by the tool USE, and the validation results give detailed feedback to the developer about the possible discrepancy between their expectations and the actual facts: *"What you write down doesn't mean exactly what you think it means. And when it does, it doesn't have the consequences you expected."* [34, p. XIII]

3.3.2 Runtime

This section will explain how to apply the proposed concepts for the example at runtime. Whereas the previously shown figures pictured structure and behavior of the library system on a type level (design time), Fig. 3.13 on page 43 displays structure and behavior of one system test scenario on the instance level (runtime). The object diagram in the lower right represents the objects, their attribute values, and links after the SOIL command sequence in the upper right part of the same figure has been executed. In the left of the figure, the upper two state machine instances show the current protocol state for the `Copy` objects `dbs42` and `dbs52`. Also in the left, the lower two state machine instances display the current protocol state for the `User` objects `ada` and `bob` in dark grey. Please note that the state of both `Copy` objects and the state of both `User` objects are different. The state sequence which the `Copy` object `dbs52` went through was `postnatal`, `available`, `borrowed` and again `available`. We can conclude this from the executed operation sequence and from the attribute value `1` for attribute `numReturns`. In the shown operation sequence, all OCL restrictions have been checked and no violation occurs: all class invariants, state invariants and transition pre- and postconditions have been evaluated to `true`.

This scenario can be extended by further operation calls. For example, the `User` object `ada` could try to borrow the `Copy` object `dbs43`. In this situation, the guard for the `borrowU` call on the transition from `living` to `living` would prevent the transition

Listing 3.4: Library Example in USE employing PSMs and SOIL

```
class User -- pre- and postconditions not shown
 operations
  init(aName:String,anAddress:String)
   begin self.name := aName;
   self.address := anAddress; end

  borrowU(aCopy:Copy)
   begin aCopy.borrowC(self); end

  returnU(aCopy:Copy)
   begin aCopy.returnC(); end

 statemachines
  psm UserLife
   states
    prenatal:initial

    postnatal
    -- state invariant
    [name.oclIsUndefined() and address.oclIsUndefined()
     and copy->isEmpty()]

    living
    [not name.oclIsUndefined() and not address.oclIsUndefined()]

    transitions
    prenatal  -> postnatal { create }

    postnatal -> living    { init()
      [not name.oclIsUndefined() and not address.oclIsUndefined()
       and copy->isEmpty()]
    }

    living    -> living    {
      -- guard
      [self.copy->collect(c|c.book)->excludes(aCopy.book)
       and aCopy.oclInState(available)]
      -- trigger
      borrowU()
      -- postcondition
      [aCopy.oclInState(borrowed)]
    }

    living    -> living    {
      [self.copy->includes(aCopy) and aCopy.oclInState(borrowed)]
      returnU()
      [aCopy.oclInState(available)]
    }
end
```

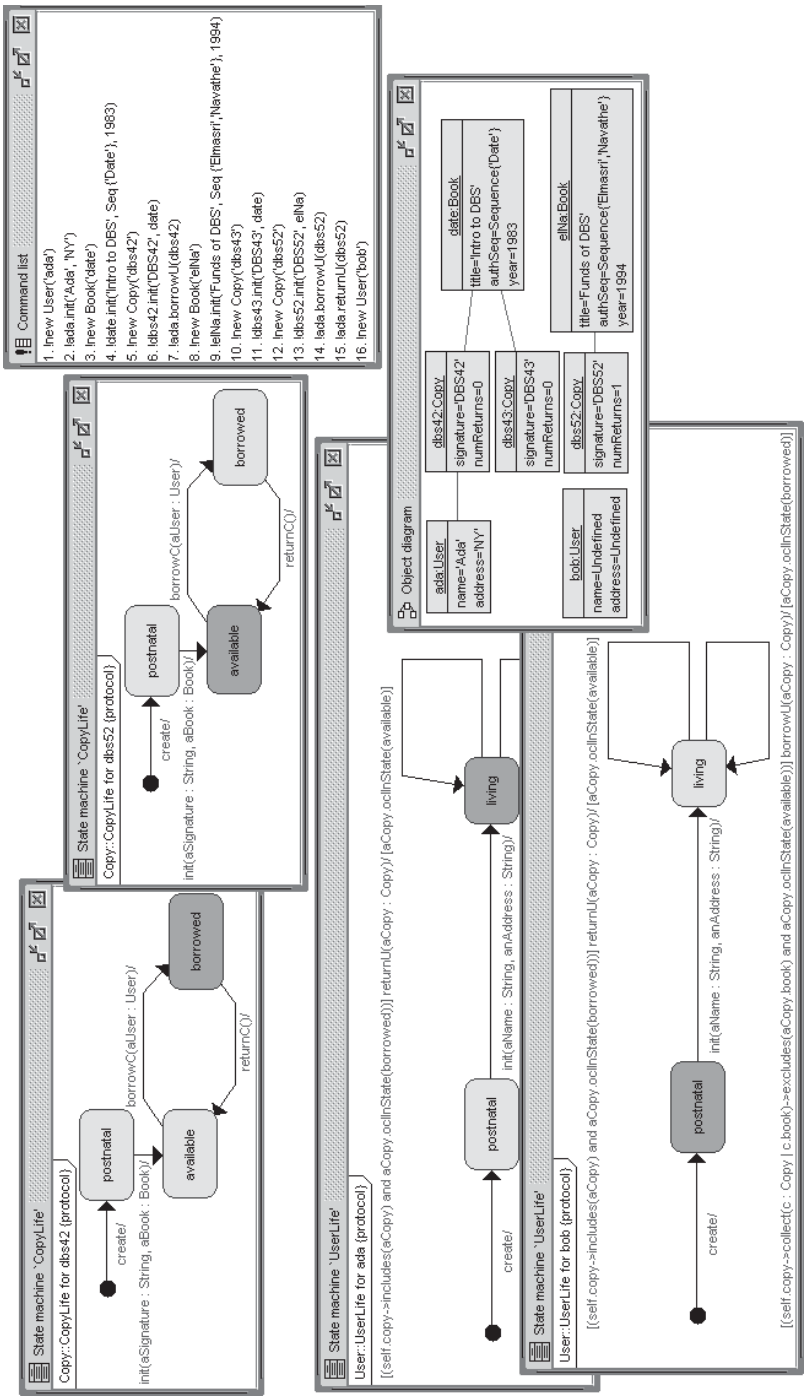


Figure 3.13: Example Scenario for Structure and Behavior (Runtime) [A22C]

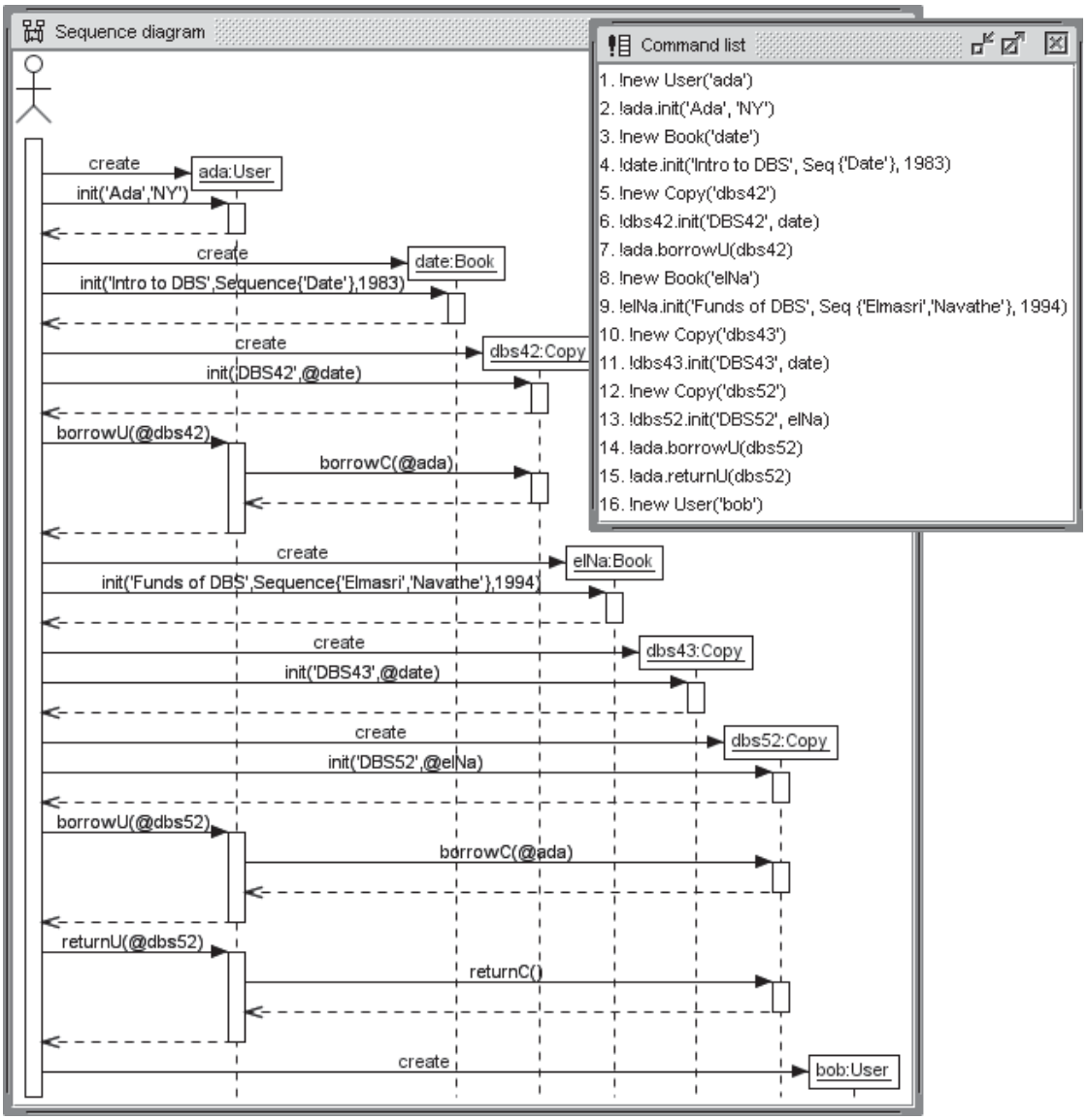


Figure 3.14: Sequence Diagram of the Example Scenario [A22C]

Listing 3.5: The USE Information About a Failing Guard

```
1 !ada.borrowU(dbs43)
2 >> Error: No valid transition available in protocol state machine
3 >> 'User::UserLife [current state: living]' for operation call
4 >> User::ada.borrowU(dbs43) due to failing transition guard.
```

to take place: User `ada` has already borrowed another copy of the `Book` object `date`. On the USE shell, a message will inform about the violation and the fact that the transition is invalid. The message given in Listing 3.5 will be shown. Analogous error messages would be displayed on the shell, if the transition postcondition or the state invariant of the next state would be violated.

Summarizing we can say that taking a transition may be aborted due to four possible reasons:

1. a failing transition guard (precondition),
2. a failing transition postcondition,
3. a failing state invariant in the resulting state, and
4. non-deterministic transitions, e. g., multiple transitions for the same trigger.

3.3.3 State Determination

One outstanding feature of our approach is the possibility to determine the current state of state machine instances by using the invariants of the states. This feature is useful, if a system state was built without following the execution trace of operation calls, e. g., if a system state was created by the USE model validator. Since the model validator tries to find a system state that is valid w. r. t. structural constraints, no information about behavior is available and therefore the current states of defined state machines must be determined. Another application of this feature is shown in Sec. 4.3 beginning on page 54 during runtime verification with USE.

In Fig. 3.15 on the following page, an example explains the usage of state invariants and the state determination option. For a `TrafficLight` class with three boolean attributes representing the red, yellow, and green bulbs, a protocol state machine allows the traffic light to step through four phases, where each phase is represented by a single state and a state invariant in the form of an OCL expression, characterizing the signal in terms of the bulbs.[2] The object diagram in the top center of the graphic shows four test traffic lights equipped with randomly determined attribute values for the bulbs, not all representing

[2]The phases are the phases used in Germany, whereas in other countries, e. g., in Italy, the phases are different.

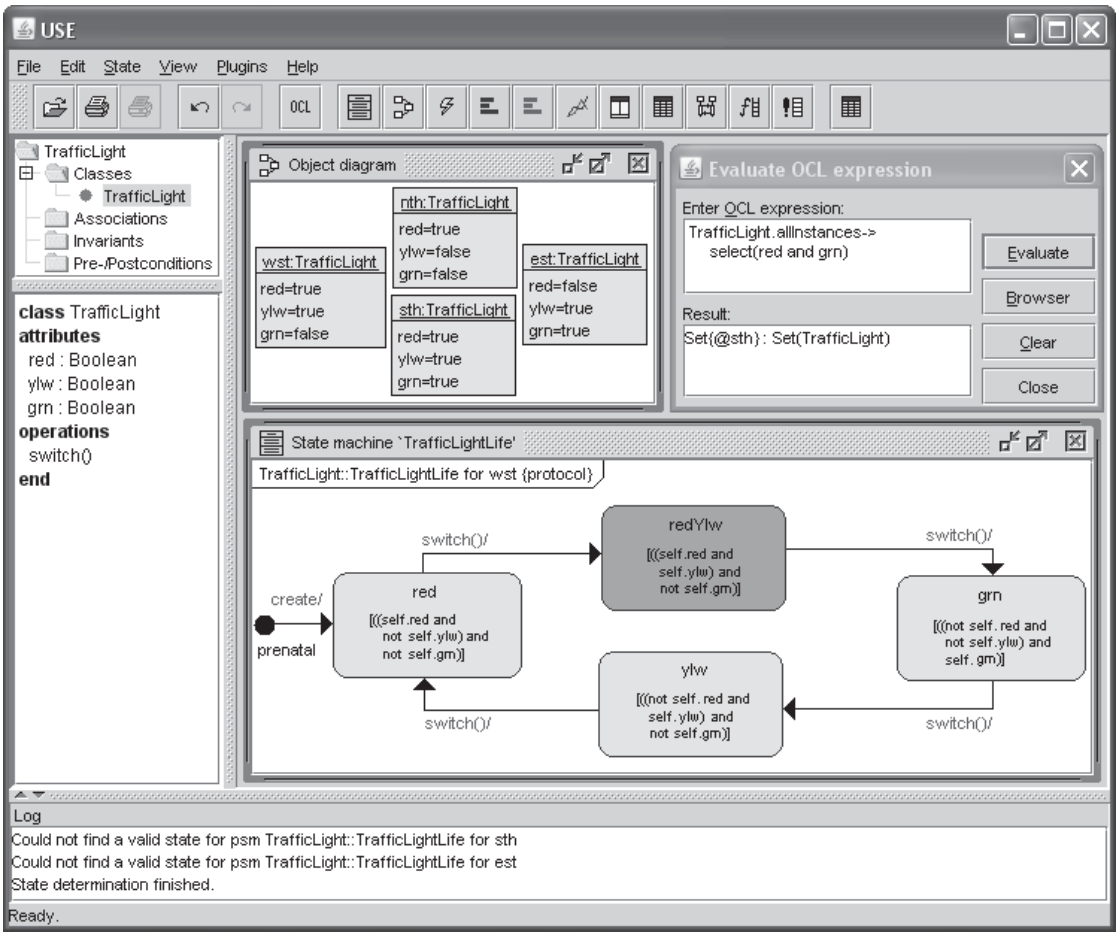


Figure 3.15: Example for the Usage of the State Determination Option

valid signal configurations. The attribute values have been modified not by operations, but with direct attribute assignments.

In the log window at the bottom, the result of executing the state determination command is given. This command aims to bring the state machine instances into the state corresponding to their state invariants, if possible. For two traffic lights (`sth` and `est`), a valid state fitting one of the four state invariants could not be found; the other state machine instances are moved into a state determined by a state invariant. The displayed state machine instance in the middle belongs to the `TrafficLight` object `wst` and shows that the attribute values (`wst.red=true` and `wst.ylw=true` and `wst.grn=false`) fit to the OCL state invariant expression (`self.red and self.ylw and not(self.grn)`) belonging to the current state `redYlw` shown in dark grey. As our approach supports OCL during all development phases, the complete system state can be inspected with OCL

expressions at any point in time. The OCL query expression in the upper right retrieves all present traffic light objects which currently show both `red` and `grn`. The state determination together with OCL querying allows to check positive and negative test cases with respect to structure (objects and attributes) and behavior (operations and state machines).

3.4 Related Work

This section reports on relevant work related to the previously covered topics. A more detailed evaluation of related work can be found in the publications summarized in this chapter ([A18C, A15C, A16C]).

To what extend a precise semantics of UML is needed is discussed in [79] and [12]. The authors of [12] discuss the benefits and drawbacks of a precise UML specification including a runtime semantics from several points of view. That there is a need for a precise semantics can be seen be the publication of newer specifications of the OMG, like the *Semantics of a Foundational Subset for Executable UML Models (FUML)* [66] and its related *Action Language for Foundational UML (ALF)* [64]. One of the first proposals on how to define the semantics of UML class diagrams including OCL constraints was published in [74]. A detailed collection of different approaches on how to define the semantics of UML elements, e. g., denotational, operational, and transformational, can be found in [47]. In [81] a unified semantics framework using predicate logic formulas for the multilayer metamodel hierarchy is presented.

A descriptive introduction to the usage of union and subsets together with composite structures is given in [9]. A more detailed discussion of the semantics has been done in [1] using a set-theoretic formalization, while graph transformations are used in [3, 2] for this. The work in [52] uses a so-called property oriented abstract syntax to define the semantics of what the authors call inter-association constraints (these include subsets and union). [20] defines the semantics of property relations by translating them into a so-called basic UML layer. This work is similar to our work in the sense that it uses UML and OCL to define the semantics, however it differs by translating these constructs into OCL invariants, like the one shown in Listing 3.1 on page 29, and not by defining generic constraints on the meta level. A similar approach that transforms relations between properties into constraints is presented in [59].

Work on derived properties in UML/OCL models has been done in [24]. The authors translate conceptual schema, i. e., UML/OCL models, enriched with derived properties specified in OCL, into first order logic to be able to reason about satisfiability and contradictions. Query operations, which can be seen as another kind of derived properties, were supported in one of the first versions of the USE tool [73, 29]. These kinds of operations are also handled in [57], using an executable metalanguage called KerMeta. However, derived attributes and furthermore derived associations are not discussed.

In our approach we use UML protocol state machines to constrain the model behavior. The structure and the semantics of protocol state machines are also discussed in [71]. The authors present an approach that applies protocol state machines to produce class contracts. The semantics of behavioral state machines is discussed in [48]. The authors apply the semantics for validity proofs of refinement transformations on behavior state machines. A formal semantics for the integration of UML statecharts into OCL, which makes it possible to formulate expressions over states in UML statecharts is presented in [25]. However the authors refer to an older UML version, whereby postconditions of protocol state machine transitions are not handled. The dynamic semantics of state machines is discussed in [11].

4 Runtime Verification using UML and OCL

In this chapter, we summarize the idea of runtime verification using UML and OCL as published in [A17W, A19W, A21C, A23C, A24C]. We show how an implementation needs to be observed by an external monitor to be able to verify its specification defined in an abstract model. One benefit of monitoring an application using this approach, is the tight coupling between the application and its representation as a model, which reveals inconsistencies between them already during the verification task. Verifying a concrete implementation against a model further raises the value of models in the overall development process, since information about the specification that was gained during the beginning of the development can still be used after manually writing code to detect errors or design flaws. After explaining the requirements for a monitor independently of a concrete validation tool, we show how this approach can make use of existing validation features by applying it using the USE tool.

The summarized work in this chapter was incrementally developed. A first approach for runtime verification using UML and OCL was published in [A19W], but was limited to applications running inside the Java Runtime Environment (JRE). Later, this limitation was removed by adding so-called adapters to support different runtime environments [A17W]. While [A17W] demonstrated the application to another object-oriented target platform, the work in [A23C] showed how to apply our approach to verify role-based access control rules on the level of relational databases.

The application of our approach using extended modeling elements, like protocol state machines, is done in [A21C]. In [A24C] the implementation of our approach together with reverse engineering techniques was shown as a tool demonstration.

4.1 Platform Aligned Model

As described in Sec. 2.4 on page 20, runtime verification requires an abstract model that defines relevant parts of the system and the assumptions, i. e., the properties, to be checked. This model should be as abstract as possible to allow a user to focus on the aspects to verify. A simple but powerful abstraction is to leave out unnecessary parts of a system. For example, if a user wants to verify the core logic of a system, plain technical elements, like classes for persisting states of objects, can be ignored. However, to be able to relate abstract model elements to the concrete runtime elements, information about the implementation is needed. Since this information depends on the concrete target platform of the monitored system, we call this model between an abstract specification

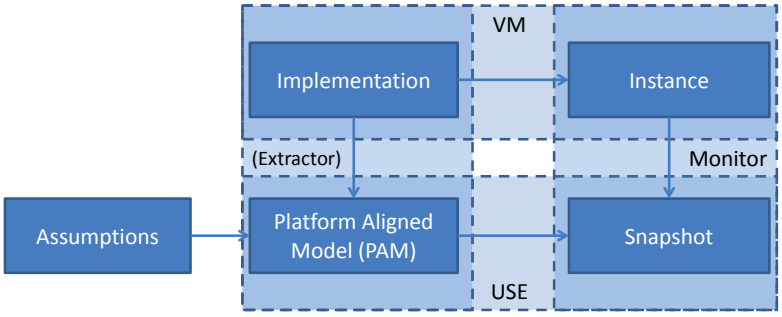


Figure 4.1: Artifacts of the Monitoring Approach ([A21C])

and the implementation a platform aligned model (PAM). The relations between the implementation, the PAM, and the assumptions are shown in Fig. 4.1. Such a PAM can be defined in several ways. For example, when model-driven development is applied that uses models as a central part of the overall development process and generates most parts of the implementation from the models, the PAM can easily be generated, too. The presented approach can then be used to verify the manually coded parts of the system. If models are not a central part of the development process, a PAM can be extracted using reverse engineering techniques as described in [A24C].

4.2 Runtime Snapshots

In our approach, we call an excerpt of a running system that is represented as an instance of a PAM a *snapshot* (see Fig. 4.1). A validation tool can use this instance as a basis to validate static and dynamic constraints. If the environment of the monitored system supports querying the current state of a running system, like it is supported by the JRE, such a snapshot can be created while the system is already running by attaching the monitor to it and following the steps described in the next section. If such a querying mechanism is absent, the monitoring needs to start at the beginning of the execution using the approach described in Sect. 4.2.2 on page 53.

4.2.1 Snapshot Generation

When attaching to an already running system an initial snapshot has to be taken. Otherwise, constraints covering all instances of classes, e. g., invariants and multiplicity constraints, cannot be validated. After this, the snapshot can be incrementally build-up by reacting on events that occur inside the running system. The creation of snapshots needs to consider certain issues that arise by using a more abstract modeling language. For example, common runtime environments like the JRE or the Microsoft CLR are not aware of the concept of associations [84]. Therefore, this information needs to be

mapped by the concrete target platform adapter to the UML metamodel. Figure 4.2 on the following page shows the metamodel defined by our approach that maps commonly available elements inside a monitored system (prefixed by `VM*`) shown in the upper part of the figure to core elements of the UML metamodel, which are shown in the lower part of the figure and are prefixed by `M*`[1]. For example, the concept of a field (`VMField`) can be mapped by an adapter either to an attribute (`MAttribute`) or an end of an association (`MAssociationEnd`). Another considerable fact of commonly used runtime environments is, that they do not necessarily contain all defined classes. Instead, classes are only initialized if accessed the first time to reduce memory consumption. Therefore, a monitor cannot expect, that all classes to monitor are available during the initial snapshot generation.

Next, we describe how a snapshot can be created. While these steps can be applied to other target platforms, we explain these steps based on the Java Virtual Machine (JVM) as the target platform.

1. For all classes in the PAM that can be matched directly (by name or by special annotation information) to an already loaded class in the JVM, all existing instances in the JVM are mapped to newly created instances of the platform aligned model. In detail, this can be done by invoking the operation `instances()` on an object of the type `ReferenceType` which returns proxies to all reachable objects inside the JVM. This – for our approach important – operation was introduced in the JVM version 1.6.

2. For each created abstract instance in step 1 the attribute values are read. The mapping of primitive Java types to primitive OCL types should follow the common practice (c. f. [92]). Attributes with a type of a class defined in the PAM, i. e., reference types, can be read by using the mapping created in step 1. The possibility to define attributes referencing other instances is the reason why the creation of instances (step 1) and this step needs to be separated.

3. For all associations in the abstract model, links are created between corresponding instances. Technically this step can be merged into step 2 for performance reasons. The retrieval of links is discussed in Sec. 4.5.1 on page 72.

These steps need to be aligned if a non object-oriented target platform is used. A developer of an adapter needs to decide how concepts of the target platform can be mapped to an object-oriented model. A first decision that needs to be done is the alignment of monitored artifacts to the metamodel level. If a user can define model elements like classes that can directly be mapped to runtime artifacts, as it is the case if monitoring JRE applications where Java classes can be mapped to modeled classes,

[1] The shown metamodel elements are aligned to the implementation of USE, which is slightly different because of historical reasons.

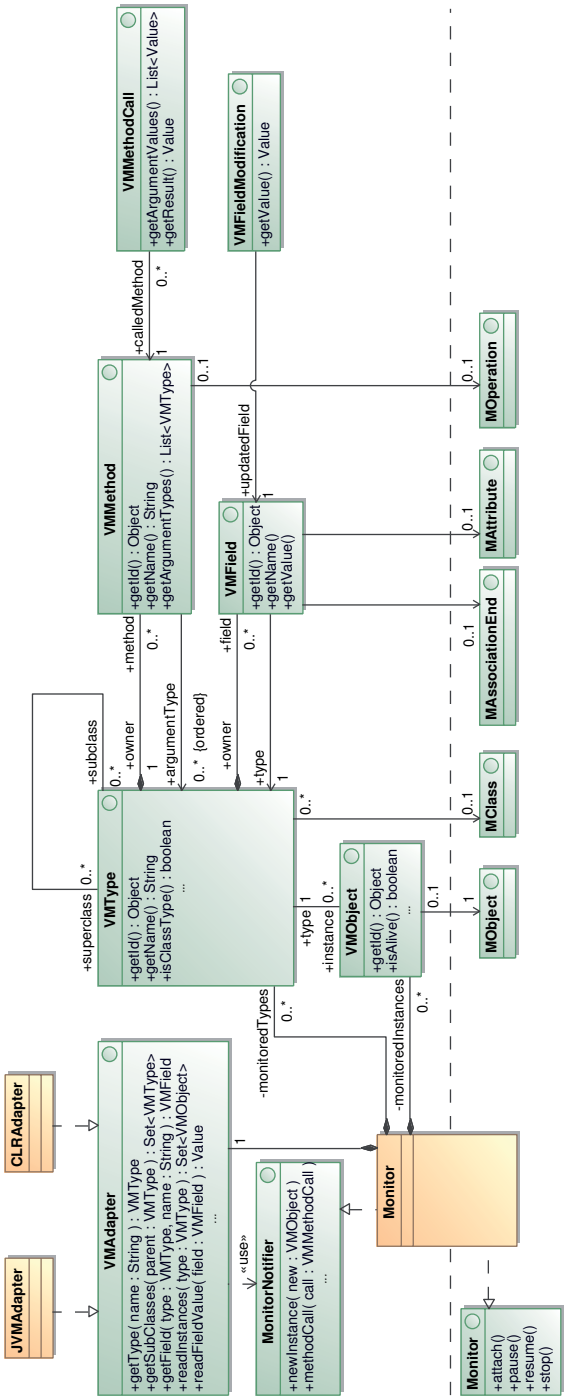


Figure 4.2: Metamodel for Virtual Machines [A17W]

an adapter needs to read the runtime artifacts on the M0 level (instances). However, sometimes it is useful to define a model that is tightly coupled to the adapter and defines the M1 level. For example, to monitor Role-based Access Control (RBAC) rules inside a relational database management system (RDBMS), as shown in Sec. 4.3.4 on page 68, tables, which can be seen as elements of the M1 level, are read as instances on the M0 level. This allows to reuse the RBAC metamodel when another database is monitored, i. e., the model provided to the validation tool is always the same, independent of the concrete database scheme.

4.2.2 Snapshot Synchronization

To be able to verify the behavior of a system, it is required to listen to relevant changes to its state. The previously described initial snapshot generation can be left out if a system is monitored already at the startup. Otherwise, it is essential since information about existing instances is required.

To keep a snapshot synchronized with a monitored system, a monitor needs to keep track of different changes that lead to a modification of the snapshot. Again, we explain our approach by using the JVM as the target. Other target platforms require different handling by a concrete adapter. For example, the first injection point described next would not be required, if all information for artifacts mapped to classes is always present, as it is the fact when monitoring a RDBMS. Further, an adapter is free to implement these intersection points using any applicable technique. Two commonly used techniques are using debugger breakpoints and aspect-oriented programming [27, 75].

The injection points to synchronize a snapshot are:

1. At class initialization to allow the registration of the injection points described next: As already mentioned, this injection point is not necessary if all runtime artifacts that are mapped to classes are always available.

2. At constructors of monitored classes, i. e., classes defined in the abstract model: This allows the monitor to keep track of newly created instances and therefore enables an incremental built-up of the system state in contrast to always building a new snapshot of the running system when needed.

3. At the start of an operation that is specified in the abstract model: This enables the monitor to validate preconditions at runtime and in case of a failure pause the monitored system.

4. Just before the exit of an operation call: This enables the monitor to validate postconditions. The break must occur after the result of the operation is calculated. The JVM provides such a mechanism. To reduce the total number of breakpoints the operation exit breakpoint can be set while entering a monitored operation and can be removed after the postconditions have been validated.

5. When a monitored attribute or link is modified: An application does not need to always use operations to modify attributes of an object. Therefore, a monitor needs the possibility to react on a modification of an object field to synchronize its snapshot. The JVM provides notifications when a field is modified to keep track of changing attributes or single valued association ends. The monitoring of changes to many to many associations is more complicated and is discussed in Sec. 4.5.1 on page 72.

6. When a runtime instance is destroyed: This allows the monitor to remove the corresponding object from the snapshot. Depending on the target platform this is not easy to achieve, since runtime environments that employ a garbage collector do not always provide information about object destruction.

Using the described injection points, a monitor can synchronize the snapshot and notify the used validation tool about the change. The tool can then perform the required validations and react on violations. At these locations, the kind of the runtime verification approach is determined (see Sec. 2.4 beginning on page 20). If the monitor waits for feedback from the validation tool an on-line approach is used, otherwise it is an off-line approach. The decision between trace-storing and non-storing is done by the validation tool, since the monitor just provides the change events. The tool itself decides if these events are stored or discarded. For example, the implementation in USE (see Sect. 2.3 beginning on page 14), used in the next section, is an on-line verification approach, since after each event the monitor waits until the validation tool has finished the required computation. However, the user can deactivate several validations, e. g., the validation of state invariants after every change, to reduce the overhead. Further, it is trace-storing, because USE requires all events to visualize the execution trace, but the computation is done by using a synchronized snapshot and not by using the stored trace.

4.3 Runtime Monitoring in USE

In this section we apply the previously described monitoring approach using the model validation tool USE. This is a summary of work presented in [A19W] and [A21C]. The work also makes use of the extended modeling capabilities introduced in Chap. 3. Afterwards, we show different target platforms that are supported by USE. The support for different targets was introduced in [A17W]. We will demonstrate the following advantages of our approach:

- Assumptions about a running implementation can be validated without the need to modify the source code.

- The state of an implementation can be examined in an abstract way to discover inconsistencies or design decisions.

- Using protocol state machines the correct usage of the defined protocol of a class can be validated.
- Concrete usage scenarios can be visualized by means of a sequence diagram.

These advantages will be exemplified by using the following case study.

4.3.1 Example System

We demonstrate our approach using the open source computer game *Free Colonization* or in short *FreeCol*[2]. FreeCol is an open source implementation of the game Colonization. Colonization was created by Sid Meier and Brian Reynolds and published in 1994[3]. The game is a turn-based strategy game. The goal of the game is to colonize the new world starting in the year 1492. The player can control units of a European country, like England or Spain. By building colonies and trading goods, the player must achieve independence from Europe to win the game. A sample game situation is shown in Fig. 4.3 on the following page. A class diagram containing relevant parts of the core of the implementation is depicted in Fig. 4.4 on the next page. Central to the game is the map the game takes place on. This is realized by the classes `Map` and `Tile` which are related by the qualified association `MapTiles`. Units acting on the map are represented by the equally named class `Unit`. Because Colonization is a turn-based game, units need to keep track of their executed actions in a single turn, as it can be seen by the number of moves (1/1) on the bottom right of the running game in Fig. 4.3. Technically, this is done by the attribute `movesLeft` of the class `Unit`. If the value of this attribute reaches zero by executing actions like moving around or chopping trees, the unit cannot act anymore in the current round of the game. The current location of a unit is determined by the association `UnitLocation`. During the game, a unit can be at several locations, for example, in Europe to wait for a transfer to the new world, on the map to move somewhere else, in a colony as a guard or in a building to produce goods. Therefore, many classes inherit from the class `Location`.

4.3.2 Validating Operation Contracts

The previously described structure of the implementation of FreeCol can be used to generate a snapshot of a running game. This snapshot allows the monitor to validate multiplicity constraints and invariants. To support the verification of contracts between a caller and a callee, pre- and postconditions can be defined in the PAM. If these contracts are defined before or in parallel to the implementation, the USE monitor can be used to verify the correctness of the manually written code w. r. t. specified contracts. If behavior, i. e., method bodies, is generated from the application model, the monitor

[2] `http://www.freecol.org`
[3] `http://en.wikipedia.org/wiki/Sid_Meier%27s_Colonization`

Figure 4.3: Screenshot of Example 'Open Source Game' [A21C]

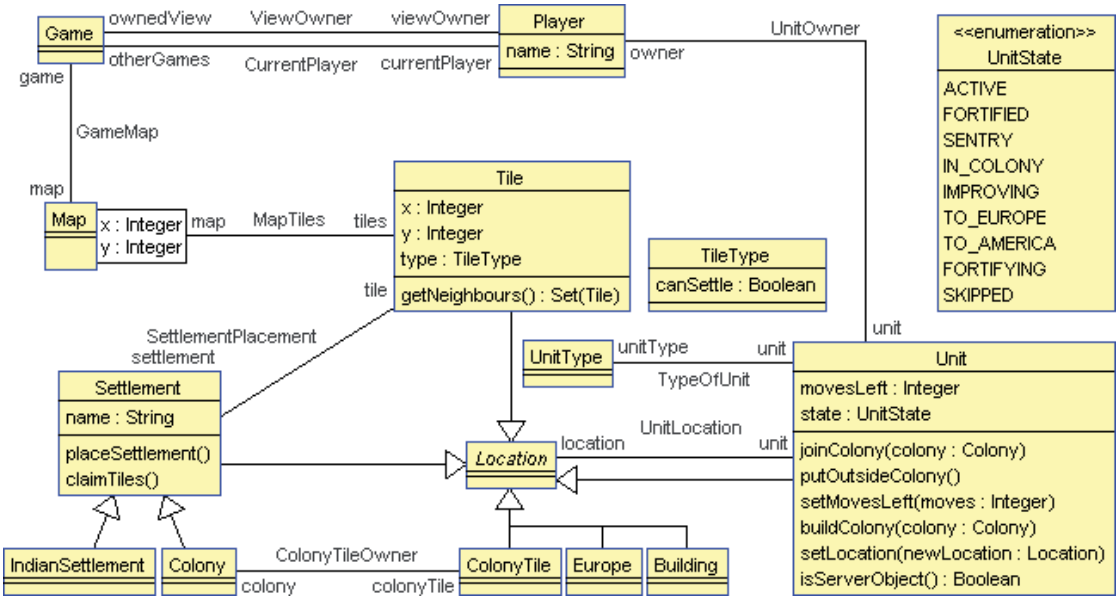


Figure 4.4: Class Diagram of Example 'Open Source Game' [A21C]

can be used to verify the transformation used for generating the implementation. Both possibilities strengthen the use of early design models throughout the whole development process.

The verification of operation contracts using the USE monitor can be exemplified by defining contracts based on the rule of games for FreeCol. Since FreeCol has many rules, we focus on a small excerpt of the overall game: the founding of new colonies. Some informal descriptions of rules for this feature are:

1. To build a new colony, a unit must have enough moves left. After a colony was build, the unit has no more moves left.

2. The tile on the map the new colony is going to be placed on must not be occupied by another settlement.

3. The type of the tile must allow for the placement of settlements on itself. For example, it is not allowed to found a colony on a mountain or on water.

4. No tile directly around a colony is occupied by another settlement.

These rules can be formally defined in OCL using pre- and postconditions, and invariants as shown in Listing 4.1 on the following page. Note, that rules 3 and 4 can be defined as preconditions as well as invariants, since both rules are inherent to all system states, i. e., they define general laws (see Sec. 2.2 on page 12). Defining these two rules in both ways has the following two advantages:

1. After taking an initial snapshot, which has no information about the previously executed operations, they can still be validated by evaluating the invariants.

2. A precondition can be evaluated before any change to the system is performed, whereas an invariant can only report a violation after the violating system state has been reached. Further, pre- and postconditions in USE are automatically evaluated during execution, whereas invariants are only checked if requested by the user. Therefore, by defining both rules as preconditions one gets the benefit of automatic validation together with immediate feedback if a condition fails.

Noteworthy for these constraints is the usage of the query operation `getNeighbours()` as shown in Listing 4.2 on page 59, since it is not directly present in the implementation, but eases the definition of constraints by encapsulating the complex coordinate system of FreeCol. The operation returns all tiles surrounding the tile it was called on. Because of the isometric map in FreeCol, the calculation of these adjacent tiles is done by using deltas that depend on the value of the y coordinate to be even or odd.

As an example consider the snapshot shown in Fig. 4.5 on page 60 that shows relevant instances of the game situation shown in Fig. 4.3 on the facing page. The coordinates of the neighbours of the tile `Tile323` with the coordinates `(15,51)`, that is the tile the

Listing 4.1: Game Rules as OCL Constraints

```
context Unit::buildColony(colony:Colony)
  -- Rule 1
  pre     movesLeft: self.movesLeft > 0
  post noMovesLeft: self.movesLeft = 0

  -- Rules 2 and 3
  pre tileIsEmptyAndFits:
    self.location.oclIsKindOf(Tile) and
    self.location.oclAsType(Tile).settlement.isUndefined() and
    self.location.oclAsType(Tile).type.canSettle

  -- Rule 4
  pre noSurroundingColonies:
    self.location.oclIsKindOf(Tile) and
    self.location.oclAsType(Tile).getNeighbours()->forAll(t |
      t.settlement.isUndefined())

context Colony inv validTileType:
-- Rule 3 as an invariant
  self.tile.type.canSettle

context Colony inv noNeighbours:
  -- Rule 4 as an invariant
  self.tile.getNeighbours()->forAll(t | t.settlement.isUndefined())
```

pioneer located north west in Fig. 4.3 on page 56 is placed on, can be retrieved by using the following OCL query:

```
let centerTile = Tile.allInstances()->any(x=15 and y=51) in
 centerTile.getNeighbours()->collect(n:Tile | Tuple{x=n.x, y=n.y})
```

Evaluating this query results in a sequence of tiles containing all surrounding tiles starting north (N) and going clockwise around the coordinates of the tile:

```
Sequence{ /*N*/ Tuple{x=15,y=49}, /*NE*/ Tuple{x=16,y=50},
          /*E*/ Tuple{x=16,y=51}, /*SE*/ Tuple{x=16,y=52},
          /*S*/ Tuple{x=15,y=53}, /*SW*/ Tuple{x=15,y=52},
          /*W*/ Tuple{x=14,y=51}, /*NW*/ Tuple{x=15,y=50}
} : Sequence(Tuple(x:Integer,y:Integer))
```

By annotating the operation with `@Monitor(isQuery="true")` the monitor ignores this operation, i. e., it will not try to listen to any calls to this operation inside the running system, but still allows a modeler to reuse complex expressions when defining constraints.

Using the class diagram shown in Fig. 4.4 on page 56 and the constraints given in Listing 4.1 as the PAM, one can verify the defined properties (the rules of game) by

Listing 4.2: Query Operation Ignored by the Monitor

```
class Tile
 ...
 operations
  @Monitor(isQuery="true")
  getNeighbours() : Sequence(Tile) =
   -- deltas for the different directions
   let N  = Tuple{oddDX =  0, oddDY = -2, evenDX =  0, evenDY= -2} in
   let NE = Tuple{oddDX =  1, oddDY = -1, evenDX =  0, evenDY= -1} in
   let E  = Tuple{oddDX =  1, oddDY =  0, evenDX =  1, evenDY=  0} in
   let SE = Tuple{oddDX =  1, oddDY =  1, evenDX =  0, evenDY=  1} in
   let S  = Tuple{oddDX =  0, oddDY =  2, evenDX =  0, evenDY=  2} in
   let SW = Tuple{oddDX =  0, oddDY =  1, evenDX = -1, evenDY=  1} in
   let W  = Tuple{oddDX = -1, oddDY =  0, evenDX = -1, evenDY=  0} in
   let NW = Tuple{oddDX =  0, oddDY = -1, evenDX = -1, evenDY= -1} in
   -- all possible directions
   let directions = Sequence{N, NE, E, SE, S, SW, W, NW} in

   -- the used deltas depend on the value of y
   let odd = self.y.mod(2) <> 0 in

   directions->collect(d |
    let x = self.x + if odd then d.oddDX else d.evenDX endif in
    let y = self.y + if odd then d.oddDY else d.evenDY endif in
     self.map.tiles[x,y])->excluding(null)
 ...
end
```

executing the system under monitor (SUM) and connecting the monitor. After the connection is established a snapshot is taken. In USE, these snapshots can be visualized by means of an object diagram, as it was done to explain the coordinate system using Fig. 4.5 on the following page for the game state given in Fig. 4.3 on page 56. Note that the object diagram is an instance of an earlier class diagram used in [A19W] that slightly differs from the one shown in Fig. 4.4 on page 56. For example, the shown class diagram (the newer one) contains an attribute `state` that was introduced to be able to verify the correct usage of protocols, which is covered in the next section. However, the structure that defines the map is still valid and can be used to describe the visualization of a snapshot as an object diagram. The overall snapshot consists of nearly 6000 objects and 4000 links which makes it impossible to manually extract an informative object diagram. USE allows a user to select objects that should be shown or hidden in an object diagram by using several features. Two useful ones are the selection by an OCL expression and the selection of related objects by path length (see [A10W] for more information). One can see, that the European colony 'Isabella' that is placed at the top right corner in the

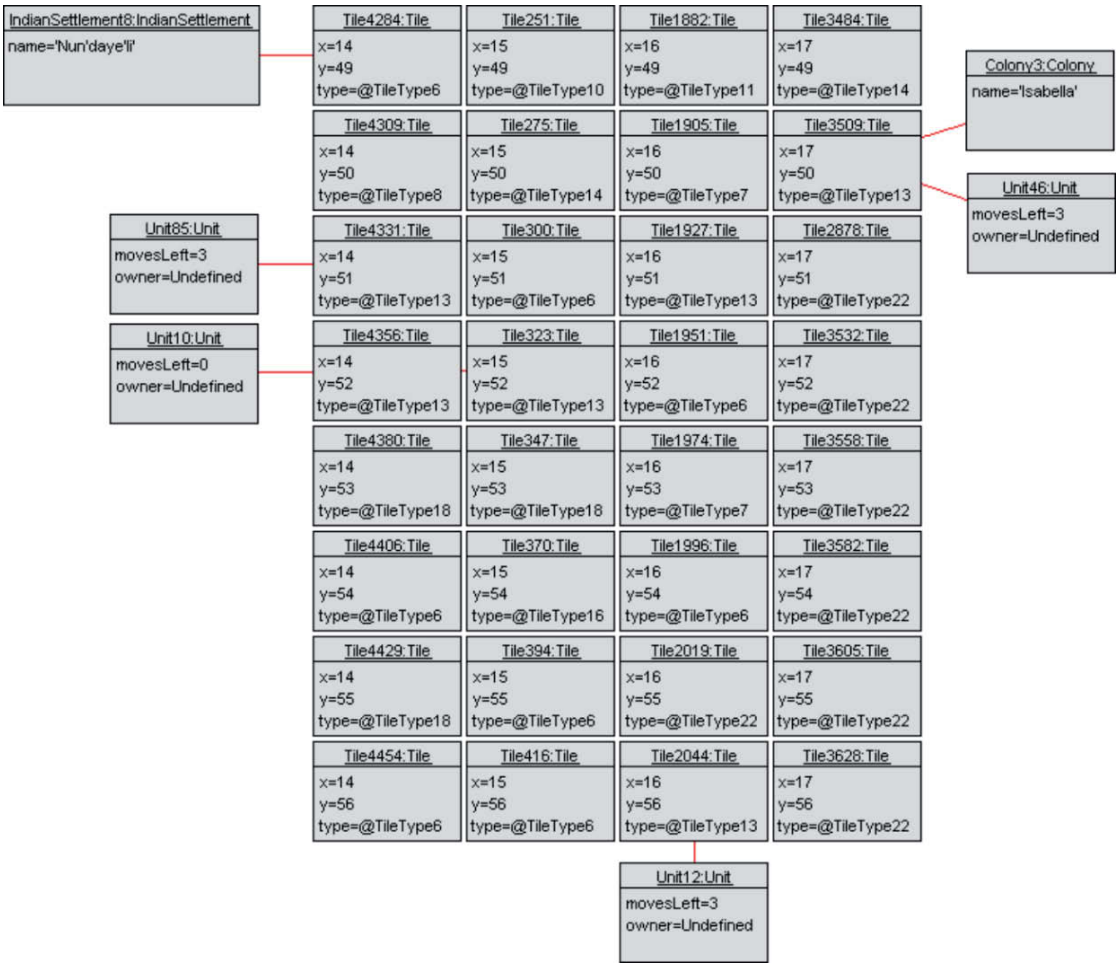


Figure 4.5: Snapshot of an Exemplary Game Situation [A19W]

screenshot is place on the tile with the coordinates (17,50) and that it further hosts a unit (in the running application this is highlighted by the number in the center of the colony). The pioneer shown in the middle of the map in Fig. 4.3 on page 56 is named `Unit12` in the shown object diagram.

While the previously shown invariants `validTileType` and `noNeighbours` can be validated with a given snapshot, the validation of dynamic contracts requires the execution of the system. To validate the defined contracts of the operation `Unit::buildColony`, the monitored system needs to be resumed and the corresponding feature of the system needs to be invoked. For this, a user needs to invoke the functionality to build a colony inside the running game. The monitor then listens to operation calls and changes of values of the modeled elements and evaluates defined constraints. If a constraint fails,

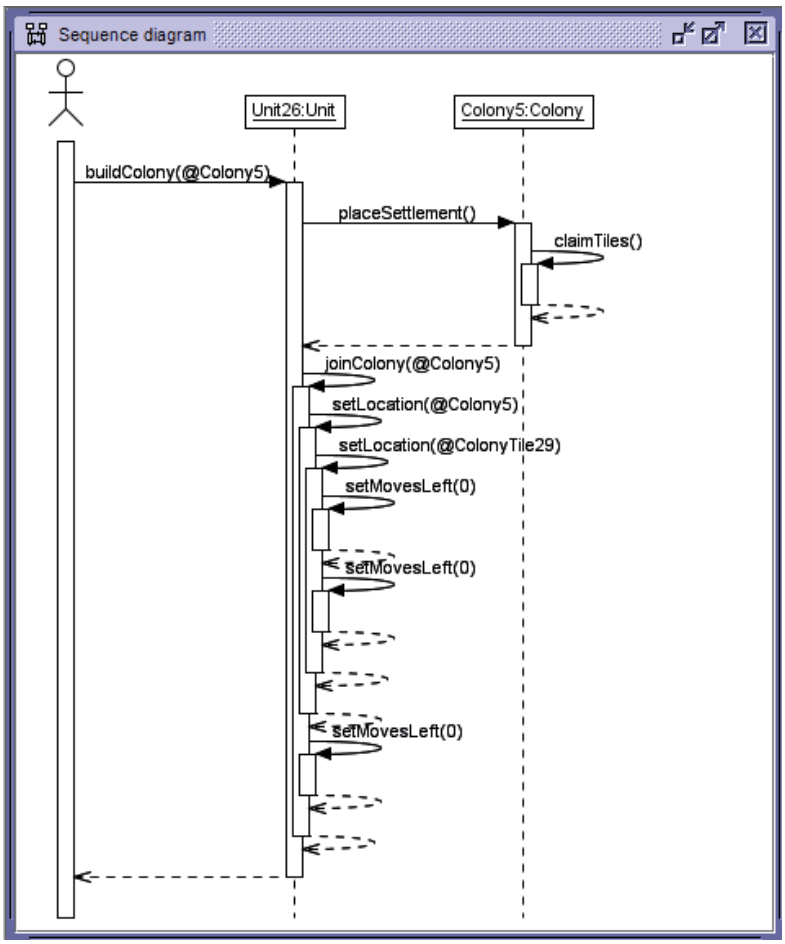


Figure 4.6: Sequence Diagram of a Monitored Execution [A21C]

the monitored application is paused and the user is informed about the violation. USE then provides all of the functionality described in Sec. 2.3 starting on page 14 to examine the failure.

One benefit of abstraction introduced by our monitoring approach can be seen by comparing the class diagram consisting of 14 UML classes that were required to validate the given game rules to the overall complexity of the implementation counting 551 Java classes. The monitored execution can further be visualized by diagrams supported by USE. For example, the sequence diagram shown in Fig. 4.6 shows the operations calls executed by the operation `buildColony` using the sample game state. Besides the described validation capabilities these visualizations of execution traces can be used for documentation purposes or to reveal unusual call sequences. For example, in the same

sequence diagram it can be seen that the operation `setMovesLeft()` is called three times with the same argument. One can use this information to examine the reason, why it is called so often.

4.3.3 Validating Protocol Usage

Pre- and postconditions can only define a contract for a single operation call, since they can only access the system state just before and just after an operation call. To be able to specify contracts that define the correct order of operation calls on an instance, i. e., a usage protocol, a modeler would need to manually keep track of the execution order by storing information about previously called operations.

For the FreeCol example, one might want to validate the correct usage of the operations of a unit that are related to the possible movements during a game. For example, a call to the operation `joinColony(...)` should not be allowed if a unit is already inside a colony or if it is not active, i. e., doing something else. For this, an attribute defining the current state of a unit together with corresponding pre- and postconditions can be used. As it can be seen in the class diagram shown in Fig. 4.4 on page 56, the implementation of FreeCol contains an enumeration named `UnitState` for the different states a unit can pass through and an attribute named `state` for the class `Unit` to keep track of the current state of a unit. Both elements, together with appropriate pre- and postconditions, can be used to define the valid state transitions for a unit. However, for complex protocols, this technique requires many pre- and postconditions that define the valid transitions.

A more abstract way to define such protocols is the usage of UML protocol state machines as described in Sec. 3.3 beginning on page 38. One benefit of using PSMs instead of pre- and postconditions is the possibility to visualize the states and transitions by means of a state machine diagram, which can directly highlight the life-cycle of an object. Further, they can be used for monitoring to verify correct usages of protocols even if an implementation does not contain any kind of state machine implementation.

Figure 4.7 on the facing page shows an example state transition while playing FreeCol. A pioneer is located in the center of the shown map on the left side. The right map shows the game state after the pioneer has build a new colony called *Jamestown* and entered it. The sketched state machine instances, displayed below the two maps, exemplify the idea of verifying a usage protocol: while executing a task in the running application, the state machines defined in the PAM keep track of the states and report any violating operation call to the user of the monitor. This helps to identify inconsistencies between the specification and the implementation as it is shown in the following example, for which we want to define and verify the protocol for the class `Unit` related to the entering and leaving of a colony.

Since our example is based on an existing implementation and is not build in a model-driven way, we reverse engineered the defined assumptions by observing the game-playing and examining the source code. This lead to a first definition of a PSM for the class

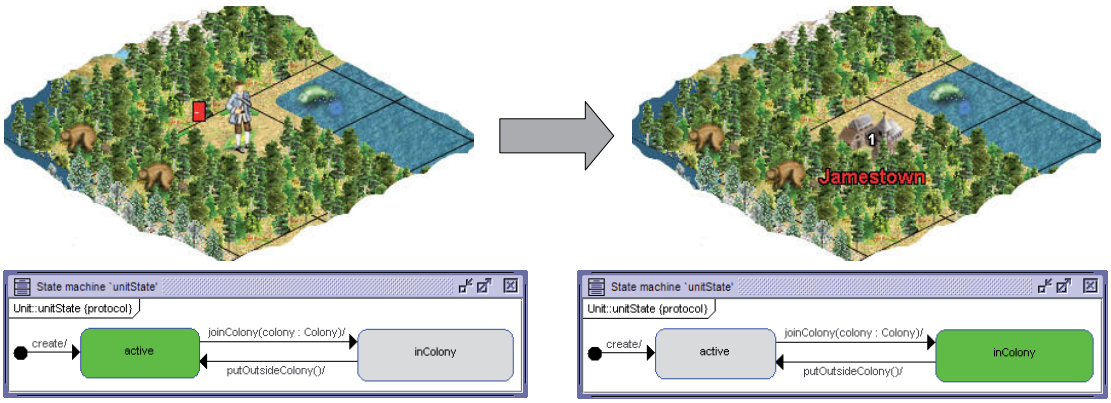


Figure 4.7: An Exemplary Game Transition in FreeCol [A21C]

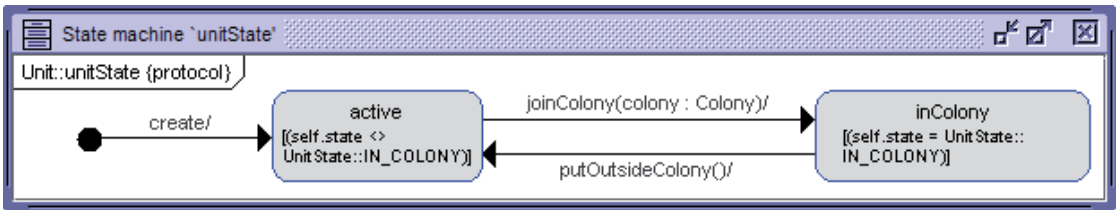


Figure 4.8: Protocol State Machine for the Class Unit

`Unit` shown in Fig. 4.8 defining the two states `active` and `inColony`. The transitions between both states are executed if the operation `joinColony(colony:Colony)` is called if the unit is in state `active` leading to the state `inColony` or if the operation `putOutsideColony()` is called while the unit is in state `inColony`. All other calls to these two operations, i. e., a call to `joinColony(...)` while the unit is in state `inColony`, are invalid. Because any call to an operation not covered by a PSM is ignored by the PSM [69, p. 549], we can leave out all other possible states and transitions of a unit, which keeps the PSM focused to the protocol to verify. Note that the states defined for the PSM are at first not related to the enumeration literals defined by the enumeration `UnitState` shown in the class diagram in Fig. 4.4 on page 56. However, to strengthen the PAM and to be able to determine the correct states after taking a snapshot, state invariants were introduced that relate the states to the monitored values of the attribute `state` of the class unit. Using the previously defined assumptions about the usage protocol of the class `Unit` that we specified by means of a protocol state machine, we can apply our approach to the running application to validate these assumptions and to identify possible mismatches between the implementation and the assumptions.

After taking an initial snapshot of the game situation shown on the left in Fig. 4.7 we need to invoke the state determination command in USE (see Sec. 3.3.3 on page 45) to

set all state machine instances to the correct state to be able validate the behavior. After the states have been determined, the states of the relevant units of the snapshot are as expected (`active`). After resuming the game and building the new colony *Jamestown* we get a valid sequence of operation calls, which can be seen in the monitored sequence diagram shown in Fig. 4.6 on page 61. We observed, that the invocation of the operation `joinColony(...)` on the object `Unit26` indeed leads to a transition of the PSM instance to the state `inColony` because no violation of the defined PSM was reported. Without querying the snapshot, we can conclude, that the value of the attribute `Unit::state` was set to `IN_COLONY`, because otherwise, a violation of the state invariant of the state `inColony` would have been occurred.

While the previously performed steps validate the correct sequences of executed operation calls, i. e., the operation `joinColony(...)` is called while a unit is in state `active`, incorrect behavior is still possible. An implementation can violate a PSM, even if none of the defined operations is called, by performing changes during the execution of other operations that lead to a violation of a state invariant. For this, USE provides the ability to evaluate the state invariants of all PSM instances. Either after each change or manually. When validating the state invariants in our example after the colony has been build, one would be notified that not all PSM instances are valid.

A user can now examine the violating instances and the recorded traces to identify the reason. For our example, Fig. 4.9 on the next page highlights the relevant parts of the snapshot of the running game before the new colony is founded as an object diagram. The shown part of the snapshot is divided into two parts, which are important while validating the assumptions about the state transitions. Because we monitored a single user game on a single machine the instance of the game contains both, the data used by the game server and the client. By looking at the instances `Tile3466` on the server side and `Tile1583` on the client side, one can see that the server part has more information about the game than the client part. Both instances represent the same tile on a map, because their positions are equal, but the client instance does not know what type the tile has. To be able to determine if an object belongs to the server or client side we also monitored the class `Game` together with the association `ViewOwner`. If a game object is not linked to a player by this association it is the server game. The object diagram further shows the owned units of the player named *ada* and the object for the tile on which we want to build a colony (`Tile4228` resp. `Tile225`).

After building the colony, the value of the attribute `state` of the client and server object for the unit that has built the colony is changed to `IN_COLONY`, but USE reports an error for the PSM instance of the client object. This is due to the fact that the operation `buildColony(...)` is only called on the server object and only the new values are transfered to the client object. Therefore, USE did not execute a transition from the source state `active` to the target state `inColony` for the client unit, but monitored the change of the attribute `state` to `IN_COLONY`, which leads to the violation of the state invariant of the state `active`.

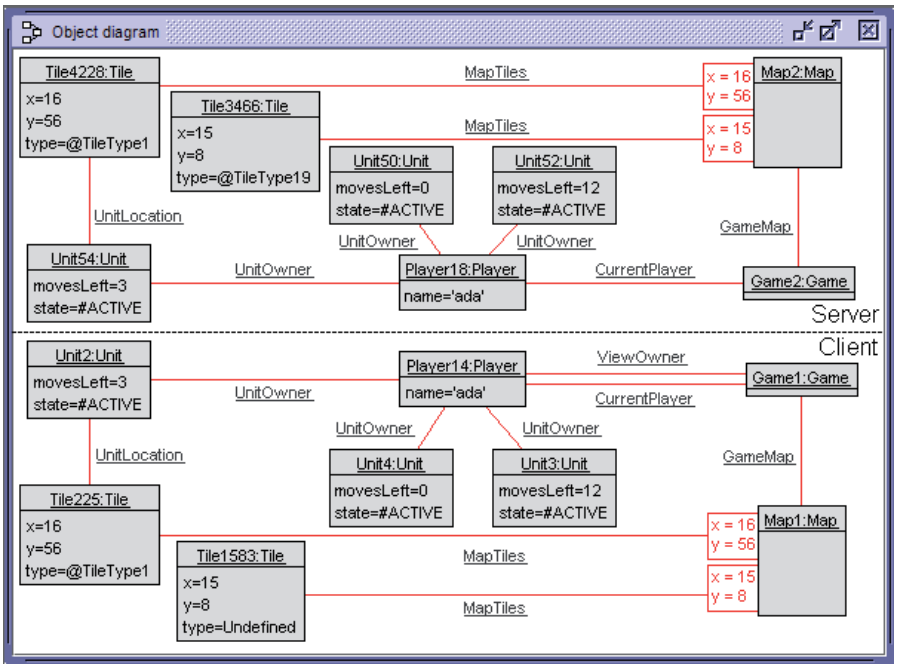


Figure 4.9: Parts of a Snapshot Taken at Runtime [A21C]

Because the separation of the client and server objects seems to be a valid design decision we can ignore the violations caused by objects belonging to the client and continue the monitoring process to retrieve further information about the validity of our assumptions. To test the defined protocol we use another unit and let it join and exit the colony. While executing this scenario another issue arises because entering an existing colony, i. e., a unit only enters a colony without building it before, does not lead to an operation call to `joinColony(...)`. Instead, only `setLocation()` is called which is not handled by the PSM and therefore does not execute a transition keeping the PSM instance in the state `active`, but the attribute value of the runtime instance is set to `IN_COLONY` which violates the state invariant of the state `active`.

Using this information a user of the monitor needs to decide where the error is located: in the implementation or in the PAM. For our example, we assume that the PAM needs to be modified although it seems to be an unsound usage of the `Unit` class. If we want to adapt our PSM to the last discovered facts, we need to handle the client server separation and the additional operation calls. The modified PSM is shown in Fig. 4.10. The additional operation `setLocation(newLocation:Location)` leads to two new transitions in the PSM. Both transitions start from the state `active` but differ in their target and guard. If the new location is of type `ColonyTile`, which represents special tiles related to a colony, the new state after the execution is `inColony` otherwise

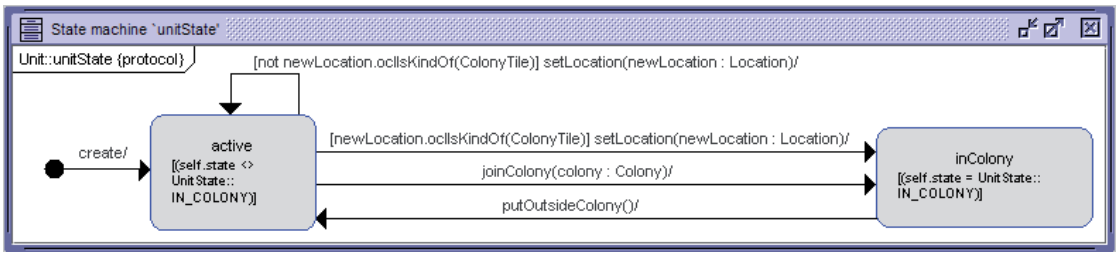


Figure 4.10: Extended PSM for the Class `Unit` [A21C]

the state does not change. Interestingly, when a `Unit` object leaves a colony this leads always to a call to `putOutsideColony()`.

When using this modified PSM shown in Fig. 4.10 all scenarios described above lead to the expected changes of the PSM states. Besides the manual execution of observed game situations the presence of computer controlled players in the game can be used as a test driver. As with the manual play, all analyzed operations are also used by computer controlled players. We used this to strengthen our PAM.

If one looks at the reverse-engineered PSM, we can conclude that the usage of the covered operations is not as clear as it could be. There are two possibilities for a unit to reach the state `inColony`: (1) by using the clearly named operation `joinColony(...)` or (2) by using the general operation `setLocation(...)`, which determines the transition to take by evaluating the runtime type of the new location. Both versions have their benefits and drawbacks: the first option is more strict, but moves more responsibilities to the caller, whereas the second one is easier for the caller, but may lead to wrong usages if the caller is unaware of the fact that this operation is more than just a setter. Regardless of which option is better, it would be a much cleaner design if only one of the options is used. If our proposed runtime verification approach using a well-defined PSM would have been used, such design flaws could have been detected.

4.3.4 Target Platforms

The source for our proposed approach was an implementation of the monitor that was directly coupled to the JRE. To support multiple target platforms, an abstraction layer between the monitor and the target platform was introduced [A17W]. An adapter is queried by the monitor to gain information about the running system, e. g., static information like available classes and attributes as well as runtime values. The next sections briefly summarize the supported runtime environments.

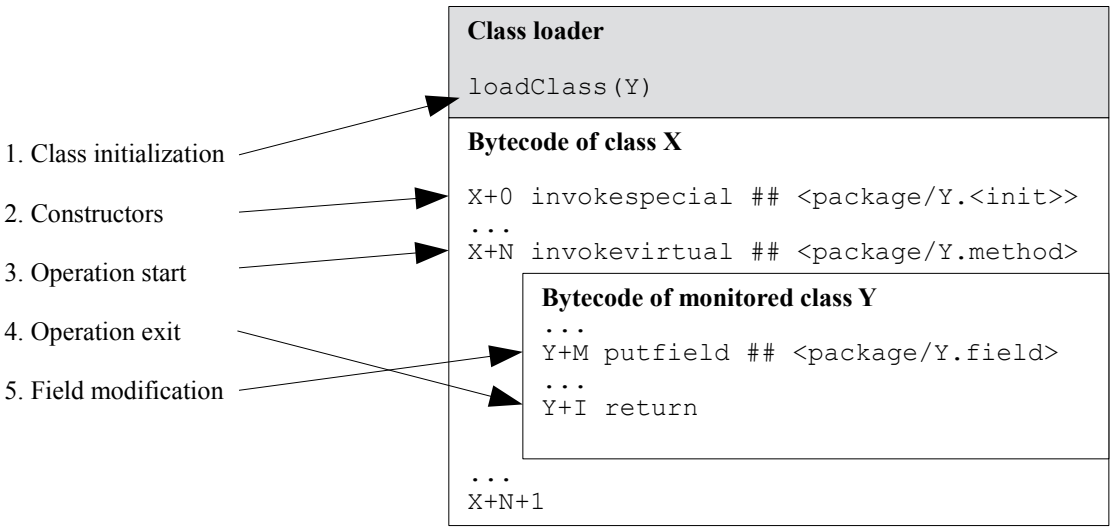


Figure 4.11: Monitoring Events and their Location on the Bytecode Level [A19W]

Java Runtime Environment

As mentioned before, the JRE was the first supported target platform [A19W, A21C]. The chosen approach to query and monitor an application executing inside a JVM is the use of the remote debugger of the JVM. One benefit of using the remote debugger is that there is no need to change either the sourcecode or the bytecode. One drawback is the relatively high delay introduced by this approach. If timing is critical, an approach using aspect oriented programing (c. f. [75]) could be used by adding a new adapter. For both kinds of listening approaches the relevant interception points, recall the injection points from Sec. 4.2.2 on page 53, on the bytecode level, as shown in Fig. 4.11, are the same. For example, to monitor an instance creation one needs to get notified if a constructor is called by the bytecode instruction `invokespecial`. The injection point for recognizing the destruction of an instance is missing in this figure, since it has no direct representation in bytecode. In Java this is done by the garbage collector and other techniques are required to react on instance destructions. For now, the USE monitor does not cover this injection point.

Microsoft Common Language Runtime

The support for the Microsoft Common Language Runtime (CLR) was added by a student for his diploma thesis [33]. In parallel, the abstraction layer for adapters was developed by the author and the results were published in [A17W]. The adapter for the CLR directly queries the heap of a running application inside the CLR, since the remote

debugger API provided by the CLR lacks some essential functionality, like for example, access to all instances of a class. Further, the implementation of the debugging API is currently not well-documented and some operations lead to unexpected errors. However, as a prototype, the adapter showed that the defined abstraction layer supports multiple target platforms.

Relational Database Systems

A simple adapter to read and monitor data in a relational database can be build by reading tuples as instances of classes. By the nature of an RDBS, monitoring dynamic behavior is limited to data manipulation statements (insert, update, delete). Therefore, this kind of monitoring is not very useful. Another approach, as presented in [A23C], combines a well-defined metamodel for role-based access control (RBAC) rules with a specific adapter for the USE monitor. Using this approach, a modeler can define RBAC rules and validate them by monitoring the database. One typical RBAC rule that is not commonly supported by database systems is dynamic separation of duty (dSoD). Such an dSoD rule could for example state that a single person is not allowed to first approve and afterward validate the same cheque, even if the person has defined rights to perform both tasks on cheques. The RBAC approach for the USE monitor can report violations of these rules. Further, it can generate test cases for possible violation scenarios by applying the USE model validator.

4.4 Abstraction Concepts

One of the main advantages of our monitoring approach is the possibility to validate an implementation in an abstract way. By using UML as the modeling language, one automatically gets a first abstraction by the support of associations. By using associations, related instances can be identified more easily in diagrams than by using attributes. However, there are other abstractions that can be used by our monitoring approach. Three of them are described in the next sections.

4.4.1 Abstracted Superclass

Since a PAM does not need to contain all implementation classes, but still needs to keep track of instances of possibly unmodeled subclasses, the concept of an *abstracted superclass* is introduced (shortly described in [A17W]). An abstracted superclass is a class in a PAM that collects instances of unmodeled subclasses of the implementation to still be able to access all instances of an incomplete modeled generalization tree.

Consider the inheritance trees shown in Fig. 4.12 on the next page. On the left side, the complete inheritance tree regarding settlements of the implementation of FreeCol is shown, whereas on the right side only the classes considered by the PAM are shown. To

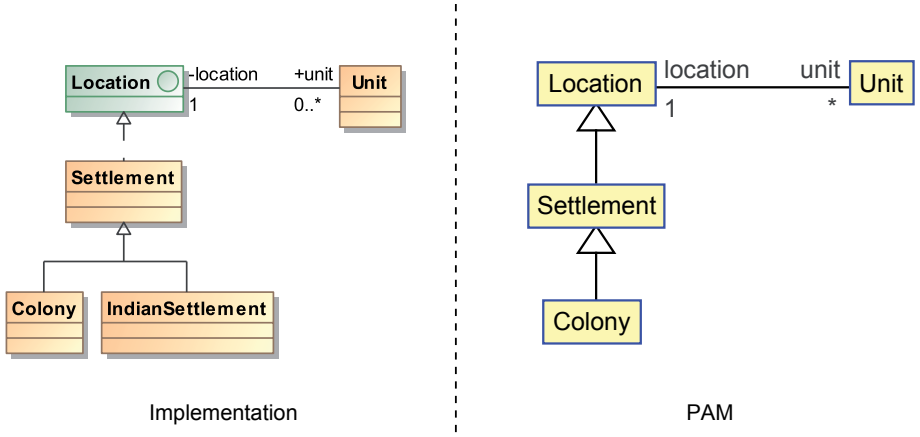


Figure 4.12: Example of an Abstracted Superclass

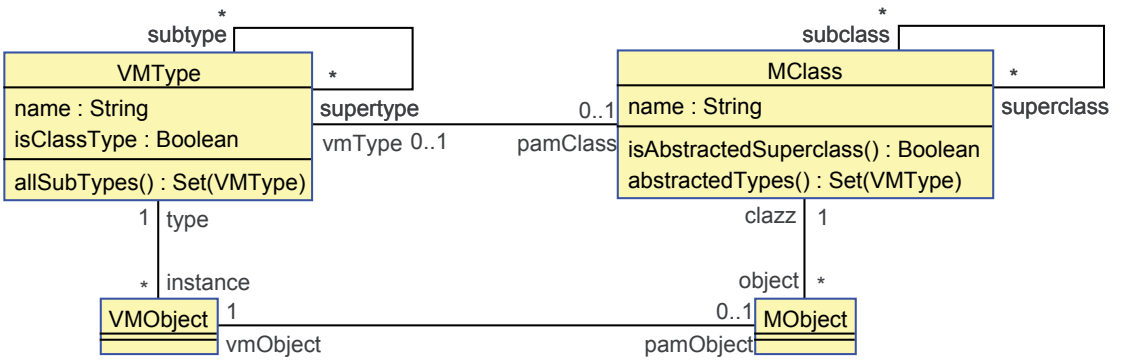
still be able to navigate from a unit to its location, all instances of the implementation class `IndianSettlement` and the corresponding links must be available. In this example, the class `Settlement` acts as an abstracted superclass by saving instances of the class `IndianSettlement` as instances of itself inside a snapshot.

This concept adds additional information to cover into the basic monitoring metamodel discussed earlier. The relevant parts of the metamodel are shown in Fig. 4.13 on the following page. The concept of an abstracted superclass adds two additional operations and new constraints to the metamodel, which are shown below the figure. The operation `isAbstractedSuperclass()` identifies a class as an abstracted superclass, if it is mapped to a runtime type that has at least one subtype that does not map to a class in the PAM. Note that only direct subtypes are considered. If an indirect subtype is not mapped to a class in the PAM, there could be another abstracted superclass in between. To retrieve all covered types for a given class in the PAM the operation `abstractedTypes()` is introduced. The operation calculates all implementation types for which instances need to be mapped to the PAM class. The operation needs to examine the whole inheritance graph, since an unmapped type can have other unmapped types. However, if a direct or indirect subtype is covered by another abstracted superclass this type and it subtypes are not included into the result.

In addition to the previously explained operations the constraints for a valid runtime snapshot need to be aligned. For a given class of a PAM, all of its objects need to be mapped to the runtime instances of all covered runtime types:

```
1 context MClass inv coveredInstances:
2     self.object.vmObject = self.abstractedTypes().instance
```

Figure 4.14 on page 71 shows the instance of the metamodel part used in this section that represents the example given in Fig. 4.12 including two different runtime instances.




```
1 context MClass::isAbstractedSuperclass():Boolean def:
2   not self.vmType.oclIsUndefined() and
3   self.vmType.subtype->exists(vmSubType |
4     vmSubType.pamClass.oclIsUndefined())
```

```
5 context MClass::abstractedTypes():Set(VMType) def:
6   Set{self.vmType}->closure(st|st.subtype->select(
7     pamClass.oclIsUndefined()))
```

Figure 4.13: Relevant Parts of the Metamodel for Abstracted Superclasses

The runtime instance `jamestownVM` of the runtime type `Colony` is linked to the snapshot object `jamestownPAM` of the class `Colony`, whereas the runtime instance `xaymacaVM` of the runtime type `IndianSettlement` is linked to the snapshot object `xaymacaPAM`, which is an instance of the abstracted superclass `Settlement`. Using the described approach it is ensured, that no information is lost at association ends or attributes of the type of an abstracted superclass. The navigation from `Unit` to the association end `location` still leads to all settlements including the unmodeled class `IndianSettlement`.

To be able to determine that an object is not a runtime instance of the given class, a monitor implementation can provide special attributes, for example, an attribute for the concrete class name.

4.4.2 Connected Instances

For runtime verification, it is sometimes necessary to include classes that are not only specific to the application model [51]. During runtime, all instances of them would be covered by a snapshot, including those instances that are not connected to any instance of a relevant PAM class. For example, if the monitoring of the Java collection class `ArrayList` is required, the monitor would include all lists contained in the runtime. Since lists are used heavily in different parts of the JRE this would lead to a much

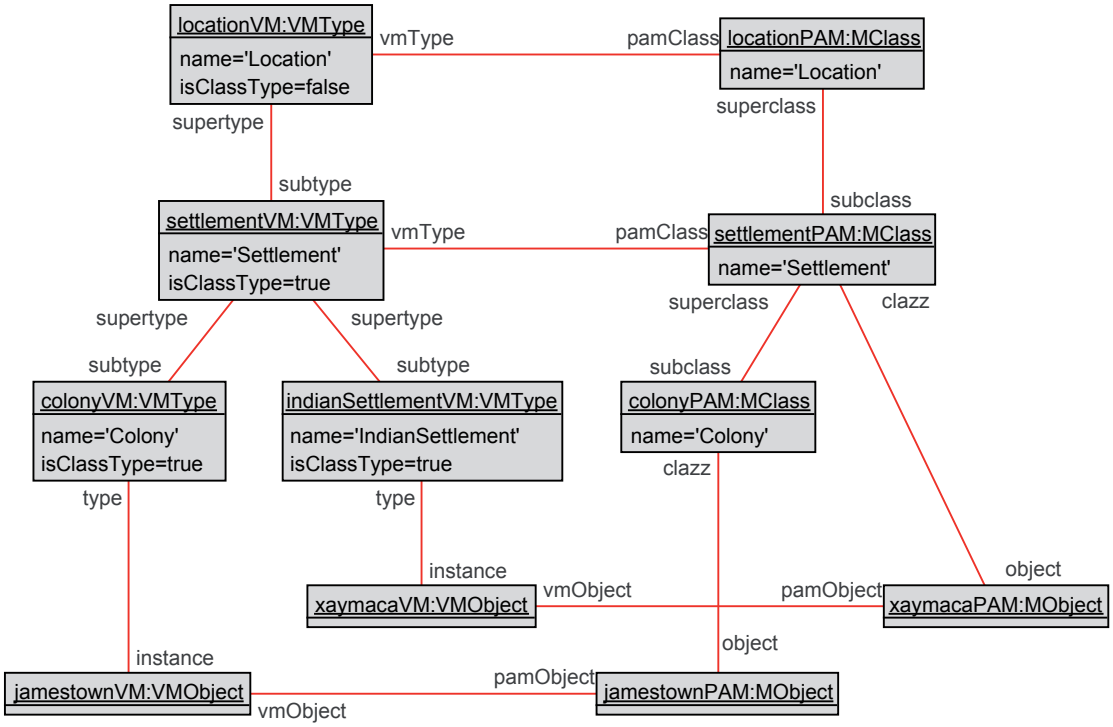


Figure 4.14: Instance of the Monitor Metamodel with an Abstracted Superclass

bigger snapshot than required. To avoid this, the USE monitor supports the annotation property `onlyConnected` which enables the monitor to exclude instances not connected to the graph of objects defined by a PAM. Note that this feature requires a lot of computation since the monitor needs to determine for each instance whether it is used by a class of the PAM.

4.4.3 Excluding Sub-Calls

While the previously described features consider structural abstractions, also abstractions for the dynamic behavior were recognized as being useful. One of such features is the possibility to exclude the monitoring of all sub-calls inside an operation, even if the called operations should be monitored. This can be used reduce the validation overhead by excluding irrelevant parts and also to reduce the complexity of the resulting diagrams.

As an example, consider the excerpt of a Java console application shown in Listing 4.3 on the next page, which asks a user for a regular expression, an input string to match, and prints all found matches to the console. Monitoring the operation `find()` of the class `Matcher` would unveil that the Java implementation of the console also uses a matcher to format the given output string, which may contain a format pattern, as shown in the

Listing 4.3: Simple Java Console Application

```
...
pattern = Pattern.compile(console.readLine("%nEnter your regex: "));
input   = console.readLine("Enter input string to search: ");
matcher = pattern.matcher(input);

while (matcher.find()) {
  console.format("I found the text \"%s\" starting at " +
                 "index %d and ending at index %d.%n",
                 matcher.group(), matcher.start(), matcher.end());
}
...
```

sequence diagram in Fig. 4.15 on the facing page. To suppress the monitoring of these calls, the operation `readLine` in the PAM can be annotated as follows:

```
class Console
 operations
  @Monitor(ignoreSubCalls="true")
  readLine(prompt:String, args:Sequence(OclAny)):String
end
```

Using this annotation excludes the internal usage of the class `Matcher` and removes the calls inside the gray box and the now unused objects including their lifelines from the shown sequence diagram. The resulting trace is now more focused on the intended monitoring.

4.5 Limitations and Possible Solutions

This section discusses encountered limitations of our runtime monitoring approach. For each limitation, the reasons are shown and possible solutions are given.

4.5.1 Link Retrieval

The retrieval of links of a one-to-many association (`1..*`) is straight forward. The instances at the association end marked as *many* are queried for an attribute of the opposite rolename and this attribute is read. This automatically leads to a one to many association in an UML tool. Problems arise, if no such single valued attribute is present, because the monitor needs detailed knowledge about the implementation of collection valued attributes. For example, to query a multi-valued attribute in Java for each container class of the collection library or, if present, for each custom container class, a handler must be implemented that can query this container class for the contained

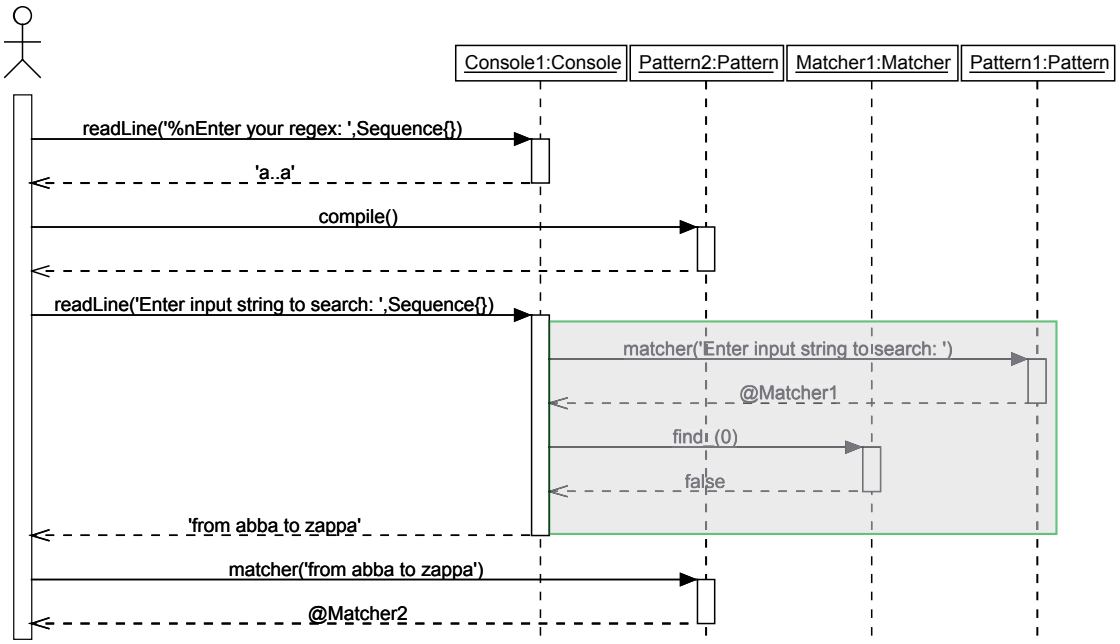


Figure 4.15: Sequence Diagram of a Detailed Execution Trace

values. While this issue is addressed in programming languages by abstracting the iteration over different containers by some kind of iterator pattern, a monitor in general cannot make use of such abstraction mechanism. First of all, this relies on the fact, that it can only read the current state of a system under monitor. But also, if the target platform allows an external program to execute program code, like it is possible using the remote debugger of the Java virtual machine, it is not guaranteed that the execution does not change the current system state.

4.5.2 Monitoring of Interfaces

For classes, the monitor can extract the required information out of the PAM by querying the defined attributes of a class and retrieving the current values. Since interfaces cannot define attributes, the monitoring of the states of instances of classes that implement such an interface is not possible without further information about the concrete implementations.

One solution would be to add additional information about the concrete implementations into the PAM, by guiding the monitor to the concrete fields of implementation classes as it is illustrated in the USE model shown in Listing 4.4 on the next page. One drawback of this approach is the high knowledge that is required about the implementation, which leads to a loss of abstraction.

Listing 4.4: Monitoring of Interfaces

```
1 class MyInterface
2   attributes
3     @Monitor(interfaceImpl="MyClassA.anInteger")
4     @Monitor(interfaceImpl="MyClassB.myInteger")
5     anInteger:Integer
6 end
```

Another solution would be to annotate operations as getters. This would allow an adapter to execute this operation and to use the return value as an attribute value. However, the aforementioned issues of executing parts of a program during debugging still exist.

4.6 Related Work

In the following, we summarize different approaches and tools for runtime verification. Since there are many of such tools, this enumeration is just an excerpt, but should give the reader an impression about the different kinds of runtime verification approaches. A more detailed comparison of such approaches can be found in [27] for Java programs as the target and in [6] for monitors using OCL as the specification language.

A runtime verification framework that aims to provide high flexibility and reuse is presented in [55, 16]. Based on the experience gained at the NASA project Java PathExplorer [32], the authors propose an approach called *Monitoring Oriented Programming* (MOP). The idea is to integrate monitoring consequently into the development process by letting the specification and the implementation *together* form the system. It is a generalized software development and analyses framework that also supports runtime verification. Besides runtime verification, the authors emphasize two other perspectives of MOP: (1) an extension of programming languages with logics and (2) as a lightweight formal method. MOP supports the reuse of so-called *logic plug-ins*, for instance a plug-in for FTLTL (see Sec. 2.4.1 on page 21), by concrete language dependent instances of the MOP framework[4], e. g., Java-MOP for monitoring Java applications. A user of MOP can then specify a property together with actions written in the target language to handle validations and violations using one of the available logics. The concrete MOP tools automatically synthesize monitors from the specifications and integrate them together with the user-defined code into the application or into an external monitor. MOP supports different kinds of monitors. For example, a monitor can be in-line or out-line (see Sec. 2.4.2 on page 23), depending on the concrete use case. In contrast to our work, the definitions of the properties to verify in MOP is very specific to the target language and

[4] `http://fsl.cs.illinois.edu/mop`

can therefore not easily be used for different platforms. Another benefit of our approach is the application of USE, which provides a rich set of querying and visualization features in case of a violation, whereas in MOP the handling of violations is completely handled by user-defined code.

Another generic framework for on-line and off-line runtime monitoring is *Kieker*[5] [88, 87] developed jointly by researchers from Kiel University's Software Engineering Group and University of Stuttgart's Reliable Software Systems Group. Kieker can be used for different monitoring aspects, e. g., for application performance monitoring or architecture discovery. Collecting architectural information about existing software systems is supported by extracting call graphs or sequence diagrams, which is similar to our approach. Required architectural entities, like classes and operations, can be collected by using static and dynamic analysis. While Kieker focuses on Java-based systems, its extensible architecture allows to add custom logging facilities for different runtime environments, called monitoring probes in the context of this framework. These probes monitor, possibly distributed, traces of method executions and transfer monitoring records to the system core. The kind of records is defined by the concrete probe, but share a common base definition (technically a common base class). To support off-line analysis of recorded traces, Kieker includes several monitoring log writers, which can serialize received records for later use. For example, they can be stored using the file system or a relational database. Further, these writers can be either synchronous or asynchronous. Collected traces can be analyzed and visualized by different features, e. g., UML sequence diagrams, dynamic call trees, dependency diagrams, and Markov chains [87]. Additional features for analyzing monitored data can be added by plug-ins.

The *BeepBeep* tool that is specifically designed for the runtime verification of computer games is presented in [90, 89]. The authors observed that nearly every computer game is running a so-called *game-loop*. Based on this observation, they reason that injecting monitoring related code in that loop instead of into fine grained operation calls is adequate for runtime verification of computer games. This approach implies that the monitoring code supplies sequences of snapshots instead of event traces. This further changes the way, how the monitored properties are expressed. The authors use the terms *snapshot-based semantics* for defining properties over a sequence of snapshots and *event-based semantics* when they are expressed over an event trace. As stated before, BeepBeep requires a developer to introduce code that extracts required snapshot information each time a game-loop is passed through. Which information is required depends on the properties to be verified. BeepBeep expects snapshot data in an XML language, because it applies an own temporal language called LTL-FO$^+$, which uses XPath for path expressions. LTL-FO$^+$ is based on linear temporal logic (see Sec. 2.4.1 on page 21) and is extended by first order logic (FO) [89]. This approach differs to our approach in the sense that our approach does not need to inject custom code to retrieve the re-

[5] `http://kieker-monitoring.net`

quired data. Using our approach, one can also leave out fine grained operation calls, by excluding operations from the PAM. Further, BeepBeep currently lacks any support of visualization.

The Dresden OCL toolkit[6] makes available two distinctive approaches for OCL-based runtime verification [22]. While the generative approach injects AspectJ code, the interpretative one integrates the Dresden OCL Interpreter into a runtime environment in order to interpret OCL constraints.

One widely known language for assertion checking in Java programs is the Java modeling language (JML) [49]. While JML itself can be directly placed inside the source code, the approaches presented in [30] and [5] translate OCL expressions into JML. A tool that enforces OCL expressions by integrating them into Java byte-code is ocl2j [23]. As with the first monitoring approach employing USE [75] the integration is done using AspectJ.

[6] `http://www.dresden-ocl.org`

5 Summary of Additional Contributions

The work presented in this thesis so far represents the main research contributions by the author. Since the database systems group is researching in several directions in the context of model-driven development sometimes together with other researchers around the world, joined work has been published. The contribution of the author of this thesis to these publications is explained in the next paragraphs.

5.1 OCL Community

Around the OCL, a community of researchers has arisen. This can be seen by the number of OCL workshops, with the workshop held in 2013 [15] being the 13^{th}. Our research group contributes regularly to the OCL community. In the following, the work of the author in the context of the OCL community is summarized.

Like other languages, OCL faces the problem of a misleading or unclear specification. For this, our group proposed a benchmark for OCL tools to improve tool compatibility [A28J]. The proposed benchmark covers a wide range of OCL features. It starts with a *Core Benchmark* using class diagrams and OCL expressions. The core part starts by using OCL to navigate in simple class diagrams containing only basic UML elements, like for example, classes, binary associations, and attributes. It is extended in a stepwise manner to cover extended features, like ternary and qualified associations. While supporting the definition of the benchmark, the main contribution to this work was to collect and evaluate the benchmark results of different tools.

A detailed discussion of newly integrated collection types in OCL was done in [A4W]. Several discussed issues were encountered while the author integrated these new types into the USE tool. For the joined work in [A5W], developers and maintainers of OCL tools were asked to align their tools to a previously defined feature model of OCL tools [17]. This has been done by the author for the USE tool.

5.2 Model Validation and Model Finding

Formal model properties like consistency, independence of invariants and consequences have been studied in [A11C]. It was shown, how such properties of models can be semi-automatically examined in the USE tool by applying the generator and its language ASSL (see Sec. 2.3.1 beginning on page 14). For this model finding task, i. e., looking for

a valid model instance with defined properties, the generator needs to examine a large search space. In [A13C] we have shown how several techniques can be used to reduce the complexity of the search algorithm.

While this previous work was based on a procedural snapshot language, which requires a developer to specify detailed search information to be applicable, the work in [A27C], focused on a declarative approach that transforms UML/OCL models into representations usable by SAT-solvers. This transformation relies on a library for relational logic which in the end uses SAT-solvers to find a satisfying model instance.

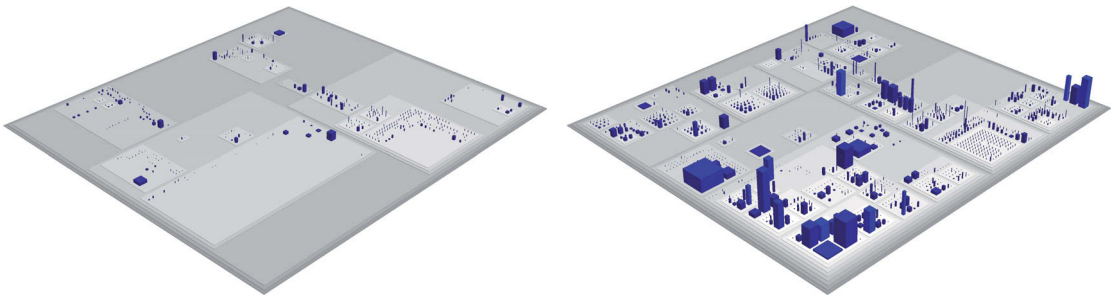
Work on applying this efficient solving technologies on behaviored models, i. e., models including pre- and postconditions, has been done in [A8C, A25C]. In this context, models including pre- and postconditions are called *application models*. These models are transformed into so-called *filmstrip models* that retain the behavior definitions by representing it as well-defined class structures including invariants only. After the transformation model-finding techniques can be used to find valid or invalid execution traces. In this work the author was highly involved into the development of the structure of the filmstrip models, as well as the conceptual definition of the applied transformation.

Supporting work of the author involved the development of the TRACTS tool chain, presented in [A29J]. This tool allows a developer to formally verify and test model transformations, which is similar to the approach presented in [A7W] and [A6W]. The former work applies ASSL (see Sec. 2.3 beginning on page 14) to generate test suites, which are used as an input for ATL [37] and, after executing the model transformation under test, are translated back to USE to validate the transformation. Whereas the latter work, solely applies USE and the USE model validator to test model transformations and to reason about transformation properties.

5.3 USE Applications and Extensions

USE is applied by different researchers for teaching and research purposes. Some of these usages lead to joint publications, like the aforementioned TRACTS tool chain [A29J] or the applications of USE to realize a declarative approach of describing workflows [A2W]. In both cases, the author of this thesis was involved in the development process.

Work on extensions to USE has been published in [A1C, A9C, A10W]. These extensions do not only improve the usability of USE, but can be integrated into other modeling tools as well to foster the usage of model-driven approaches. In [A1C] a technique for debugging and examining OCL expressions is described. It allows a user, to deeply dig into expressions by highlighting relevant aspects of the evaluation process. This extension allows for an easier correction of specification errors, since all evaluation steps can be accessed. [A10W] shows how OCL can be used to query graph like structures to identify relevant elements to show in a diagram. In addition, several other graphical user interface features, not only useful while working with UML class and object dia-



(a) USE 0.9.0 (b) USE 3.1.0

Figure 5.1: Visualization of the Evolution of USE

grams, are presented. [A9C] identifies new kinds of diagrams to support the process of identifying independence of invariants.

An experience report about the development of USE has been published in [A20W]. This report highlights key extensions to USE and reports about experiences on how to successfully retain high quality. Further, the evolution of USE is shown by means of city maps [93] as shown in Fig. 5.1. These maps visualize the structure of object-oriented source code by showing packages as districts containing their owned classes as buildings. The height and the width of each building are defined by the number of attributes and by the number of operations. As it can be seen by the different populations in the two city maps, which cover 15 years of development, USE has been growing noticeably.

5.4 Model-Driven Engineering in the Context of eGovernment

For several years, the author was involved in joint work together with a standardization organization of the German government called KoSIT [44, 14].

The joint work started with a model-driven approach of standardizing the data exchange between registration offices in Germany (called XMeld). After this starting project, other areas of the German public authorities, like justice and civil status registrations started to use this approach. This interest in the MDE approach lead to a generalized version of the framework called XÖV (*XML in public authorities* or in German *XML in der öffentlichen Verwaltung*). A central component of this MDE approach is a model-to-text transformation application called XGenerator. The XGenerator itself applies the USE tool as a validation and query component for UML models. Therefore, some of the previously presented work on USE was directly applied to this project as well. A report about the successful application of MDE in the context of eGovernment was presented in [A3J]. Some insights about the usage of OCL in this context are reported in [A26W]. Further, the question of how to apply automated tests on the model level in this context was addressed in [A14W].

6 Conclusion and Future Work

This thesis summarizes and relates the two main research contributions of the author: work on defining precise meanings to elements of the UML and applying UML and OCL to runtime verification.

6.1 Conclusion

In the first part, an approach has been presented that allows to define a precise semantics of the UML by using itself together with OCL. The main advantages of this approach are that it does not introduce another formalism and that the defined semantics can be validated by using existing tools. We exemplified the approach by picking out central elements of the UML specification that are widely used, not only to specify the UML metamodel, but are also used by other modeling languages. We have further shown, how the support of derived modeling elements can ease the understanding of larger models. For this, we identified requirements for an evaluator and for the derive expressions to be able to compute them at runtime. At the end of this part, we presented how the integration of protocol state machines can foster the definition of behavioral models.

The second part of this thesis recapped work on runtime verification using UML and OCL. This work supports a model-driven development process in the sense, that it makes models an important artifact during the implementation phase, even if they only partly define the resulting system. This allows for the verification of the manually written implementations against properties that were modeled in an early phase of design. A mid-sized real word application was used to highlight the possibilities of our approach. In addition, we have shown, how our approach can easily be reused to validate business rules on a non object-oriented platform, namely a relational database. In more detail than the published work, we discussed abstraction concepts our approach supports and limitations as well as possible solutions for them.

Both parts of this thesis influenced each other. For instance, during the work on runtime verification, a need for protocol state machines was identified. Their integration into the USE tool has not only increased the features for monitoring, but also extended the modeling capabilities of USE. In the other direction, work on the semantics of relations between properties and the support for derived properties added further capabilities for the runtime verification approach.

Finally, additional publication of the author were summarized. Most of them are results of the development of USE. For example, ambiguities in the specifications of UML

and OCL were encountered or the application of USE in other research areas, e. g., model finding or verification of model transformations, lead to new research contributions.

6.2 Future Work

All of the presented work can be extended in many directions. At first, a broader coverage of the runtime semantics of the UML would strengthen its usability. For this, the semantic domain model needs to be extended by adding new elements and constraints. Providing a runtime semantics for state machines using our approach might be a starting point for this. To be able to do so, a possibility to evaluate constraints defined in the user model, e. g., transition guards, on the level of a metamodel instance needs to be developed.

For the USE tool, several extensions to improve its capabilities as a model validation tool are possible. An integration of behavioral state machines would highly increase the capabilities of modeling behavior. This would also lead to the ability to validate asynchronous message flows. Further, it might show ambiguities or missing elements in the OCL specification in the context of messaging. For this, SOIL would need to be extended to achieve an Executable UML like modeling environment covering full OCL support. Working with complex behavioral models would raise the need for debugging capabilities for executed models, which are currently absent in USE.

To extend the support of model transformations, a mechanism for the navigation on the metamodel level of UML and OCL models could be developed. This would allow for an easier definition of generic constraints, since they could be specified on the meta level and could be applied to all user models. To gain higher reuse for USE models the possibility to extend the OCL library by defining new operations for built-in data types and by adding new user defined data types would be useful. This would imply an extension of OCL, which allows for a usage of template parameters in operation definitions.

Also for the USE monitor several possibilities of improvement exist. First, additional adapters for other platforms would not only increase its usefulness, but could also reveal new extensions to the monitor. Additional adapters for already covered target platforms that use other injection mechanism would allow for a comparison of the different mechanisms. The difficulties encountered while using state invariants to determine the state of monitored instances could be reduced by adding specific state determination expressions that are used by the monitor. A promising research direction for these state determination expressions would be to examine how the work done for proving global invariant independence needs to be aligned to prove the independence of such expressions for a single object.

Bibliography of the Author

[A1C] Jens Brüning, Martin Gogolla, Lars Hamann, and Mirco Kuhlmann. Evaluating and Debugging OCL Expressions in UML Models. In Achim D. Brucker and Jacques Julliand, editors, *Proc. 6th Int. Conf. Tests and Proofs (TAP'2012)*, pages 156–162. Springer, Berlin, LNCS 7305, 2012.

[A2W] Jens Brüning, Lars Hamann, and Andreas Wolff. Extending ASSL: Making UML Metamodel-based Workflows Executable. In Jordi Cabot, Robert Clariso, Martin Gogolla, and Burkhart Wolff, editors, *Proc. Workshop OCL and Textual Modelling (OCL'2011)*. ECEASST, Electronic Communications, `http://journal.ub.tu-berlin.de/eceasst/issue/view/56`, 2011.

[A3J] Fabian Büttner, Ullrich Bartels, Lars Hamann, Oliver Hofrichter, Mirco Kuhlmann, Martin Gogolla, Lutz Rabe, Frank Steimke, Yorck Rabenstein, and Alina Stosiek. Model-Driven Standardization of Public Authority Data Interchange. *Science of Computer Programming*, 89:162–175, 2014.

[A4W] Fabian Büttner, Martin Gogolla, Lars Hamann, Mirco Kuhlmann, and Arne Lindow. On Better Understanding OCL Collections *or* An OCL Ordered Set Is Not an OCL Set. In Sudipto Ghosh, editor, *Workshops and Symposia at 12th Int. Conf. Model Driven Engineering Languages and Systems (MODELS'2009)*, pages 276–290. Springer, Berlin, LNCS 6002, 2010.

[A5W] Joanna Dobroslawa Chimiak-Opoka, Birgit Demuth, Andreas Awenius, Dan Chiorean, Sebastien Gabel, Lars Hamann, and Edward Willink. OCL Tools Report based on the IDE4OCL Feature Model. In Jordi Cabot, Robert Clariso, Martin Gogolla, and Burkhart Wolff, editors, *Proc. Workshop OCL and Textual Modelling (OCL'2011)*. ECEASST, Electronic Communications, `http://journal.ub.tu-berlin.de/eceasst/issue/view/56`, 2011.

[A6W] Martin Gogolla, Lars Hamann, and Frank Hilken. Checking Transformation Model Properties with a UML and OCL Model Validator. In Moussa Amrani, Eugene Syriani, and Manuel Wimmer, editors, *Proc. 3rd Int. Workshop on Verification of Model Transformation (VOLT'2014)*, `http://ceur-ws.org/`, To appear, 2014. CEUR Proceedings.

[A7W] Martin Gogolla, Lars Hamann, and Frank Hilken. On Static and Dynamic Analysis of UML and OCL Transformation Models. In Jürgen Dingel, Juan de Lara, Levi Lucio, and Hans Vangheluwe, editors, *Proc. Int. Workshop on Analysis of Model Transformations (AMT'2014)*, pages 24–33, `http://ceur-ws.org/Vol-1277/`, 2014. CEUR Proceedings, Vol. 1277.

[A8C] Martin Gogolla, Lars Hamann, Frank Hilken, Mirco Kuhlmann, and Robert B. France. From Application Models to Filmstrip Models: An Approach to Automatic Validation of Model Dynamics. In Hans-Georg Fill, Dimitris Karagiannis, and Ulrich Reimer, editors, *Proc. Modellierung (MODELLIERUNG'2014)*, pages 273–288. GI, LNI 225, 2014.

[A9C] Martin Gogolla, Lars Hamann, and Mirco Kuhlmann. Proving and Visualizing OCL Invariant Independence by Automatically Generated Test Cases. In Gordon Fraser and Angelo Gargantini, editors, *Proc. 4th Int. Conf. Test and Proof (TAP'2010)*, pages 38–54. Springer, Berlin, LNCS 6143, 2010.

[A10W] Martin Gogolla, Lars Hamann, Jie Xu, and Jun Zhang. Exploring (Meta-)Model Snapshots by Combining Visual and Textual Techniques. In Fabio Gadducci and Leonardo Mariani, editors, *Proc. Workshop Graph Transformation and Visual Modeling Techniques (GTVMT'2011)*. ECEASST, Electronic Communications, `http://journal.ub.tu-berlin.de/eceasst/issue/view/53`, 2011.

[A11C] Martin Gogolla, Mirco Kuhlmann, and Lars Hamann. Consistency, Independence and Consequences in UML and OCL Models. In Catherine Dubois, editor, *Proc. 3rd Int. Conf. Test and Proof (TAP'2009)*, pages 90–104. Springer, Berlin, LNCS 5668, 2009.

[A12W] Martin Gogolla, Matthias Sedlmeier, Lars Hamann, and Frank Hilken. On Metamodel Superstructures Employing UML Generalization Features. In Colin Atkinson, Georg Grossmann, Thomas Kühne, and Juan de Lara, editors, *Proc. Int. Workshop on Multi-Level Modelling (MULTI'2014)*, pages 13–22, `http://ceur-ws.org/Vol-1286/`, 2014. CEUR Proceedings, Vol. 1286.

[A13C] Lars Hamann, Fabian Büttner, Mirco Kuhlmann, and Martin Gogolla. Optimierte Suche von Modellinstanzen für UML/OCL-Beschreibungen in USE. In Elmar J. Sinz and Andy Schürr, editors, *Proc. Modellierung (MODELLIERUNG'2012)*, pages 155–170. Springer, LNI 201, 2012.

[A14W] Lars Hamann and Martin Gogolla. Improving Model Quality by Validating Constraints with Model Unit Tests. In Levi Lucio, Elisangela Vieira, and Stephan Weissleder, editors, *Proc. Workshop on Model-Driven Engineering, Verification, and Validation (MODEVVA'2010)*, pages 49–55. IEEE, 2010.

[A15C] Lars Hamann and Martin Gogolla. Endogenous Metamodeling Semantics for Structural UML 2 Concepts. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke, editors, *Proc. 16th Int. Conf. Model-Driven Engineering Languages and Systems (MoDELS'2013), Miami, FL, USA*, pages 488–504. Springer, Berlin, LNCS 8107, 2013.

[A16C] Lars Hamann, Martin Gogolla, and Oliver Hofrichter. Zur Integration von Struktur- und Verhaltensmodellierung mit OCL. In Wilhelm Hasselbring and Nils Christian Ehmke, editors, *Proc. Software Engineering (SE'2014)*, pages 75–76. GI, LNI 227, 2014.

[A17W] Lars Hamann, Martin Gogolla, and Daniel Honsel. Towards Supporting Multiple Execution Environments for UML/OCL Models at Runtime. In Nelly Bencomo, Gordon Blair, Sebastian Götz, Brice Morin, and Bernhard Rumpe, editors, *Proc. 7th Int. Workshop Models at Runtime (MRT'2012)*, pages 46–51. ACM Digital Library, 2012.

[A18C] Lars Hamann, Martin Gogolla, and Mirco Kuhlmann. Zur Validierung von Kompositionsstrukturen in UML mit USE. In Gregor Engels, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Proc. Modellierung (MODELLIERUNG'2010)*, pages 169–177. GI, LNI 161, 2010.

[A19W] Lars Hamann, Martin Gogolla, and Mirco Kuhlmann. OCL-Based Runtime Monitoring of JVM Hosted Applications. In Jordi Cabot, Robert Clariso, Martin Gogolla, and Burkhart Wolff, editors, *Proc. Workshop OCL and Textual Modelling (OCL'2011)*. ECEASST, Electronic Communications, `http://journal.ub.tu-berlin.de/eceasst/issue/view/56`, 2011.

[A20W] Lars Hamann, Frank Hilken, and Martin Gogolla. Collected Experience and Thoughts on Long Term Development of an Open Source MDE Tool. In Francis Bordelau, Jürgen Dingel, Sebastien Gerard, and Sebastian Voss, editors, *Proc. Int. Workshop on Open Source Software for Model Driven Engineering (OSS4MDE'2014)*, pages 42–52, `http://ceur-ws.org/Vol-1290/`, 2014. CEUR Proceedings, Vol. 1290.

[A21C] Lars Hamann, Oliver Hofrichter, and Martin Gogolla. OCL-Based Runtime Monitoring of Applications with Protocol State Machines. In Antonio Vallecillo and Juha-Pekka Tolvanen, editors, *Proc. 8th European Conf. Modelling Foundations and Applications (ECMFA'2012)*, pages 384–399. Springer, Berlin, LNCS 7349, 2012.

[A22C] Lars Hamann, Oliver Hofrichter, and Martin Gogolla. On Integrating Structure and Behavior Modeling with OCL. In Robert France, Juergen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Proc. 15th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'2012), Innsbruck, Austria*, pages 235–251. Springer, Berlin, LNCS 7590, 2012.

[A23C] Lars Hamann, Karsten Sohr, and Martin Gogolla. Monitoring Database Access Constraints with an RBAC Metamodel: a Feasibility Study. In *Proc. International Symposium on Engineering Secure Software and Systems (ESSoS'2015).* Springer, Berlin, LNCS 8978, 2015. To be published.

[A24C] Lars Hamann, Laszlo Vidacs, Martin Gogolla, and Mirco Kuhlmann. Abstract Runtime Monitoring with USE. In Tom Mens, Anthony Cleve, and Rudolf Ferenc, editors, *Proc. European Conf. Software Maintenance and Reengineering (CSMR'2012)*, pages 549–552. IEEE, 2012.

[A25C] Frank Hilken, Lars Hamann, and Martin Gogolla. Transformation of UML and OCL Models into Filmstrip Models. In Davide Di Ruscio and Dániel Varró, editors, *Proc. 7th Int. Conf. Model Transformation (ICMT'2014)*, pages 170–185. Springer, LNCS 8568, 2014.

[A26W] Oliver Hofrichter, Lars Hamann, Martin Gogolla, and Frank Steimke. The Secret Life of OCL Constraints. In Mira Balaban, Jordi Cabot, Martin Gogolla, and Claas Wilke, editors, *Proc. 12th Int. Workshop Object Constraint Language (OCL'2012)*, pages 63–64. ACM Digital Library, 2012.

[A27C] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. Extensive Validation of OCL Models by Integrating SAT Solving into USE. In Judith Bishop and Antonio Vallecillo, editors, *Proc. 49th Int. Conf. Objects, Models, Components, and Patterns (TOOLS'2011)*, pages 289–305. Springer, Berlin, LNCS 6705, 2011.

[A28J] Mirco Kuhlmann, Lars Hamann, Martin Gogolla, and Fabian Büttner. A Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency. *Software and Systems Modeling*, 11(2):165–182, 2012.

[A29J] Antonio Vallecillo, Martin Gogolla, Loli Burgueno, Manuel Wimmer, and Lars Hamann. Formal Specification and Testing of Model Transformations. In Marco Bernardo, Vittorio Cortellessa, and Alphonso Pierantonio, editors, *Proc. 12th Int. School Formal Methods for the Design of Computer, Communication and Software Systems: Model-Driven Engineering*, pages 399–437. Springer, Berlin, LNCS 7320, 2012.

Bibliography

[1] Marcus Alanen and Ivan Porres. A Metamodeling Language Supporting Subset and Union Properties. *Software and System Modeling*, 7(1):103–124, 2008.

[2] Carsten Amelunxen. *Metamodel-based Design Rule Checking and Enforcement*. PhD thesis, Technische Universität Darmstadt, 2009.

[3] Carsten Amelunxen and Andy Schürr. Formalizing Model Transformation Rules for UML/MOF 2. *IET Software Journal*, 2(3):204–222, June 2008. Special Issue: Language Engineering.

[4] Roman Asendorf. Entwicklung einer Plugin-Architektur für USE. Master's thesis, Universität Bremen, 2009.

[5] Carmen Avila, Guillermo Flores, and Yoonsik Cheon. A Library-Based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking. In Hamid R. Arabnia and Hassan Reza, editors, *Proceedings of the International Conference on Software Engineering Research & Practice (SERP'2008)*, pages 403–408. CSREA Press, 2008.

[6] Carmen Avila, Amritam Sarcar, Yoonsik Cheon, and Cesar Yeep. Runtime Constraint Checking Approaches for OCL, A Critical Comparison. In *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010)*, pages 393–398. Knowledge Systems Institute Graduate School, 2010.

[7] Helmut Balzert. *Lehrbuch der Softwaretechnik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum, Akademischer Verlag, 1997.

[8] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking. In *Advances in Computers*, volume 58, pages 117 – 148. Elsevier, 2003.

[9] Conrad Bock. UML 2 Composition Model. *Journal of Object Technology*, 3(10):47–73, December 2004.

[10] Grady Booch. *Object-oriented Analysis and Design with Applications (2Nd Ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[11] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. Modeling the Dynamics of UML State Machines. In Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, editors, *Proceedings of International Workshop on Abstract State Machines, Theory and Applications (ASM'2000)*, pages 223–241. Springer, Berlin, LNCS 1912, 2000.

[12] Manfred Broy and María Victoria Cengarle. UML Formal Semantics: Lessons Learned. *Software and System Modeling*, 10(4):441–446, 2011.

[13] Fabian Büttner. *Reusing OCL in the Definition of Imperative Languages*. PhD thesis, Universität Bremen, Fachbereich Mathematik und Informatik, 2010.

[14] Fabian Büttner, Mirco Kuhlmann, Martin Gogolla, Jens Dietrich, Frank Steimke, Andre Pankratz, Alina Stosiek, and Alexander Salomon. MDA Employed in a Joint eGovernment Strategy: An Experience Report. In Terry Bailey, editor, *Proc. 3rd ECMDA Workshop "From Code Centric To Model Centric Software Engineering" (2008)*. European Software Institute, 2008.

[15] Jordi Cabot, Martin Gogolla, István Ráth, and Edward D. Willink, editors. *Proceedings of the MODELS 2013 OCL Workshop co-located with the 16th International ACM/IEEE Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, September 30, 2013*, volume 1092 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[16] Feng Chen, Marcelo d'Amorim, and Grigore Roşu. A Formal Monitoring-Based Framework for Software Development and Analysis. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *Proceedings of 6th International Conference on Formal Engineering Methods (ICFEM'2004)*, pages 357–372. Springer, Berlin, LNCS 3308, 2004.

[17] Joanna Dobroslawa Chimiak-Opoka and Birgit Demuth. A Feature Model for an IDE4OCL. In Jordi Cabot, Tony Clark, Manuel Clavel, Martin Gogolla, editor, *Proceedings of the Workshop on OCL and Textual Modelling (OCL'2010)*, volume 36. ECEASST, 2010.

[18] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, Mass., 6. edition, 2008.

[19] Séverine Colin and Leonardo Mariani. Run-Time Verification. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, pages 525–555. Springer, Berlin, LNCS 3472, 2005.

[20] Dolors Costal, Cristina Gómez, and Giancarlo Guizzardi. Formal Semantics and Ontological Analysis for Understanding Subsetting, Specialization and Redefinition of Associations in UML. In Manfred Jeusfeld, Lois Delcambre, and Tok-Wang Ling, editors, *Proceedings of 30th International Conference on Conceptual Modeling (ER'2011)*, pages 189–203. Springer, Berlin, LNCS 6998, 2011.

[21] John Deacon. *Object-oriented Analysis and Design: A Pragmatic Approach*. Pearson Addison Wesley, 2005.

[22] Birgit Demuth and Claas Wilke. Model and Object Verification by Using Dresden OCL. In *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice*, pages 687–690, Ufa, Russia, 2009.

[23] Wojciech J. Dzidek, Lionel C. Briand, and Yvan Labiche. Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In *Satellite Events at the MoDELS 2005 Conference, Revised Selected Papers*, pages 10–19. Springer, Berlin, LNCS 3844, 2006.

[24] Carles Farré, Anna Queralt, Guillem Rull, Ernest Teniente, and Toni Urpí. Automated Reasoning on UML Conceptual Schemas with Derived Information and Queries. *Information & Software Technology*, 55(9):1529–1550, 2013.

[25] Stephan Flake and Wolfgang Müller. Formal Semantics of Static and Temporal State-oriented OCL Constraints. *Software and System Modeling*, 2(3):164–186, 2003.

[26] David Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[27] Lorenz Froihofer, Gerhard Glos, Johannes Osrael, and Karl M. Goeschka. Overview and Evaluation of Constraint Validation Approaches in Java. In *Proceedings of 29th International Conference on Software Engineering (ICSE'2007)*, pages 313–322, Washington, DC, USA, 2007. IEEE Computer Society.

[28] Martin Gogolla, Jörn Bohling, and Mark Richters. Validation of UML and OCL Models by Automatic Snapshot Generation. In Grady Booch, Perdita Stevens, and Jonathan Whittle, editors, *Proceedings of the 6th International Conference Unified Modeling Language (UML'2003)*, pages 265–279. Springer, Berlin, LNCS 2863, 2003.

[29] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69:27–34, 2007.

[30] Ali Hamie. Translating the Object Constraint Language into the Java Modelling Language. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC'2004)*, pages 1531–1535, New York, NY, USA, 2004. ACM.

[31] Klaus Havelund and Thomas Pressburger. Model Checking JAVA Programs Using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[32] Klaus Havelund and Grigore Roşu. Monitoring Java Programs with Java PathExplorer. *Electronic Notes in Theoretical Computer Science*, 55(2):200 – 217, 2001. Special Issue: RV'2001, Runtime Verification (in connection with CAV'01).

[33] Daniel Honsel. Technologieübergreifende Verifikation von Laufzeit-Annahmen mit UML- und OCL-Modellen. Master's thesis, Universität Bremen, August 2013.

[34] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[35] Ivar Jacobson. Object-Oriented Software Engineering - a Use Case Driven Approach. In Boris Magnusson, Bertrand Meyer, and Jean-François Perrot, editors, *Proceedings of 10th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'1993)*, page 333. Prentice Hall, 1993.

[36] Jun Zhang Jie Xu. Komfortable Darstellung von und komplexe Selektion in UML-Klassen- und Objektdiagrammen. Master's thesis, Universität Bremen, 2007.

[37] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

[38] Frederick P. Brooks Jr. No Silver Bullet - Essence and Accidents of Software Engineering (Invited Paper). In *IFIP Congress*, pages 1069–1076, 1986.

[39] Frederick P. Brooks Jr. *The Mythical Man-Month - Essays on Software Engineering (2. Ed.)*. Addison-Wesley, 1995.

[40] Joost-Pieter Katoen and Christel Baier. *Principles of Model Checking*. MIT Press, Cambridge, Mass., 2008.

[41] Alfons Kemper and André Eickler. *Datenbanksysteme*. Oldenbourg Wissenschaftsverlag, München, 5. edition, 2004.

[42] Anneke Kleppe. Object Constraint Language: Metamodeling Semantics. In Kevin Lano, editor, *UML 2 Semantics and Applications*, pages 163–178. John Wiley & Sons, Inc., 2009.

[43] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[44] Die Koordinierungsstelle für IT-Standards (KoSIT). `http://www.it-planungsrat.de/DE/Organisation/KoSIT/KoSIT_node.html`.

[45] Mirco Kuhlmann and Martin Gogolla. From UML and OCL to Relational Logic and Back. In Robert France, Juergen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (MoDELS'2012)*, pages 415–431. Springer, Berlin, LNCS 7590, 2012.

[46] Orna Kupferman and Sharon Zuhovitzky. An Improved Algorithm for the Membership Problem for Extended Regular Expressions. In Krzysztof Diks and Wojciech Rytter, editors, *Proceedings of 27th International Symposium Mathematical Foundations of Computer Science (MFCS'2002)*, pages 446–458. Springer, Berlin, LNCS 2420, 2002.

[47] Kevin Lano. *UML 2 Semantics and Applications*. John Wiley & Sons, Inc., 2009.

[48] Kevin Lano and David Clark. Semantics and Refinement of Behavior State Machines. In José Cordeiro and Joaquim Filipe, editors, *Proceedings of the 10th International Conference on Enterprise Information Systems (ICEIS'2008)*, pages 42–49, 2008.

[49] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, 2005.

[50] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009. Proceedings of 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[51] Felipe Lopez. Fallstudien zum Monitoring von Java-Anwendungen mit UML und OCL. Master's thesis, Universität Bremen, 2012.

[52] Azzam Maraee and Mira Balaban. Inter-association Constraints in UML2: Comparative Analysis, Usage Recommendations, and Modeling Guidelines. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS'2012)*, pages 302–318. Springer, Berlin, LNCS 7590, 2012.

[53] Nicolas Markey. Temporal Logic with Past is Exponentially more Succinct. *Bulletin of the EATCS*, 79:122–128, 2003.

[54] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley, 2002.

[55] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An Overview of the MOP Runtime Verification Framework. *International Journal on Software Techniques for Technology Transfer*, 14(3):249–289, 2012.

[56] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[57] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In Lionel C. Briand and Clay Williams, editors, *Proceedings of 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'2005)*, pages 264–278. Springer, Berlin, LNCS 3713, 2005.

[58] Quang Dung Nguyen. Integration von Objektzuständen aus Statecharts in Sequenzdiagramme und Implementierung von Kommunikationsdiagrammen in USE. Master's thesis, Universität Bremen, 2014.

[59] Pilar Nieto, Dolors Costal, and Cristina Gómez. Enhancing the Semantics of UML Association Redefinition. *Data Knowledge Engineering*, 70(2):182–207, 2011.

[60] Object Management Group (OMG). UML Human-Usable Textual Notation (HUTN). `http://www.omg.org/spec/HUTN/`, August 2004.

[61] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.1. `http://www.omg.org/spec/QVT/1.1/`, January 2011.

[62] Object Management Group (OMG). Meta Object Facility (MOF) Core Specification 2.4.1. `http://www.omg.org/spec/MOF/2.4.1`, August 2011.

[63] Object Management Group (OMG). Object Constraint Language 2.3.1. `http://www.omg.org/spec/OCL/2.3.1/`, January 2012.

[64] Object Management Group (OMG). Concrete Syntax for a UML Action Language: Action Language for Foundational UML (ALF). `http://www.omg.org/spec/ALF/`.

[65] Object Management Group (OMG). Model Driven Architecture (MDA). `http://www.omg.org/mda/`.

[66] Object Management Group (OMG). Semantics of a Foundational Subset for Executable UML Models (FUML). `http://www.omg.org/spec/FUML`.

[67] Object Management Group (OMG). UML Specifications. `http://www.omg.org/spec/UML`.

[68] Object Management Group (OMG). UML Infrastructure 2.4.1. `http://www.omg.org/spec/UML/2.4.1/Infrastructure`, August 2011.

[69] Object Management Group (OMG). UML Superstructure 2.4.1. `http://www.omg.org/spec/UML/2.4.1/Superstructure`, August 2011.

[70] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of 18th Annual Symposium on Foundations of Computer Science (FOCS'1977)*, pages 46–57. IEEE Computer Society, 1977.

[71] Ivan Porres and Irum Rauf. From Nondeterministic UML Protocol Statemachines to Class Contracts. In *Proceedings of 3rd International Conference on Software Testing, Verification and Validation (ICST'2010)*, pages 107–116. IEEE Computer Society, 2010.

[72] Elaine Rich. *Automata, Computability and Complexity: Theory and Applications*. Pearson Prentice Hall, 2008.

[73] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Fachbereich Mathematik und Informatik, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[74] Mark Richters and Martin Gogolla. On Formalizing the UML Object Constraint Language OCL. In Tok-Wang Ling, Sudha Ram, and Mong Li Lee, editors, *Proceedings of 17th International Conference Conceptual Modeling (ER'1998)*, pages 449–464. Springer, Berlin, LNCS 1507, 1998.

[75] Mark Richters and Martin Gogolla. Aspect-Oriented Monitoring of UML and OCL Constraints. In Omar Aldawud, Mohamed Kande, Grady Booch, Bill Harrison, Dominik Stein, Jeff Gray, Siobhan Clarke, Aida Zakaria, Peri Tarr, and Faisal Akkawi, editors, *Proceedings of UML'2003 Workshop Aspect-Oriented Software Development with UML*. Illinois Institute of Technology, Department of Computer Science, 2003.

[76] Grigore Roşu and Klaus Havelund. Rewriting-Based Techniques for Runtime Verification. *Automated Software Engineering*, 12(2):151–197, 2005.

[77] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented Modeling and Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[78] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2. edition, 2004.

[79] Bernhard Rumpe and Robert B. France. Variability in UML Language and Semantics. *Software and System Modeling*, 10(4):439–440, 2011.

[80] Koushik Sen and Grigore Roşu. Generating Optimal Monitors for Extended Regular Expressions. *Electronic Notes in Theoretical Computer Science*, 89(2):226–245, 2003.

[81] Lijun Shan and Hong Zhu. Unifying the Semantics of Models and Meta-Models in the Multi-Layered UML Meta-Modelling Hierarchy. *Software and Informatics*, 6(2):163–200, 2012.

[82] Sally Shlaer and Stephen J. Mellor. *Object Lifecycles: Modeling the World in States.* Yourdon Press, EngleWood Cliffs, NJ, 1992.

[83] Sally Shlaer and Stephen J. Mellor. *Object-Oriented Systems Analysis: Modelling the World in Data.* Yourdon Press, EngleWood Cliffs, NJ, 1992.

[84] Dilek Stadtler and Friedrich Steimann. Wie die Objektorientierung relationaler werden sollte: Eine Analyse aus Sicht der Datenmodellierung. In Gregor Engels, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Proceedings of Modellierung 2010*, pages 149–167. GI, LNI 161, 2010.

[85] Marco Torchiano, Federico Tomassetti, Filippo Ricca, Alessandro Tiso, and Gianna Reggio. Relevance, Benefits, and Problems of Software Modelling and Model Driven Techniques - A Survey in the Italian Industry. *Journal of Systems and Software*, 86(8):2110–2126, 2013.

[86] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *Proceedings of 13th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2007)*, pages 632–647. Springer, Berlin, LNCS 4424, 2007.

[87] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Technical Report TR-0921, Department of Computer Science, Kiel University, Germany, November 2009.

[88] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE'2012)*, pages 247–248. ACM, April 2012.

[89] Simon Varvaressos, Kim Lavoie, Alexandre Blondin Masse, Sebastien Gaboury, and Sylvain Halle. Automated Bug Finding in Video Games: A Case Study for Runtime

Monitoring. In *Proceedings of 7th International Conference on Software Testing, Verification and Validation (ICST'2014)*, pages 143–152. IEEE, March 2014.

[90] Simon Varvaressos, Dominic Vaillancourt, Sébastien Gaboury, Alexandre Blondin Massé, and Sylvain Hallé. Runtime Monitoring of Temporal Logic Properties in a Platform Game. In Axel Legay and Saddek Bensalem, editors, *Proceedings of 4th International Conference on Runtime Verification (RV'2013)*, pages 346–351. Springer, Berlin, LNCS 8174, 2013.

[91] D.R. Wallace and R.U. Fujii. Software verification and validation: an overview. *Software*, 6(3):10–17, May 1989.

[92] Jos Warmer and Anneke Kleppe. *Object Constraint Language 2.0*. mitp, 2004.

[93] Richard Wettel and Michele Lanza. Visual Exploration of Large-Scale System Evolution. In Ahmed E. Hassan, Andy Zaidman, and Massimiliano Di Penta, editors, *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE'2008)*, pages 219–228. IEEE, 2008.

Publication A15C

Endogenous Metamodeling Semantics for Structural UML 2 Concepts

Authors: *Lars Hamann and Martin Gogolla*

Proc. 16th International Conference Model-Driven Engineering Languages and Systems (MoDELS'2013)

The final publication is available at Springer via
`http://dx.doi.org/10.1007/978-3-642-41533-3_30`

Endogenous Metamodeling Semantics for Structural UML 2 Concepts

Lars Hamann and Martin Gogolla

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany
{lhamann,gogolla}@informatik.uni-bremen.de
http://www.db.informatik.uni-bremen.de

Abstract. A lot of work has been done in order to put the Unified Modeling Language (UML) on a formal basis by translating concepts into various formal languages, e.g., set theory or graph transformation. While the abstract UML syntax is defined by using an endogenous approach, i. e., UML describes its abstract syntax using UML, this approach is rarely used for its semantics. This paper shows how to apply an endogenous approach called metamodeling semantics for central parts of the UML standard. To this end, we enrich existing UML language elements with constraints specified in the Object Constraint Language (OCL) in order to describe a semantic domain model. The UML specification explicitly states that complete runtime semantics is not included in the standard because it would be a major amount of work. However, we believe that certain central concepts, like the ones used in the UML standard and in particular property features as subsets, union and derived, need to be explicitly modeled to enforce a common understanding. Using such an endogenous approach enables the validation and verification of the UML standard by using off-the-shelf UML and OCL tools.

Keywords: Metamodeling, Semantics, Validation, UML, OCL

1 Introduction

In order to describe the abstract syntax of modeling languages, well-known concepts like classes, associations, and inheritance are used to express the structure of a language. These elements are commonly used in combination with a textual language to express further well-formedness rules which cannot be expressed using a graphical syntax. To improve the expressiveness of graphical modeling languages, especially when using complex inheritance relations, additional annotations have been developed to express more detailed information about the relation between model elements. Examples of these annotations are the subsets relations between properties and tagging a property as a derived union. The abstract syntax definition of the UML [23, 26] uses these newer modeling elements

since UML 2. Such a distinguished usage calls for the need of a precise definition at the syntax level (design time) and also on the semantic level (runtime)[1].

In this paper, we present an endogenous approach to specify the syntax and the semantics of central concepts of modeling languages. To this end, we use the same formalism, i. e., class diagrams enriched with constraints expressed in the Object Constraint Language (OCL) [24, 32], as used currently for the syntax description of modeling languages. To demonstrate our approach we choose particular UML language features (subsets, union and derived), but the same method may be applied to all UML language elements. The language features we choose are also important on their own, because they are used in MOF (i. e. as a description language for UML) without having a proper formal semantics currently. Our work is different to other approaches, like for example [1, 19], that define a formal semantics for the modeling elements mentioned above, in the sense, that we use the same languages to describe the syntax and the semantics instead of translating syntactical elements into a different formalism.

The rest of this work is structured as follows: In the next section we describe the concept of metamodeling semantics. In Sect. 3 we explain our approach for metamodeling the runtime semantics of modeling elements by using well-known examples. Section 4 identifies benefits arising when using tool-based validation of modeling concepts. Before the paper ends with a conclusion and future work, we discuss related approaches in Sect. 5.

2 Metamodeling Semantics

The notion *Metamodeling Semantics* can be explained well by quoting a statement from [16]:

> Metamodeling semantics is a way to describe semantics that is similar to the way in which popular languages like UML are defined. In metamodeling semantics, not only the abstract syntax of the language, but also the semantic domain, is specified using a model.

Metamodeling a language by defining the abstract syntax using a graphical modeling language combined with a formal textual language to express well-formedness rules is a well-known technique. The UML specification for example uses UML (or MOF which itself uses UML) in combination with the Object Constraint Language (OCL) to define its abstract syntax. In [16] this is called the *Abstract Syntax Model (ASM)*, which defines the valid structures in a model. The same technique is rarely used to define the semantics of a language, i. e., to specify a *Semantic Domain Model (SDM)* of a modeling language. A *semantic domain* defines the meaning of a particular language feature, whereas a semantic domain model describes this meaning by modeling the runtime behavior of a (syntactically) valid model using its runtime values and applying meaning to

[1] In this work, we distinguish between design time and runtime by using classes and objects. Note, that this distinction is not always appropriate.

them. For example, later we will see that in the UML there is the class `Class` in the abstract syntax part, and there is the class `InstancesSpecification` in the semantic domain part which together can describe (through an appropriate association) that a class (introduced at design time) is interpreted (at runtime) by a set of objects, formally captured as instance specifications. Another publicly available example for metamodeling semantics can be found in Section 10 of the OCL specification [24]. It defines constraints on values, i. e., runtime instances, which are part of the SDM. For example, the runtime value of a set is constrained as follows:

```
context SetTypeValue inv: self.element->isUnique(e : Element | e.value)
```

The central idea behind the approach in [24] is to describe the runtime behavior of OCL using OCL, which is similar to the UML metamodel described by UML models. While this is done in the UML to constrain the metamodel level M1, i. e., the valid structure of models, very little formal information is given for the level M0. Nearly only, the structure for the runtime snapshots is specified, but little use is made of defining runtime constraints in a formal language like OCL. An excerpt of the UML metamodel which shows important elements for our work is shown in Fig. 1. The diagram combines elements from roughly six syntax diagrams of the UML metamodel. On the left side, the ASM (syntax) of the UML is shown. On the right, the SDM (semantics) elements are given as they are present in the current specification. In the next section we define runtime constraints on the semantic domain model for several modeling constructs which are frequently used in the definition of the UML metamodel, but are only defined in an informal way with verbal descriptions in the current UML.

3 OCL-based Instance and Value Semantics

In this section we describe our approach of metamodeling semantics for different language features. We start with commonly used constraints on properties and how they can be described without leaving the technology space. Next we explain the semantics for evaluating derived properties.

3.1 Subsetting and Derived Unions

We explain our proposal by starting with a basic class diagram, which uses subsetting and union constraints on attributes of classes. Later on, we extend this diagram by using subsetting and union on associations. Subsetting and union constraints on properties (a property can be an attribute or an association end) define a relation between these two properties. The values of a subsetting property must be a subset of the values for the subsetted property. Union can be used on a single property. Its usage defines that the values of a property are the union of all its subsetting properties.

Figure 2 shows a simple model of vehicles (c. f. [4]). A vehicle consists of vehicle parts. For a car, information about the front and back wheels is added to

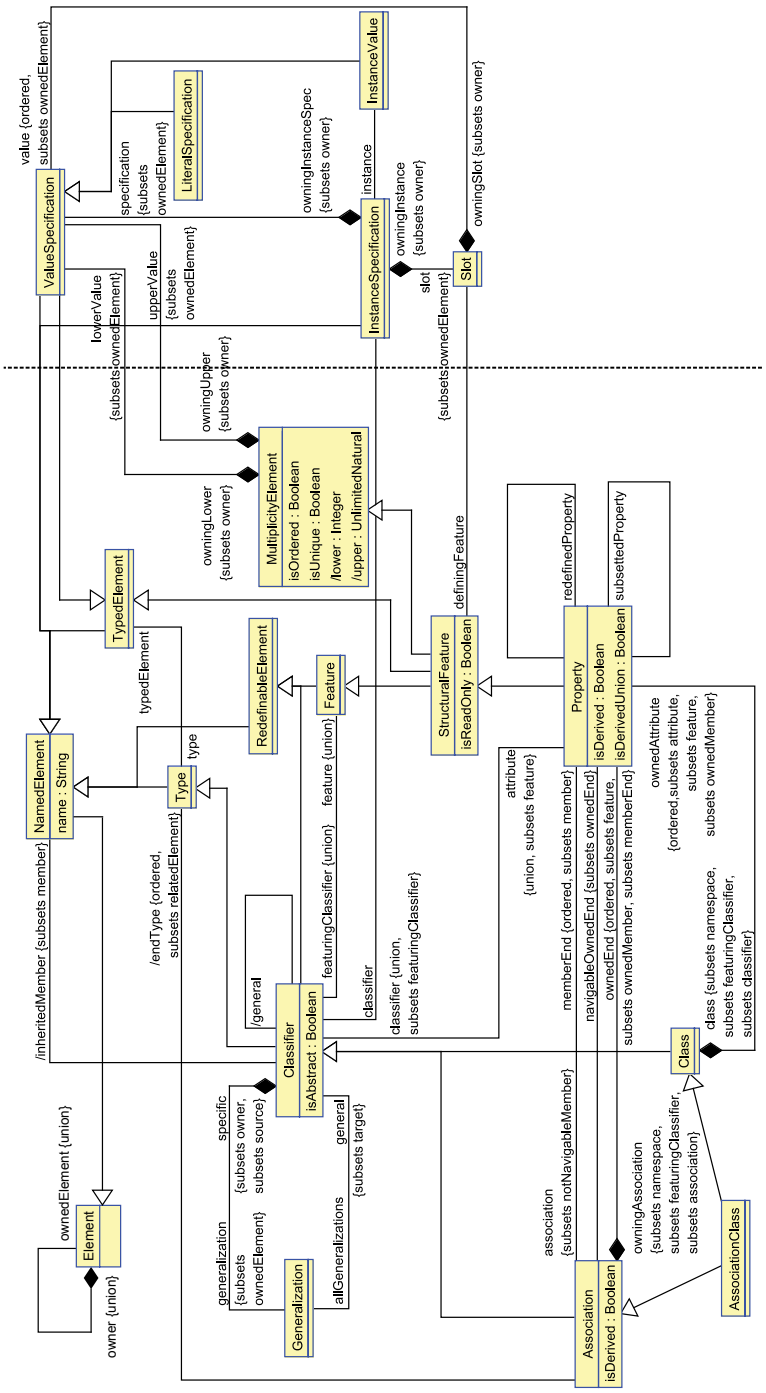


Fig. 1. Combined view of UML metamodel elements important for our work

the class Car. Because these wheels are part of the overall vehicle, the properties `front` and `back` are marked as subsets of the general property `part`. The property `part` itself is marked as a derived `union` of all of its subsets. Furthermore, the subsetting properties restrict the lower and upper bounds of the wheels to the common number of wheels for a car (2 is equivalent to 2..2). A valid object

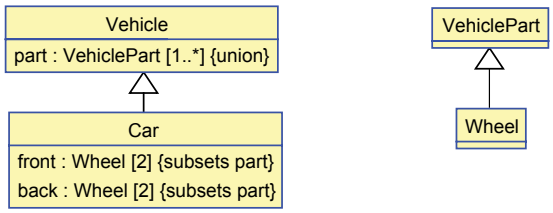


Fig. 2. Class diagram using `subsets` and `union` on attributes

diagram w. r. t. the given class diagram is shown in Fig 3. For this simple diagram, one can see directly that the intended constraints are fulfilled. However, for more complicated models, an automatic validation is required. If the used modeling language would not provide `subsets` and `union` constraints, a modeler could still specify constraints on the classes `Vehicle` and `Car`:

```
context Vehicle inv partIsUnion: let selfCar = self.oclAsType(Car) in
    selfCar <> null implies self.part = selfCar.front->union(selfCar.back)
context Car inv frontIsSubset: self.part->includesAll(self.front)
context Car inv backIsSubset:  self.part->includesAll(self.back)
```

However, these constraints would strongly couple the abstract class `Vehicle` and its subclass `Car`, because `Vehicle` needs information about its subclasses to validate the union constraint. This breaks well-known design guidelines. The above constraints are similar to the generated constraints from [20]. Using such an automatic approach would reduce the coupling.

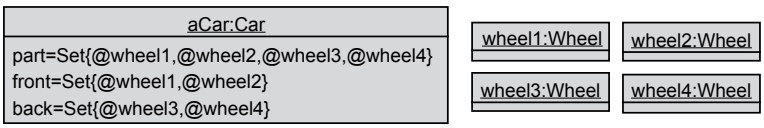


Fig. 3. A valid object diagram of the class diagram shown in Fig. 2

To allow a generic usage of these constraints the UML provides the ability to specify subset relations between properties using a reflexive association on `Property` (which represents class attributes and association ends) and to mark

a property as a derived union (see Fig. 1). Further, several well-formedness OCL rules are given, to ensure the syntactical correctness of the usage. For example, the type of the subsetting property must conform to the type of the subsetted end [23, p. 126]. However, information about the semantics of the UML language element `subsets` is only provided textually, not in a formal way. We propose to add (what we call) runtime semantics by means of OCL constraints to the already present elements describing runtime elements. For the above example, a constraint describing the runtime semantics of subsets can be specified on the UML metaclass `Slot` (a slot allows, for example, to assign an attribute value to an attribute):

```
context Slot inv subsettingIsValid:
let prop = self.definingFeature.oclAsType(Property) in
(prop <> null and prop.owner.oclIsKindOf(Class)) implies
 prop.subsettedProperty->forAll(subsettedProp |
  let subsettedValues = self.owningInstance.slot->
   any(definingFeature=subsettedProp).value.getValue()->asSet() in
  let currentValues = self.value.getValue()->asSet() in
  subsettedValues->includesAll(currentValues))
```

This constraint checks for each slot that defines a value or values for an attribute of a class, if it is a subset of the values defined by the slots of the subsetted properties. Because this constraint only considers attributes of classes, the navigation to the slots of the owning instance of the context slot is enough. For associations, and especially for associations with more than two ends, the calculation of the values to be considered is more complicated.

A class diagram which makes use of `subsets` and `union` on association ends is given in Fig. 4. The previously specified attributes `part` and `front` are changed to association ends, while the attribute `back` is left out in order to keep the following examples at a moderate size.

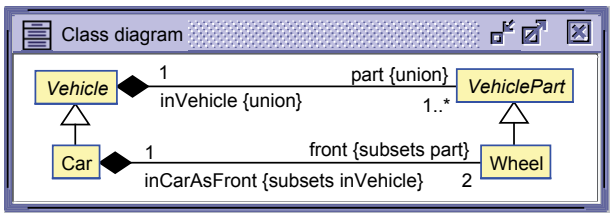


Fig. 4. Class diagram using `subsets` and `union` on association ends

Figure 5 shows an example instantiation of the class diagram. The links shown as a solid line are inserted by the user, while the dashed links are automatically calculated by our tool, because they are part of a derived union. In our tool, all derived links (either established through a derived union or through an explicit derived association end) are shown as dashed links.

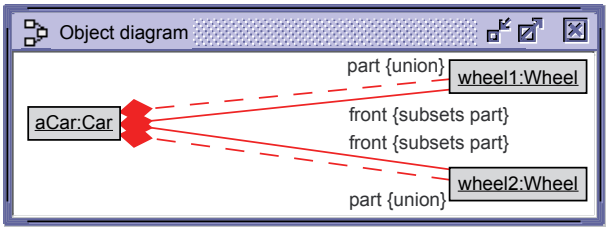


Fig. 5. A valid object diagram of the class diagram shown in Fig. 4

The object diagram in Fig. 6 shows an instantiation of the UML metamodel representing the class diagram of Fig. 4 at the top and the object diagram shown in Fig. 5 at the bottom. This figure intentionally includes so many dashed lines and compositions, in order to show the inherent complexity of the UML metamodel. This complexity can automatically be revealed by using our tool. In Sect. 4 we are going to explain these so-called virtual links in more detail. On the other side, these virtual links allow us to suppress certain elements in the object diagram to make it easier to be read. For example, the generalization relationships are only shown as derived links between the classes leaving out the generalization instance. To be more concrete, in the left upper part of Fig. 6 the dashed link between `Class3` (Vehicle) and `Class4` (Car) corresponds to the left generalization arrow in Fig. 4. We use this diagram in the following to explain an extended runtime semantics which also covers associations.

A runtime semantics for subsetting that covers attributes and association ends must consider all tuples of instances which are linked to a subsetted property and the set of instances linked to this tuple at the subsetting end. For the previously shown example on attributes, this tuple contains only one element, namely the defining instance, whereas for association ends of an association with n ends, this tuple contains $n - 1$ elements. We accomplish this by using a query operation called `getConnectedObjects()` which is similar to the operation `Extent::linkedObjects(...)` defined in the MOF specification[22], but covers n-ary associations, properties, and derived unions. We do not show the operation in detail, because it is rather lengthy[2]. The query operation uses the metaclasses of the semantic domain model to obtain all connections specified for a property. For this, it navigates to all instance specifications to consider and their owned slots. If a property is defined as a derived union, this operation is recursively invoked on all properties subsetting the derived union property and collects all connected values in a single set, i. e., it builds the union of the values. To give a more detailed view of the usage of this central operation, Fig. 7 shows the result of invoking it on the property `part` using the state shown in Fig. 6.

[2] Interested readers are referred to the USE distribution which contains a well-defined subset of the UML metamodel including this operation.

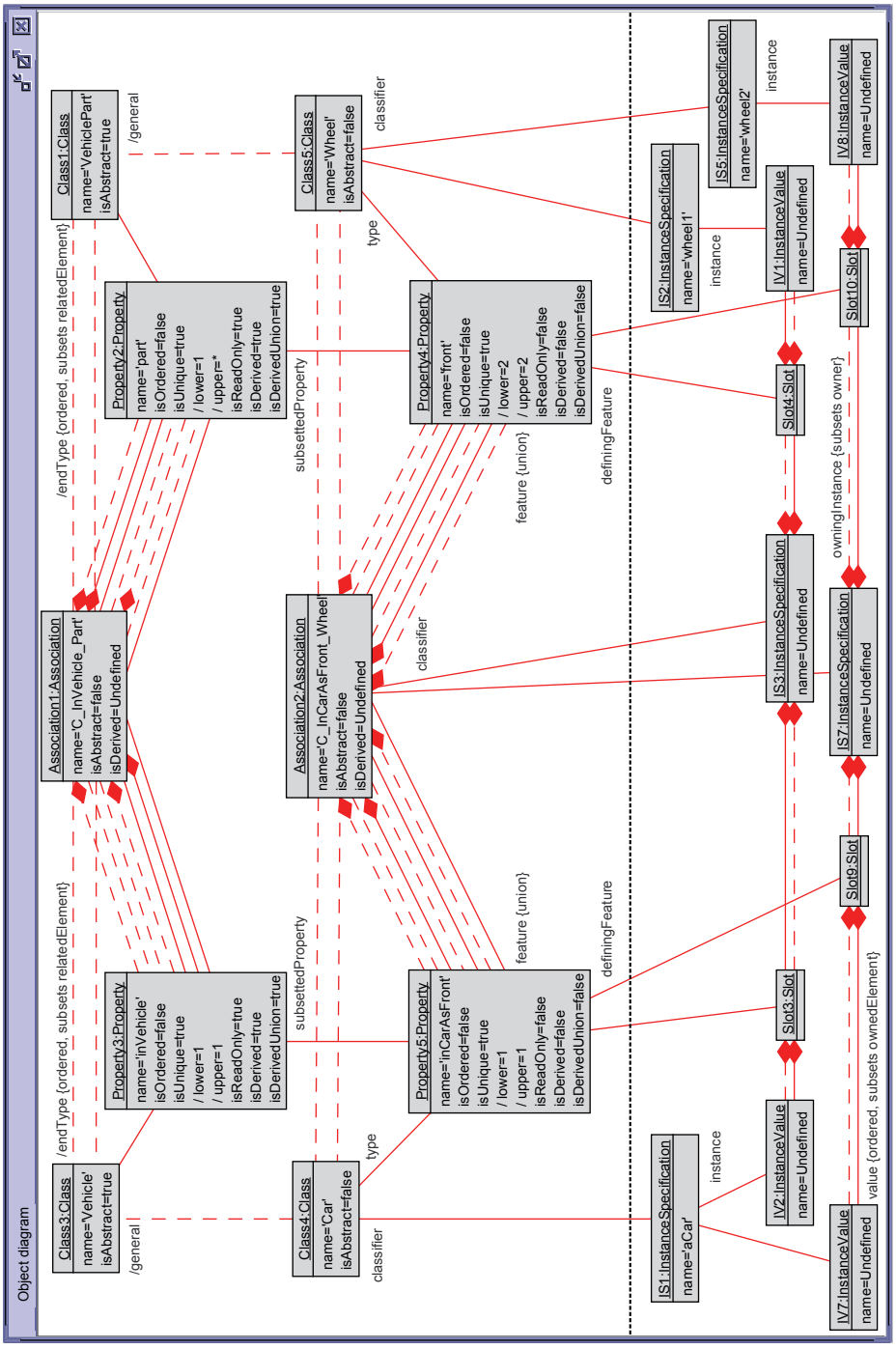


Fig. 6. The diagrams shown in Fig. 4 and 5 as an instantiation of the UML metamodel

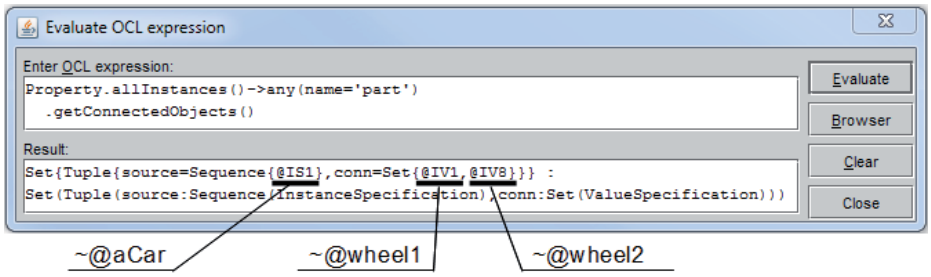


Fig. 7. Querying runtime values by using the operation `getConnectedObjects()`

The result is a set of tuples with two parts:

1. **source**: The sequence of source objects in the same order as the association ends, if the property is owned by an association.
2. **conn**: The objects connected to the source objects at the property.

The result of the evaluation is the calculated union of the property values for all possible source objects. Because only one vehicle (named `aCar`), is present in the given state, the set contains a single tuple. This tuple consists of the sequence containing the instance specification representing the object `aCar` and a set of values which are linked to this instance via subsetting properties of `part`.

Given the previously described operation `getConnectedObjects()`, we can define a constraint which ensures the subsetting semantics:

```
1 context Property inv subsettingIsValid:
2  let subsetLinks = self.getConnectedObjects() in
3  self.subsettedProperty->forAll(supersetProperty |
4    let supersetLinks = supersetProperty.getConnectedObjects() in
5    subsetLinks->forAll(t1 |
6      supersetLinks->one(t2 | t1.source=t2.source and
7        t2.conn.getValue()->asSet()->includesAll(
8          t1.conn.getValue()->asSet())))))
```

The central part of the given invariant can be seen on line 7 where the operation `includesAll` is used, which is the OCL way to validate, if a collection is a superset of another one. Some things need to be explained in a more detail. First, the usage of the operation `getValue():OclAny`, which is an extension to the UML metaclass `ValueSpecification`, is required to be able to get the concrete value of a value specification. The UML metamodel defines several operations on this class for retrieving basic types like `stringValue():String` but excludes a generic definition. Second, the collected values need to be converted to a set using `->asSet()` (see lines 7 and 8) because values can map to the same specifications. It should be mentioned, that if evaluated at runtime, the invariant only validates the union calculation if subsets is used in the context of a derived union. If subsets is used on a property which is not a derived union, the

constraint validates the user defined structure. Including the described invariant and similar invariants for other runtime elements, adds a precise definition of its semantics to the modeling language.

3.2 Derived Properties

Derived properties are widely used during the specification of models and metamodels, because they allow to shorten certain expressions and to assign associated elements an exact meaning by naming them. If a formal expression is given which describes how to calculate the values of the derived properties, the definition of the metamodel is even stronger. If the derived property is marked as read only, a query language can be used to evaluate these derive expressions. Writable derived properties are allowed for example in the UML, but we exclude this type of properties, because the computational overhead of computing the inverse values would be too high. Furthermore, only bijective derive expressions can be used. For example, an attribute `weight` for the class `Car` used in the example could be derived as follows:

```
context Car::weight:Integer derive: self.part.weight->sum()
```

Assigning a value to the attribute weight of a car cannot lead to a single result in the weights of the parts. A common way to overcome this issue is to use a declarative approach like it is done in the UML specification by using invariants for a derive expression [23, p. 128]. This transfers the responsibility to set the correct derived values or the inverse direction to an implementation. Therefore, the UML metamodel excludes the ability to add a derive expression to a property like it is done with default values. Whereas, the OCL specification links to the UML metamodel for the placement of derive expressions [24, p. 182]. We propose to add such a possibility, to allow the specification of the runtime semantics of derived read only properties. For this, we extend the metamodel by defining an additional association between `Property` and `ValueSpecification`. To ensure, that a derived expression is only used on read only properties, the following well-formedness rule needs to be added:

```
context Property inv: self.derivedValue <> null implies self.readOnly
```

The context of such a derive expression used during evaluation is related to the previously explained semantics of `subsets` and `union`. To recapitulate the essentials, for a generic solution it is necessary to consider the combinations of source objects and their connected objects. Only this allows to use derived association ends on associations with more than two association ends and further allows the evaluation of backward navigations, i. e., from a derived end to an opposite end. The major difference to the validation of subsetting is, that only if a derived association end of a binary association or an attribute are the target of a navigation, the source objects are known. If a navigation uses instead the derived end as the source, for all possible combinations of the connected end types the expression needs to be evaluated and checked if the source object of

the navigation is in the result. As an example consider the derived association end `/general` of the reflexive association defined on the class `Classifier` shown in Fig. 1. The UML specification defines the derived end using a constraint on classifier as follows [23, p. 52][3]:

```
general = self.generalization.general->asSet()
```

Used as a derive expression, the result for a navigation from a classifier instance to the association end `general` can be calculated using the source instance as the context object `self`. For the opposite direction of the navigation, i. e., navigating from a classifier instance to its subclasses, the derive expression needs to be evaluated for all instances of `Classifier`:

```
superclass = Classifier.allInstances()->select(general->includes(self))
```

For n-ary associations navigating to the derived association end, the derive expression needs to be evaluated with each combination of the source object and all possible instances at the other ends (excluding the derived end). The resulting set is the union of all evaluation results. If a navigation starts at the derived end of an n-ary association, the calculation is similar to the case of navigating backward in a binary association. Except, that the evaluation is performed for the cross product of all instances which can participate in the association. This means all instances of the end types except the derived end.

4 Tool based Validation

Because of the endogenous nature of the semantics described in the previous chapter, they were developed in parallel to extensions to a modeling tool. To validate the structural constraints used inside the UML metamodel, these were added to the tool, which allowed us to represent greater parts of the metamodel. Using a tool based validation approach and extending it in a step-wise manner added a reverse link to the specification of the runtime semantics. Without a validation tool, it is rather hopeless to bring a metamodel including well-defined semantics for a modeling language to a consistent state. Using a modeling tool to validate its modeling language, like the bootstrapping approach used for compilers, allows to discover issues beyond syntactical errors in an early state. For example, only after using derived unions in combination with derived association ends we discovered an infinite recursive definitions at the metamodel level in the current UML standard. In this particular case, a derived association end was used inside a union and the derive expression used this union. In the following parts of this section, we explain some beneficial features supporting the definition of (meta-)models which are integrated in our modeling tool USE [11, 30]. Additional supporting features are beyond the scope of this paper, but can be

[3] The constraint has slightly been modified to be more expressive. In detail, the body of the operation `Classifier::parent()` was embedded into the constraint. Further, `asSet()` was added to establish type soundness.

found in several publications of our group, e. g., [13, 14, 12]. Such a left out feature is the possibility to evaluate the specified constraints on a model instance, which was used to validate the invariants presented in this paper.

During the development of a metamodel, already on the syntactical level the usage of automatically generated dynamic views can support the user. While the size of a model increases, the usage of the modeling elements discussed in this paper (subsets, union and derived properties) can get unmanageable without adequate support by a tool. USE provides a comprehensive view which provides information about these elements defined for an association. An example of this view is presented in Fig. 8. It shows the derived union association specified between the metaclasses `Classifier` and `Feature` in the UML metamodel. A user can directly see which associations are related to the selected one and what kind of relations are defined. Implicit information, like for example a missing subsets on the opposite end is highlighted.

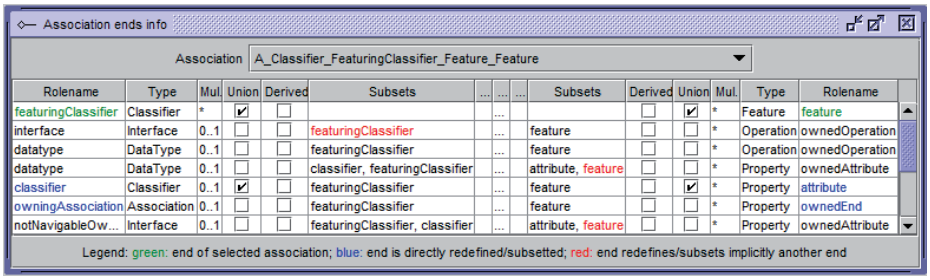


Fig. 8. Information about association relations available in USE

Another valuable functionality, which was touched slightly while explaining Fig. 5 and 6 is the automatic calculation and presentation of virtual links (presented as dashed lines) which result from associations that include a derived expression or derived unions. In Fig. 9 an in-depth view on the defined and derived links between the instances representing the composition `C_InCarAsFront` and its owned end `front` is shown. While the three lower links are specified by the user, the upper four links are automatically presented to the user because they are part of a derived union. Another usage of virtual links is to compress diagrams as it was done in Fig. 6 by excluding the generalization instances, but still showing the generalization link between classes using the derived end `/general`.

Furthermore, using derived associations allows a user to model information in a different way which may be more suitable to express her intention. The USE session presented in Fig. 10 shows an example, which uses a derived ternary association to show the direct relation of associated objects. The example defines a small library model composed of classes for users, copies and books. The fact that a user can borrow copies of books is modeled by two binary associations which together link all three classes. A third association is defined, that is derived

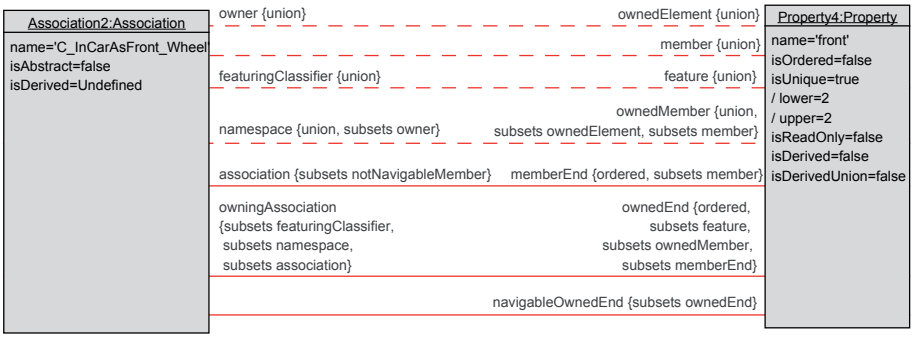


Fig. 9. A detailed view on virtual links present in the UML metamodel instance (Fig. 6)

and combines the aforementioned associations into a single ternary one. The definition of the derived association in the concrete syntax of USE is as follows:

```
association BorrowsCombined between
  User[*] role dUser
  Copy[0..1] role dCopy derived(aUser:User,aBook:Book) =
    aUser.copy->select(c | c.book=aBook)
  Book[*] role dBook
end
```

The shown textual language is an excerpt of the language used to define UML models in USE. It is comparable to HUTN (UML Human-Usable Textual Notation) of the OMG [21]. To be able to show derived links, our language defines the keyword `derive` to mark an an association end as derived. The derive keyword requires an OCL expression which defines the derived links. For n-ary associations, also the naming of the parts of a combination is required to be able to evaluate an arbitrary OCL expression. In contrast to this, a derived expression on a binary association can use a single context variable `self`, because there is no combination of instances at association ends.

For example, to calculate the links for the association `BorrowsCombined` the derive expression at the association end `dCopy` is evaluated for all pairs of `User` and `Book` objects (these pairs are expressed by the signature `(aUser:User, aBook:Book)` of the derive definition shown above. The derive expression returns all copies associated with a given pair of a user and a book. For each `Copy` object in the result set a link connected to the input pair and the copy object is shown in the object diagram. In addition, the example shows how one can use a multiplicity constraint on derived associations. In this example, the multiplicity constraint `0..1` in the association end `dCopy` excludes double borrowings (a user borrows more than one copy of the same book). The multiplicity violation of the example state is reported to the user, as can be seen at the bottom of Fig. 10.

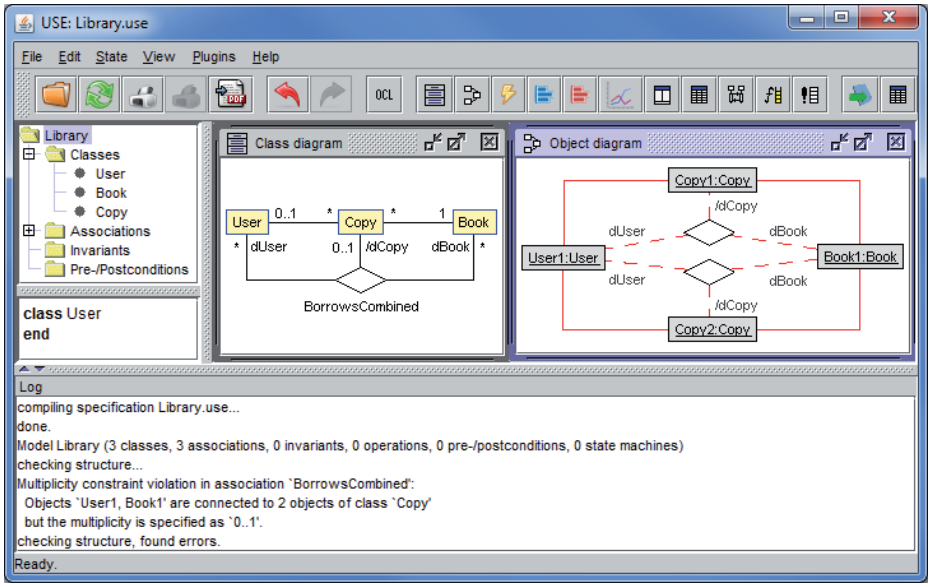


Fig. 10. Screenshot of USE while validating a snapshot with derived ternary association

5 Related Work

Metamodeling semantics has been used in areas not focused in this paper. In [8] it is applied to define the semantics of multiple inheritance using a set-theoretic based metamodel. [16] shows its application to specify the semantics of OCL, whereas [9, 15] cover a detailed view on the overall topic of metamodeling semantics. A combined view of different metamodeling levels is used in [10] to specify the semantics of entity relationship diagrams and their transformation into the domain of relational schemata.

As examples for the ongoing discussion about the need of a formal semantics for UML and to what extend it should be defined, we refer to [27] and [5]. The authors of [5] discuss the benefits and drawbacks of a precise UML specification including runtime semantics from several points of view. Furthermore, the problems arising by trying to be a general purpose language for different domains implying semantic variation points is explained. We believe, that both points of view are valid, but the viewpoints change during the development process. At an early stage of design, the used modeling language could allow to violate the precise semantics. While the process continues, these violations should be more and more forbidden until a state is reached where no violation is allowed.

Beside the vast amount of publications defining the semantics of UML, e. g. [18, 31, 28], work covering the UML language elements presented in this paper has been done. [4] gives a descriptive insight of using union and subsets and shows its relation to composite structures.

Exogenous definitions of the semantics for subset and union properties have, for example, been provided in [1] using a set-theoretic formalization, [3, 2] using graph transformations, and [19] using a so-called property oriented abstract syntax to define the semantics of what the authors call inter-association constraints (these include subsets and union). These examples of exogenous definitions of semantics all require to have expertise in the respective external semantic technology space. [20] introduces a UML profile covering redefinition and other elements. While the work is similar to ours in the sense that it stays in the same technological space, the runtime semantics is enforced generating model specific OCL constraints, like the ones shown at the beginning of Sect. 3. A semantics for subsetting using the same transformation approach is given in [7]. Another transformation approach to describe the runtime semantics of UML constraints using OCL is shown in [6]. Here, the runtime semantics implied by UML compositions are translated to OCL constrains, i. e. the semantics must be defined by a transformation into a specific application model. Whereas our semantics works in a universal way, where constraints are formulated on the metamodel level without the need for transformation.

In this paper we presented a way to validate (meta-)model instances by creating snapshots, i. e., instantiations, of these models and by examining their behavior, for example, by checking the multiplicity constraints on an instance or by examining the current states of the defined invariants. Other approaches use automatic techniques to reason about models specified in UML/OCL. An approach like [17] could, for example, be used to find valid configurations of writable derived properties as discussed earlier in this paper. In addition, it can be used like the ones in [29] and [25] to answer questions about the satisfiability and other properties of a model.

6 Conclusion and Future Work

We presented a proposal to specify the runtime semantics of a modeling language using a metamodel describing syntax and semantics in the same language. Using the same technology space reduces the overall complexity of the language description, because knowledge of other languages is not required. Furthermore, the process of specifying the language is improved, if this self describing technique is used in combination with tool-supported validation. As we have shown in Sect. 4, bringing models into being by creating snapshots can give insights into the model which are rather vague if only the static specification is used.

As future work, the application of our approach to other areas of modeling languages, for example property redefinition and association generalization, seem to be promising directions to extend our work. The covered elements of the UML metamodel for validation and the options on the user interface in our tool USE can be strengthened as well. Larger case studies with other modeling language, for example domain-specific languages, will give further feedback on the usability of the approach.

References

1. Alanen, M., Porres, I.: A metamodeling language supporting subset and union properties. Software and Systems Modeling 7(1), 103–124 (Feb 2008)
2. Amelunxen, C.: Metamodel-based Design Rule Checking and Enforcement. Ph.D. thesis, Technische Universität Darmstadt (2009), dissertation
3. Amelunxen, C., Schürr, A.: Formalizing Model Transformation Rules for UML/MOF 2. IET Software Journal 2(3), 204–222 (June 2008), special Issue: Language Engineering
4. Bock, C.: UML 2 Composition Model. Journal of Object Technology 3(10), 47–73 (Dec 2004), http://www.jot.fm/issues/issue_2004_11/column5
5. Broy, M., Cengarle, M.V.: UML formal semantics: lessons learned. Software and System Modeling 10(4), 441–446 (2011)
6. Chavez, H.M., Shen, W.: Formalization of UML Composition in OCL. In: Miao, H., Lee, R.Y., Zeng, H., Baik, J. (eds.) ACIS-ICIS. pp. 675–680. IEEE (2012)
7. Costal, D., Gómez, C., Guizzardi, G.: Formal Semantics and Ontological Analysis for Understanding Subsetting, Specialization and Redefinition of Associations in UML. In: Jeusfeld, M.A., Delcambre, L.M.L., Ling, T.W. (eds.) ER. Lecture Notes in Computer Science, vol. 6998, pp. 189–203. Springer (2011)
8. Ducournau, R., Privat, J.: Metamodeling semantics of multiple inheritance. Science of Computer Programming 76(7), 555–586 (2011)
9. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In: Evans, A., Kent, S., Selic, B. (eds.) UML. Lecture Notes in Computer Science, vol. 1939, pp. 323–337. Springer (2000)
10. Gogolla, M.: Exploring ER and RE Syntax and Semantics with Metamodel Object Diagrams. In: Nürnberg, P.J. (ed.) ACM Int. Conf. Proceeding Series (Vol. 214), Proc. Metainformatics Symposium (MIS'2005). ACM Press, New York (2005), ACM Digital Library, 12 pages
11. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69, 27–34 (2007)
12. Gogolla, M., Hamann, L., Xu, J., Zhang, J.: Exploring (Meta-)Model Snapshots by Combining Visual and Textual Techniques. In: Gadducci, F., Mariani, L. (eds.) Proc. Workshop Graph Transformation and Visual Modeling Techniques (GTVMT'2011). ECEASST, Electronic Communications, journal.ub.tu-berlin.de/eceasst/issue/view/53 (2011)
13. Hamann, L., Hofrichter, O., Gogolla, M.: OCL-Based Runtime Monitoring of Applications with Protocol State Machines. In: Vallecillo, A., Tolvanen, J.P. (eds.) Proc. 8th European Conf. Modelling Foundations and Applications (ECMFA 2012). pp. 384–399. Springer, Berlin, LNCS 7349 (2012)
14. Hamann, L., Hofrichter, O., Gogolla, M.: Towards Integrated Structure and Behavior Modeling with OCL. In: France, R., Kazmeier, J., Breu, R., Atkinson, C. (eds.) Proc. 15th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'2012). pp. 235–251. Springer, Berlin, LNCS 7590 (2012)
15. Hausmann, J.H.: Dynamic META modeling: a semantics description technique for visual modeling languages. Ph.D. thesis, University of Paderborn (2005)
16. Kleppe, A.: Object constraint language: Metamodeling semantics. In: Lano, K. (ed.) UML 2 Semantics and Applications, pp. 163–178. John Wiley & Sons, Inc. (2009)

17. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: Bishop, J., Vallecillo, A. (eds.) Proc. 49th Int. Conf. Objects, Models, Components, and Patterns (TOOLS'2011). pp. 289–305. Springer, Berlin, LNCS 6705 (2011)
18. Lano, K.: UML 2 Semantics and Applications. John Wiley & Sons, Inc. (2009)
19. Maraee, A., Balaban, M.: Inter-association Constraints in UML2: Comparative Analysis, Usage Recommendations, and Modeling Guidelines. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MoDELS. Lecture Notes in Computer Science, vol. 7590, pp. 302–318. Springer (2012)
20. Nieto, P., Costal, D., Gómez, C.: Enhancing the semantics of UML association redefinition. Data Knowl. Eng. 70(2), 182–207 (2011)
21. OMG (ed.): UML Human-Usable Textual Notation (HUTN). Object Management Group (OMG) (Aug 2004), http://www.omg.org/spec/HUTN/
22. OMG (ed.): Meta Object Facility (MOF) Core Specification 2.4.1. Object Management Group (OMG) (Aug 2011), http://www.omg.org/spec/MOF/2.4.1
23. OMG (ed.): UML Superstructure 2.4.1. Object Management Group (OMG) (Aug 2011), http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF
24. OMG (ed.): Object Constraint Language 2.3.1. Object Management Group (OMG) (Jan 2012), http://www.omg.org/spec/OCL/2.3.1/
25. Queralt, A., Teniente, E.: Verification and Validation of UML Conceptual Schemas with OCL Constraints. ACM Trans. Softw. Eng. Methodol. 21(2), 13 (2012)
26. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language - Reference Manual. Addison-Wesley, 2 edn. (2004)
27. Rumpe, B., France, R.B.: Variability in UML language and semantics. Software and System Modeling 10(4), 439–440 (2011)
28. Shan, L., Zhu, H.: Unifying the Semantics of Models and Meta-Models in the Multi-Layered UML Meta-Modelling Hierarchy. Int. J. Software and Informatics 6(2), 163–200 (2012)
29. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In: Gogolla, M., Wolff, B. (eds.) TAP. LNCS, vol. 6706, pp. 152–170. Springer (2011)
30. A UML-based Specification Environment. Internet, http://sourceforge.net/projects/useocl/
31. Varró, D., Pataricza, A.: Metamodeling Mathematics: A Precise and Visual Framework for Describing Semantics Domains of UML Models. In: Jézéquel, J.M., Hußmann, H., Cook, S. (eds.) UML. Lecture Notes in Computer Science, vol. 2460, pp. 18–33. Springer (2002)
32. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Object Technology Series, Addison-Wesley, Reading/MA (2003)

Publication A19W

OCL-Based Runtime Monitoring of JVM Hosted Applications

Authors: *Lars Hamann, Martin Gogolla, and Mirco Kuhlmann*

Proc. Workshop OCL and Textual Modelling (OCL'2011)

OCL-based Runtime Monitoring of JVM hosted Applications

Lars Hamann[1], **Martin Gogolla**[2], **Mirco Kuhlmann**[3]

[1] lhamann@informatik.uni-bremen.de
[2] gogolla@informatik.uni-bremen.de
[3] mk@informatik.uni-bremen.de
University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany

Abstract: In this paper we present an approach that enables users to monitor and verify the behavior of an application running on a virtual machine at the model level. Concrete implementations of object-oriented software usually contain a lot of technical classes. Thus, the central parts of an application, e.g., the business rules, may be hidden among peripheral functionality like user-interface classes or classes managing persistency. Our approach makes use of modern virtual machines and allows the devloper to profile an application in order to achieve an abstract monitoring and verification of central application components. We represent virtual machine bytecode in form of a so-called platform-aligned model (PAM) comprising OCL invariants and pre- and postconditions. In contrast to related work, our approach uses the original source or bytecode of the monitored application as it stands and does not require any changes. We show a prototype implementation as an extension of the UML and OCL tool USE. Also, we investigate the impact of our approach to the execution time of a monitored system.

Keywords: Runtime Validation, Monitoring, OCL, UML, Virtual Machine, Profile

1 Introduction

Model-driven development (MDD) is currently considered to be a promising paradigm for software production. MDD aims at employing models in all development phases and for different purposes. Quite common is the forward transformation of a platform-independent model (PIM) into a platform-specific model (PSM). Less common, but also studied is the backward direction transforming a PSM into a PIM. This paper studies the latter direction and concentrates on how to connect, monitor and analyse applications running on a virtual machine (e.g., the Java virtual machine (JVM) for Java or the common language runtime (CLR) for .NET languages) in terms of a design-like model formulated as a UML class diagram and enriched with OCL state invariants and OCL operation pre- and post-conditions [OMG09, OMG10].

The aim of our work is to detect general properties of a running application. When saying 'general', we think of properties that are not explicitly part of the source code but reflect characteristics which generalize and abstract certain implementation details. Our aim is to formulate central properties of a running application as OCL invariants and OCL pre- and postconditions. We call a collection of such properties a platform-aligned model (PAM) which can be seen as a link between a PSM and a PIM. A PAM will be formulated by means of assumptions which have

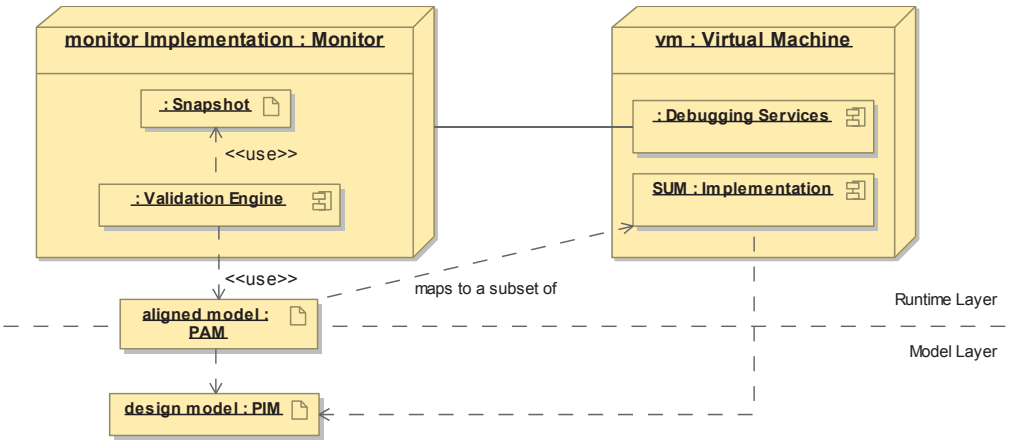


Figure 1: Deployment diagram of the monitoring approach

to be checked in prototypical scenarios invented and formulated by the developer. Designing a PAM is an iterative process in which assumptions are stated, checked and refined. Failure of an assumption may be due to an unjustified assumption which was made in the model or due to a justified assumption which does not hold in the implementation. According to the failure reason, one either has to change the model or report the failed assumption to the implementor. Thus, the development of a PAM may be seen as a (further) testing and quality assurance process for the running application.

The rest of this paper is structured as follows. In Section 2 we put forward the basic ideas of our proposal for analyzing applications running in the Java virtual machine. Section 3 explains these ideas by means of a middle-sized case study applied with a plugin for the tool USE [GBR07]. Section 4 examines the impact on the runtime performance of a system and shows details about special parts of our approach. Section 5 discusses related work. The paper ends with a conclusion and ideas for future work.

2 General approach

The main idea of our approach is to bridge the gap between platform independent models (PIM or abstract models) and the most platform specific models (PSM or implementation models). The bytecode of applications running inside a virtual machine can be seen as a PSM which is abstract enough to apply our approach, but also specific enough to make assumptions about the running system. This level of abstraction is needed because at this level one can make use of already existing features of the runtime environment of the PSM.

Modern virtual machine implementations like the JVM or the CLR of Microsoft .NET provide a rich pool of debugging and profiling interfaces. For example, the Java Platform Debugger Architecture [Ora11] allows easy access to applications running inside a (possible remote) virtual machine. We applied our approach to the Java virtual machine, but it should be possible to apply it to other virtual machines as well.

The first step of our approach is to define an platform aligned model (PAM) of the system under monitoring (SUM) which describes the expected behavior in a declarative way. This PAM could, for example, be generated out of a PIM, or reverse engineered out of an implementation. Further a PAM could be derived from a component specification to validate the possible externalized implementation of the component during the integration test phase. For this scenario our approach fits well because it does not need full access to the sourcecode of a component or system.

The PAM lies in between the runtime layer of an application and the modeling layer when using a model driven development process. Figure 1 shows the position and relations of the platform aligned model in the overall monitoring approach.

The PAM is provided as a UML model containing central classes of the SUM with attributes and associations. The class definitions contain relevant attributes, operations and OCL invariants. The dynamic behavior of a class is specified by means of OCL pre- and postconditions of the operations. The PAM should only contain central aspects of the SUM, i. e., it should abstract as far as possible from technical implementation aspects. To be able to monitor systems without modifying their source- or bytecode, the model needs to be enriched with annotations containing some information about implementation details. These implementation details are for example the concrete package a class is located in or a different name of an attribute. Further, query operations used inside the monitor need to be explicitly annotated because the monitor should not trace their execution inside the SUM.

The next step is to execute the SUM with enabled remote debugging capabilities. In the case of the JVM this can be done by providing specific arguments at startup. We do not make any assumptions about how the SUM is executed. Two possibilities are to execute it manually or by a test driver.

Once the SUM is started, the monitor with the PAM specified in the first step needs to be attached to the running system to start the monitoring process. In USE this is done by invoking a `monitor start` command with information how to connect to the remote application. The required information consists of the name of the host on which the application is running and the port on which the virtual machine is listening for a remote debugger. This port can be set as a startup parameter of the virtual machine. After the monitor has successfully connected to the SUM, it is left to the concrete implementation of the monitor, if the SUM is further executed or immediately suspended. However, the dynamic monitoring of a running SUM can only be done after it has once been suspended and an initial abstract snapshot of the system state has been taken. Such an abstract snapshot, e. g., an instantiation of a PAM, can be build up following these steps:

1. For all classes in the PAM which can be matched directly (by name or by special annotation information) to an already loaded class in the JVM[1], all existing instances in the JVM are mapped to newly created instances of the platform aligned model. In detail, this can be done by invoking the operation `instances()` on an object of the type `ReferenceType` which returns proxies to all reachable objects inside the JVM. This – for our approach important – operation was introduced in JVM version 1.6.

[1] Using the default class loader Java uses lazy initialization for classes. Therefore, not all classes might be loaded when building a snapshot.

2. For each created abstract instance in step 1 the attribute values are read. The mapping of primitive Java types to primitive OCL types should follow the common practice (c. f. [WK03]). Attributes with a type of a class defined in the PAM, i. e., reference types, can be read by using the mapping created in step 1. The possibility to define attributes referencing other instances is the reason why the creation of instances (step 1) and this step needs to be separated.

3. For all associations in the abstract model, links are created between corresponding instances. Technically this step can be merged into step 2 for performance reasons. The retrieval of links is discussed in Sec. 4.2.

After such a snapshot has been build, the monitor needs to register to several events that occur in the VM in order to allow a dynamic monitoring of the SUM. For example, the monitor needs to get informed if a not yet loaded class is initialized to be able to react on operation calls on instances of that class. However a user can already examine the SUM at this time by performing a check of the system state, e. g., by checking multiplicity constraints and invariants, by querying the system state with OCL expressions, or by visualizing the system state using examination patterns as described in [GHXZ11].

The next step in the monitoring process is to resume the suspended SUM to monitor its runtime behavior. In USE, this is done by simply invoking the command `monitor resume`. Now, a monitor can make use of the before mentioned events that it registers for. To keep the snapshot synchronized with the SUM, a monitor needs to set and listen to breakpoints inside the VM at several locations:

1. At class initialization to allow the registration of the breakpoints described next.

2. At constructors of monitored classes, i. e., classes defined in the abstract model. This allows the monitor to keep track of newly created instances and therefore enables an incremental built-up of the system state in contrast to always building a new snapshot of the running system when needed. Additional issues need to be considered for this dynamic build-up of the system state which are discussed later.

3. At the start of an operation which is specified in the abstract model. This enables the monitor to validate preconditions at runtime and in case of a failure pause the SUM.

4. Just before the exit of an operation call. This enables the monitor to validate postconditions. The break must occur after the result of the operation is calculated. The JVM provides such a mechanism. To reduce the total number of breakpoints the operation exit breakpoint can be set while entering a monitored operation and can be removed after the postconditions have been validated.

5. When a monitored attribute or link is modified. An application does not need to always use operations to modify attributes of an object. Therefore, a monitor needs the possibility to react on a modification of an object field to synchronize its snapshot. The JVM provides notifications when a field is modified to keep track of changing attributes or single values association ends. The monitoring of changes to many to many associations is more complicated and is discussed in Sec. 4.2.

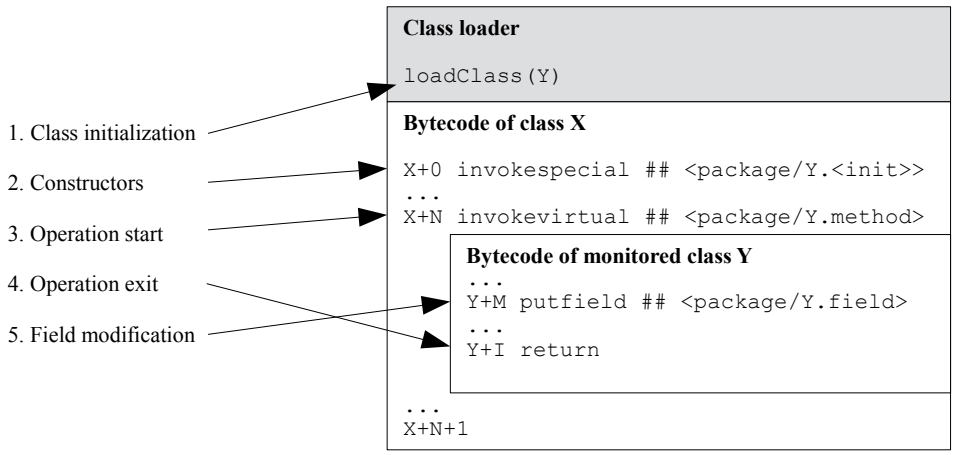


Figure 2: Monitoring events and the corresponding locations on the bytecode level

Figure 2 maps these listening locations to their adequate representation in Java bytecode, except the event when new classes are initialized. This event has no direct representation as a bytecode instruction and is also very specific to the virtual machine and the used class loader. Therefore, it is shown in an informative way.

These event locations allow a monitor to capture the relevant modifications inside a running application and trace its execution. This incremental build-up can be done until the application is exited or the monitoring process is ended. However, while applying this approach we found it useful to rebuild the snapshot when pausing the monitored application again. This enables the monitor to clean-up internal states.

Monitoring an application in the presented way allows a user to monitor the validity of UML constraints like multiplicities or compositions, invaraints, pre- and postconditions without the need to modify the source code of the application or to use special bytecode intersection mechanism which might alter the behavior of the system. A user can validate formulated assumptions about the application at runtime. This can be useful when validating a third party component where the sourcecode itself is not available, but the specification of the public interfaces can be used to create a PAM. When encountering an error during the monitoring process a user can make use of the, in contrast to the usage of a debugger, more abstract snapshot of the system. This more abstract snapshots focuses on the central parts of an application by hiding technical details. This task can be seen as *abstract debugging*. After locating the error, the user has to decide if the implementation or the PAM has to be corrected. This is equal to the task when testing and finding an error. To reduce the errors in the PAM, unit tests can be used as introduced for OCL in [CO09] and discussed in detail in [HG10].

3 Case Study

In this section we apply our monitoring approach to an existing mid-sized application using an developed plugin for the USE tool. We monitor the application to validate assumptions about

its structure and behavior. These assumptions are formulated by multiplicities, OCL invariants and OCL pre- and postconditions. Further, we show how the examination of a snapshot helps to explore unexpected behavior of a system, e. g., memory leaks.

We exemplify our approach by using an open source computer game called *Free Colonization*[2] or in short *FreeCol*. It is a modern Java-based implementation of the 1994 published game *Sid Meier's Colonization*[3]. The game itself is a round-based strategy game with the goal to colonize America and finally to achieve independence. The game takes place on a matrix-like map which consists of tiles with different types, e. g., water, mountain, forest. Different units operate on this map and can explore unknown territory, build colonies, trade goods, etc. Fig. 3 shows an example state of a running game. One unit (i. e. a pioneer) is placed in the center of the shown map part surrounded by several different tile types.

To formulate assumptions about the application we start by taking a look at some central game rules. While there are many other rules, we only use some rules related to the founding of a colony to keep the example moderate. The following rules are derived by examining the documentation and by own observations while executing the game. A unit can build a colony if

1. its current position is on a tile which does not contain another colony,
2. the unit has enough moves left to build a colony, or
3. there are no other colonies placed directly to the current tile.

Because we are monitoring an existing application which does not provide a design model we need to build one from scratch. Another approach would be to reverse engineer the source-code and then simplify the extracted model to the required elements. As we will see, building a model from scratch does fit well to our purpose. When analyzing the rules using the common approach to find candidate classes by nouns, we find four class candidates in the rules: `Position`, `Tile`, `Colony`, `Unit`. However there are some other needed classes, e. g., `Map` which is not mentioned in the rules but the class is needed as a container. Other candidates are no classes but roles of them, e. g., position as role of tile.

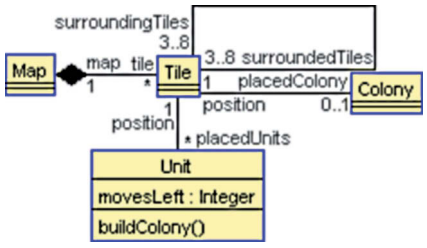
A possible platform independent model which can be created out of the information given by the above rules is shown in Fig. 4(a). In this model a unit is positioned on a tile which is part of exactly one map. A tile has three to eight surrounding tiles and can be the position of at most one colony. The available moves of a unit a stored inside of the attribute `movesLeft`. Our assumptions about when a unit is allowed to build a colony are shown as OCL preconditions in Fig. 4(b).

As described before, the PIM has to be aligned to the platform the application is running on. Therefore information about the concrete implementation is needed. When applying our approach as part of a model driven process these information is encoded inside the transformation rules used to generate the PSM and can be reused to generate the PAM. While we are examine an application which is not developed in a model driven way, we need to align it manually by examining the implementation.

[2] Project website: http://www.freecol.org

[3] The corresponding Wikipedia article gives detailed information about the game play. http://en.wikipedia.org/wiki/Sid_Meier%27s_Colonization

Figure 3: Sample game situation in FreeCol



(a) Class diagram

```
context Unit::buildColony()
  pre tileIsEmpty:
    self.position.placedColony.isUndefined()

  pre noSurroundingColonies:
    self.position.surroundingTiles->forAll(t |
      t.placedColony.isUndefined())

  pre hasMovesLeft:
    self.movesLeft > 0
```

(b) Preconditions

Figure 4: Platform independent model derived from above game rules

The source code of version 0.9.2 of FreeCol contains an overall of 551 classes, but as we will show relevant to our goal to validate the implementation of the above rules are only few of them. The central "business logic" of FreeCol is located in a package called `net.sf.freecol.common.model`. This package still contains 92 classes. The concrete implementation differs from our first model because of various reasons. First, it takes into account a lot of other features which are not relevant to our assumptions. Further, the developers took other design decisions when implementing the game. For example the implementation of the map stores the tiles inside of a multi-dimensional array whereas we modeled it as some kind of linked list, i. e., the map is constructed by linking a tile to its surrounding tiles. From the modeling perspective, that makes sense, but taking performance considerations into account the array implementation fits better.

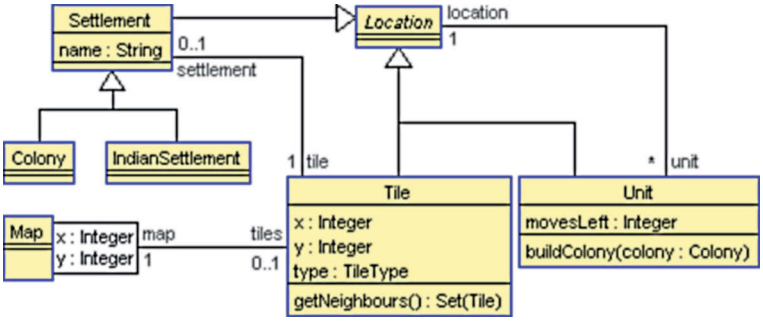
A model which is aligned to the concrete implementation is given in Fig. 5(a). One can see that the reflexive association of tile is no longer needed because the neighbored tiles can be calculated by the x and y coordinates. The implementation as a multidimensional array is represented as a qualified association which also guides the snapshot generation process to read an array at runtime. Another interesting change is the introduction of the class `Location`. While examining the rules we stated that position is a role instead of a class. It turns out that due other features a class `Location` is needed because there are several entities that can serve as a location. A unit itself can be the location of other units, e. g., a ship. Another important change is the introduced parameter `colony` of the operation `Unit::buildColony()`. The developers decided that not the class `Unit` should take care of creating a new instance of the class `Colony`. Instead, an already created instance is passed as an argument.

Because the structure of the model changed, the OCL constrains defined for the PIM need to be changed, too. The adjusted constraints are shown in Fig. 5(b). One might wonder why the invariant `Colony::noNeighbours` is contained in the model. Looking at the preconditions of the operation `buildColony()` it seems to be redundant. The reason for explicitly considering the invariant is that while monitoring, our approach allows a user to attach to a system at any time. Therefore we cannot make any assumptions about the validity of the preconditions in previous calls to operations.

The operation `Tile::getNeighbours()` is introduced to simplify the definitions of the constraints. To notify USE to ignore this operation while monitoring it is annotated as a query operation. This is done by the USE annotation mechanism that is provided to allow plugins to read additional information out of a USE model without the need to change the model parser. USE annotations look very like Java annotations. After an `@` symbol the name of the annotation is given following a possible empty list of attribute values pairs enclosed in brackets:

```
@Monitor(isQuery="true")
getNeighbours() : Set(Tile) = let neighbours = Set{} in ...
```

On the semantic level, these annotations are conceptually equal to UML stereotypes. The only difference in USE is that they are not statically typed, e. g., no profile has to be defined and referenced. The model can now be used to monitor the execution of the application. In contrast to simplify an automatically reversed engineered model with all 551 classes their attributes and operations which would have been reverse engineered, the demonstrated forward modeling approach resulting in seven classes seems to be more efficient when validating central aspects of a system.



(a) Class diagram

```
context Unit::buildColony(colony:Colony)
  pre movesLeft: self.movesLeft > 0

  pre tileIsEmptyAndFits:
    self.location.oclIsKindOf(Tile) and
    self.location.oclAsType(Tile).
      settlement.isUndefined()

  pre noSurroundingColonies:
    self.location.oclIsKindOf(Tile) and
    self.location.oclAsType(Tile).
      getNeighbours()->forAll(t |
        t.settlement.isUndefined())
```

```
context Colony inv noNeighbours:
  self.tile.getNeighbours()->forAll(t |
    t.settlement.isUndefined())
```

(b) Constraints

Figure 5: Platform aligned model

To begin the monitoring process the application needs to be started with additional parameters which setup the interfaces of the virtual machine to listen for remote connections. The parameters are well documented in the JVM documentation and are not described here, except one interesting parameter. The parameter `suspend` allows to specify the execution behavior of the virtual machine. When using the value `yes` the JVM immediately pauses execution until a remote application instruments it to resume. This option is useful to monitor an application including the whole initialization process.

After FreeCol is started with a JVM listening for a connection, the monitoring process can be started by USE. Before it can attach itself to the JVM the PAM has to be loaded. After this, the monitoring can be started by the command `monitor start`. After a successful connect, USE registers for important events and keeps track of changes inside the virtual machine. However when an application was started without the suspend option, USE at first needs a snapshot of the running application. This can be achieved by invoking the command `monitor pause`. USE suspends the monitored application and reads all instances of the classes specified in the PAM, sets their attributes and creates links as described in Sec. 2. Figure 6 shows parts of the snapshot taken at the state of FreeCol as shown in Fig. 3. We only show a part of it because already with the smallest map and at the very beginning of a game the snapshot read into USE consists of about 6,000 objects most of them (5,750) of type Tile and 4,000 links.

Please note that the alignment of the tile objects is following their x and y values and not their positions in the screenshot of the game. FreeCol uses a rather complicated approach following the layout on the screen to save the game maps. For example, when moving to north a unit decreases its x position by two instead of one.

While the colony *Isabella* and the Indian settlement can easily be found, the units are harder to identify because they are not named. `Unit85` is the Indian unit placed south of the Indian settlement. `Unit10` is the unit placed south-east of the Indian settlement. `Unit12` is the pioneer located in the center of the screen, whereas `Unit46` is not visible because it resides inside of the colony *Isabella* which is denoted inside of the screenshot by the number displayed in the center of the colony.

The difference between the number of tiles (5,750) and the overall number of links (4,000) already indicates that our assumption about the multiplicity specification at the association end `map` reachable from `Tile` is wrong. When examining the snapshot it turns out, that 1,830 tiles are not linked to a map but are referenced inside the virtual machine by some other objects. A possible cause of such a situation could be an implementation which leads to memory leaks. Although Java uses a garbage collector (GC) to reduce the possibilities of memory leaks, they still can happen. For example, when using static container classes the containing objects will never be collected by the GC because they are always reachable by the static container. In fact, the detection of memory leaks was one of the reasons why the used operation `instances()` was added to the JDA[4].

In our example, we used the following approach to examine the cause of the missing links to a map. In a step wise manner, we added classes to the PAM which use an attribute of the type `Tile`. For each step we connected to the a running game and took a snapshot of the running system and evaluated OCL queries on it. We quickly found classes which use delegates

[4] See http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5024119

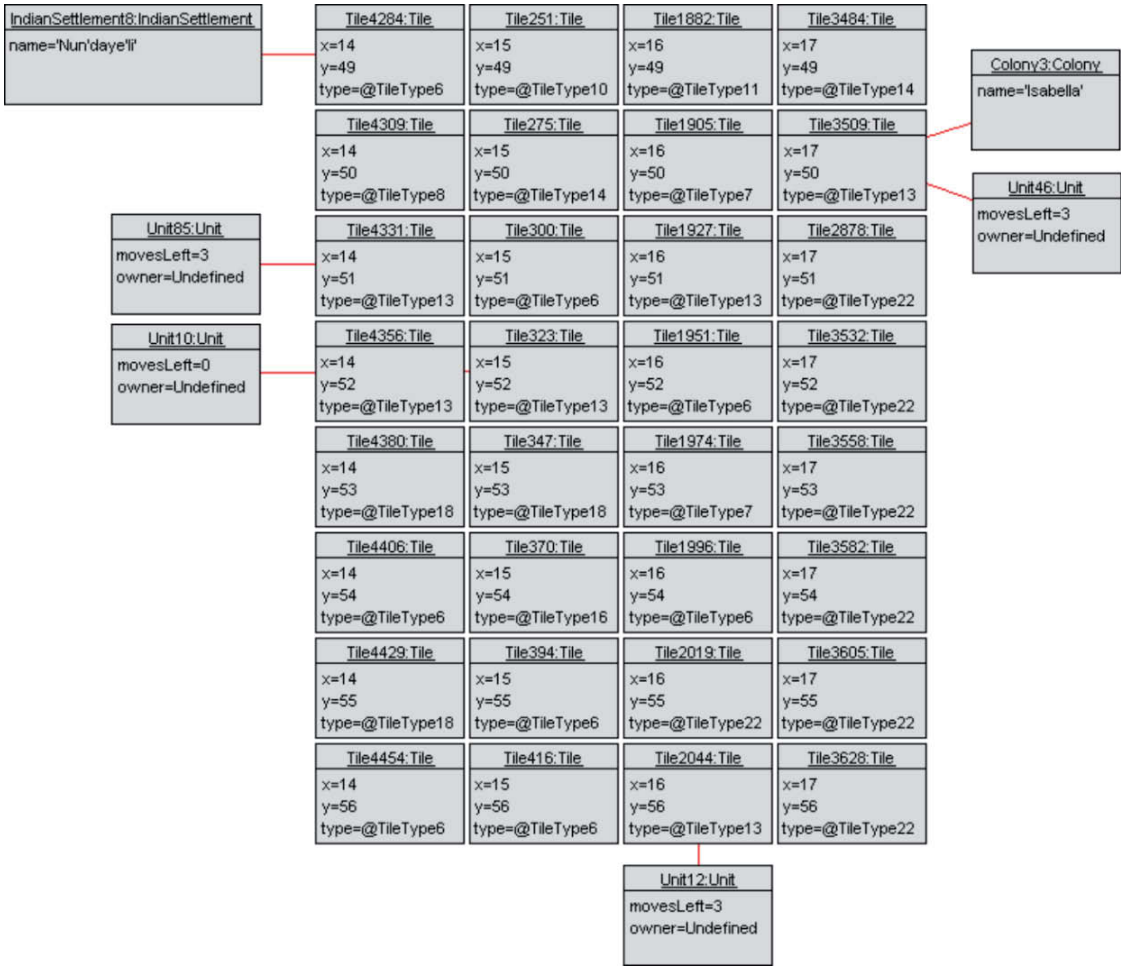


Figure 6: Parts of the snapshot taken at runtime

of tiles not connected to a map. These classes are mostly used inside of the graphical part of the application and are not used by the data model containing the important rules for our assumptions. Therefore, we could exclude a memory leak at this part and needed to align our assumption about the multiplicities. Interestingly, there seems to be still a memory leak related to the class Tile. After loading a saved game the number of tile instances is growing. We have not examined this issue any further, but it indicates that when loading a game the old game state is not disposed correctly.

While we have shown that examining a snapshot of a suspended application can be useful to detect possible structural issues, it can be used to examine some dynamic aspects of the application as well. One can check, for example, if an operation can currently be called on any instance of the defining class. Taking our snapshot into account one can check if any unit can currently build a colony. This can be achieved by using the preconditions as query conditions. However this is only possible in a simple way for preconditions that do not use parameter values. A skeleton for the combined query representing the precondition of the operation `Unit::buildColony` for all units owned by the player 'lhamann' is shown below. Instead of repeating the bodies of the preconditions shown in Fig. 5(b) they are represented by the placeholder `<preBody>`. The variables used inside of the let expressions denote the corresponding body.

```
let myUnits = Unit.allInstances()->select(owner.name='lhamann') in
  myUnits->select(self |
    let preMovesLeft = <preBody> in
    let preTileIsEmptyAndFits = <preBody> in
    let preNoSurroundingColonies = <preBody> in
      preMovesLeft and preTileIsEmptyAndFits and preNoSurroundingColonies)
```

This query results in a set of units which should be able to build a colony w.r.t. our assumptions. To validate our assumptions we resume the game and let the unit placed in the center of the sample state build a colony. Using our assumptions, this is indeed successful. The overall command list can be examined in USE and is shown in Fig. 7. Note that the object identifier are different to the identifier of the snapshot shown in Fig. 6, although the operation was called exactly at the same state. This is because we used a different run of the application to record the operation call using a saved game to start at the same state. This exemplifies, that when taking a snapshot one can not rely on the order in which instances are read, because the virtual machine could, for example, have reordered the objects on the heap.

The shown command list leads to another interesting observation. Some commands are executed more then once, e. g., setting the attribute `movesLeft` to 0. One can now examine the implementation to work out why this command is executed that often or she can refine the model to include more operation calls that should be monitored. When using the latter approach we quickly find out that several operations are setting the attribute value to zero. This behavior is indeed needed, because the operation can be called independent from each other.

Because the monitored product FreeCol is in a stable state of development and the observed operation is a central part of it, it is hard to identify a real bug to show a failing precondition. To simulate it, we interspersed a simple error (changing `movesLeft>0` to `movesLeft=0`) into our assumed precondition. Given this circumstances the last visible command in the command

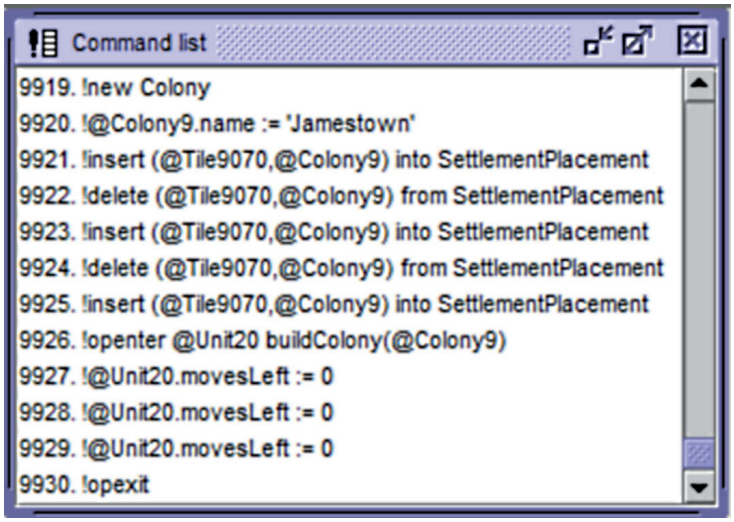


Figure 7: Monitored commands of buildColony()

list shown in Fig. 7 is 9925. A user now can examine the current system state and try to identify the error. As mentioned before the user has to take the specification of the PAM and the implementation into account and needs to judge what caused the error: a flawed implementation or incorrect assumptions as it is the case with our incorrectly defined precondition.

It could also be the case, that the design of an application uses a defensive programming style, i. e., the called operation validates it parameters and informs the caller of the failed preconditions by raising an exception. Therefore, in our approach the normal execution can be continued by resuming the application. Using such a defensive programming style will move the assumptions specified in a PAM into the postconditions, e. g., forcing the return value of an operation to the undefined value when an argument violates assumptions.

As with the preconditions, the handling of postconditions is nearly the same, except the access to the system state before the operation was called using the `@pre` operator. When using and OCL validation engine which supports the `@pre` operator and manages an own instance of the system state, this feature can be used without much effort. This is one reason, why the validation of constraints is done with an own snapshot instead of querying the Java heap.

When running the monitoring process with a more detailed PAM, the overall call stack can be taken into account when resolving failed assumptions. Call stacks can be visualized using a UML sequence diagram as shown in Fig. 8. Again, the object identifier changed because we needed to reattach to the SUM with a more detailed model. This visualization of call sequences is in our opinion also useful for documentation purposes. It allows an easy way to show central operation calls of real executions of a system, in contrast to exemplified call sequences constructed by hand or reversed engineered sequence diagrams showing an abstract execution path.

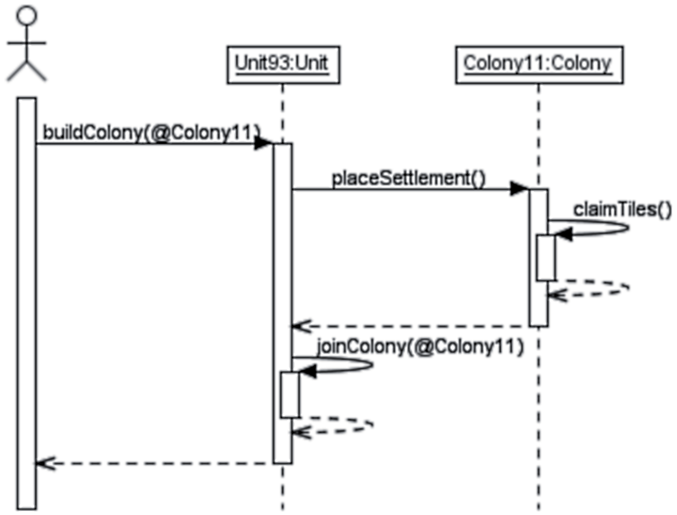


Figure 8: Monitored sequence diagram of an execution of buildColony()

Table 1: Performance of snapshot creation

Task	SOIL	Native	
Instance creation	≈8,700	≈9,700	instances/s
Attribute assignment	≈8,700	≈17,400	attributes/s
Link creation	≈4,100	≈4,100	links/s

4 Discussion

In this section we discuss technical aspects of out approach in detail. First give some brief information about general performance and the runtime overhead introduced by using our monitor implementation. After this, we discuss the link retrieval task in detail to show various ways with their advantages and disadvantages how to achieve this.

4.1 Performance and Runtime Overhead

Our implementation can use two different kinds of snapshot generation. It can be built by either using native USE system operations or by evaluating SOIL[5] statements [Büt11]. Using SOIL statements, the whole build-up process of the snapshot is encapsulated in command objects. These commands can be used to save an initial snapshot to a script file for later use. Table 1 shows the average values for the three main task when creating a snapshot, i. e., instance creation, attribute assignment and link insertion.

The values were measured on a Intel Core 2 Duo notebook running at 2.5 GHz while taking the whole snapshot which is partly shown in Fig. 6. The snapshots were taken several times to exclude the overhead of the just in time compiler. It can be seen that the impact of SOIL

[5] SOIL is an acronym for simple OCL-based imperative language.

Table 2: Performance of dynamic monitoring

Monitored events	Duration	#Events monitored	#Events/s
None (no monitor attached)	6 ms	0	n/a
None (monitor attached)	6 ms	0	n/a
Instance creation	≈7,600 ms	10,001	≈760
+ Attribute assignment	≈8,500 ms	30,002	≈3,530
+ Link creation	≈9,400 ms	40,002	≈4,225
+ Operation call	≈18,000 ms	60,002	≈3,333

comes to play only while assigning attribute values. This is due to the fact that an assignment of an attribute needs fewer validation tasks when executed than a link creation and therefore the encapsulation of the commands has a greater influence.

To examine the overhead of the dynamic monitoring we used a small application which executes several steps that can be monitored in a loop. We used an own small application because it allows a more precise measurement of the overhead in contrast to our case study which monitored an operation that is called rarely. For the case study we can only state that there is a marginal impact to the runtime behavior which leads to small delays that are barely noticeable, for example, when moving units, which changes parts of the snapshot, e. g., the unit position.

The application creates a new instance and calls an operation on it inside each iteration. The operation sets a primitive attribute of type integer and an object valued attribute. The loop was iterated 10,000 times. The time needed to execute the whole iteration with different granularity of monitored events is shown in Tab. 2. The overhead of one or two events respectively results from the fact that a single instance is created before the loop which is used to set the object valued attribute. When monitoring attribute assignments for each iteration step, two events are monitored: the initialization inside the constructor and the assignment inside the operation.

At a first look, this overhead seems to be out of scale, but as described before our approach is meant to be applied only to central parts of a system. Unrelated parts of the system are not tangled by the monitoring, e. g., graphical operations which are called very often, and therefore perform as without an attached monitor.

4.2 Link retrieval

While retrieving links of one-to-many associations can easily be done by reading the value of the field at the association end with multiplicity one, reading many-to-many associations is more complicated. This is similar to the issue how to generate association implementations when applying model transformations in an MDA process (c. f. [AHM07]).

We identified two potential ways to read links of a many-to-many association into a snapshot of an platform aligned model, either by examine the fields of the container object which saves the corresponding objects or by using iterators.

The main drawback of reading the details of container classes is that it requires a deep knowledge about the internal structure of them. Further, when new versions of the collection library, e. g., a new Java runtime version, is released the monitoring framework has to be adopted.

The usual way to abstract from these detailed information is to use some kind of iterator pattern [GHJV95]. However using a iterators requires to execute parts of the application out of the normal program flow. While this could be done with current virtual machines this could lead to forged results when monitoring an application. An implementation can for example write something while iterating over a container, but our monitoring approach should not alter the system state of a running application. The main benefit of this approach is that a monitored application does not need to keep all linked objects in memory at once, e. g., they can be stored in a database and retrieved when needed. We decided to retrieve links only by examining the fields of container classes to keep the execution flow of the monitored application untouched. However, we plan to support both approaches in the future.

Nearly the same considerations are valid in the context of the dynamic built-up of many-to-many links during program execution. A monitor can listen for a modification of the underlying data structure or it can set breakpoints at operations which modify the content of a container, e. g., `List.add(Object o)`. Which technique to use depends on the concrete implementation of the monitored system. For example, if the monitor uses operation breakpoints no detailed knowledge about the underlying container is needed, but it cannot be sure that an element is really added. This would be the case when using modification events, but as stated above a mapping to the concrete implementation is needed.

5 Related Work

Today, several approaches to applying runtime monitoring for verification and validation purposes exist. General comparisons regarding different methods for checking constraints at runtime have been carried out in [FGOG07] and [ASCY10]. The authors in [FGOG07] call approaches using AspectJ and other reactive techniques like proxy implementations 'Interceptor Mechanisms'. These mechanisms are related to our approach. However, all presented interceptor mechanisms alter the implementation of the monitored system, either by changing the sourcecode, by injecting bytecode, or by enforcing a particular architecture like the application of proxy classes.

In [ASCY10], the authors identify four distinctive approaches using OCL constraints to performing runtime checks:

(1) using implementation languages such as Java,
(2) using built-in assertion facilities such as the assert statement,
(3) using assertion or design-by-contract languages such as JML,
(4) using aspect-oriented programming language such as AspectJ

The first two categories are based on built-in structures of the target platform like `if`- or assertion statements. In contrast to our approach, the integration of approaches belonging to these categories into a system requires a full access to the sourcecode.

The Java Modeling Language (JML) can be applied for formal verification and runtime assertion checking [LCC+05]. Approaches for translating OCL expressions and constraints into JML are, for example, presented in [Ham04] and [AFC08]. In [CLSE05], program code is separated from code intended for specification purposes by introducing model methods and model fields

which abstract from concrete program variables and query methods. The respective features were implemented in the runtime assertion checker for JML. A JML compiler built on the Eclipse Java compiler is presented in [SC10] which, in contrast to the original JML compiler, supports Java 5 features, and is significantly faster, since it makes use of an AST merging technique.

The tool 'ocl2j' enforces OCL constraints in Java through translating OCL expressions into Java code [DBL06]. The generated assertion code is integrated at the bytecode level using AspectJ. Analog approaches are presented in [BDL05] (focusing on templates for automatically integrating invariants and pre- and posconditions at the bytecode level) and [GR08, GM09, RG03]. In [CA10], the AspectJ approach is applied to program testing by using OCL constraints for filtering test data and determining test results.

The Dresden OCL toolkit provides for two distinctive approaches to runtime verification based on OCL constraints [DW09]. Within the so-called interpretative approach the Dresden OCL2 Interpreter is integrated into a runtime environment interpreting the OCL constraints for all instances of the underlying model during execution. The 'generative' approach is currently based on the generation of AspectJ code which can ensure constraints at software runtime.

In [BSG10] the monitoring of state machines is focused. OCL is not used. The authors, though, sketch three general possibilities to extract runtime models. Beside the already mentioned 'aspect oriented approach', a so called 'listener approach' and a 'debugging approach' is described. The debugging approach is closely related to our method of using the debugging facilities. However, the tool presented in [BSG10] relies on the listener approach which can be seen as an architecture enforcing approach.

So called 'synchronizers' are used in [SHCS10] to synchronize a running system with a runtime model, i. e., to immediately change the system when the model has been updated, and to immediately adapt the model if the system progresses. Synchronizers can be generated for specific platforms. They make use of the APIs provided by the target systems. As discussed in Sec. 4.2, the use of APIs may lead to side-effects while querying the system state. In [SHCS10], the runtime model is represented in form of an EMF model. Thus, various MDE tools can be applied.

6 Conclusion

We presented an approach for monitoring assumed properties in form of OCL constraints for a running Java application. The approach was made possible by taking advantage of the powerful features of the Java virtual machine. Assumptions are formulated as state invariants or operation contracts and are understood as a platform-aligned model (PAM). We reported on a prototypical implementation of a monitor integrated into the UML-based Specification Environment (USE). The connection between the PAM and the platform-specific model (JVM byte code) was established through particular annotations in the PAM. Our approach does not need to modify the PSM as in approaches based on aspect-orientation. We explained our work by a non-trivial example of an open-source game.

As future work we want to (semi-)automatically detect the constraints in the platform-aligned model. For example, it could be possible to extract invariants or pre- and post-conditions (or at least parts thereof) from boolean expressions in the source code. The extraction of classes,

attributes and role ends of associations could be based on run-time metrics. We have to work further on the detection of associations and links in the case of many-to-many relationships. Comprehensive case studies will help to improve our work. An in-depth comparison to related approaches, for example, based on aspect-orientation or approaches considering the JML as a target language is needed. The prototype has to be improved in various directions. Moreover, a direct integration of OCL-like features into a virtual machine (e.g., by means of the plugin-like agent mechanism in the JVM) seems a promising line of research as well.

Bibliography

[AFC08] C. Avila, G. Flores, Y. Cheon. A Library-Based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking. In Arabnia and Reza (eds.), *roceedings of the 2008 International Conference on Software Engineering Research & Practice, SERP 2008*. Pp. 403–408. CSREA Press, 2008.

[AHM07] D. Akehurst, G. Howells, K. McDonald-Maier. Implementing associations: UML 2.0 to Java 5. *Software and Systems Modeling* 6(1):3–35, mar 2007.

[ASCY10] C. Avila, A. Sarcar, Y. Cheon, C. Yeep. Runtime Constraint Checking Approaches for OCL, A Critical Comparison. In *roceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010)*. Pp. 393–398. Knowledge Systems Institute Graduate School, 2010.

[BDL05] L. C. Briand, W. J. Dzidek, Y. Labiche. Instrumenting Contracts with Aspect-Oriented Programming to Increase Observability and Support Debugging. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*. Pp. 687–690. IEEE Computer Society, Washington, DC, USA, 2005.

[BSG10] M. Balz, M. Striewe, M. Goedicke. Monitoring Model Specifications in Program Code Patterns. In *Proceedings of the 5th International Workshop Models@run.time*. Pp. 60–71. 2010.

[Büt11] F. Büttner. *Reusing OCL in the Definition of Imperative Languages*. PhD thesis, University of Bremen, 2011.

[CA10] Y. Cheon, C. Avila. Automating Java Program Testing Using OCL and AspectJ. In *Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations*. ITNG '10, pp. 1020–1025. IEEE Computer Society, Washington, DC, USA, 2010.

[CO09] J. Chimiak-Opoka. OCLLib, OCLUnit, OCLDoc: Pragmatic Extensions for the Object Constraint Language. In Schürr and Selic (eds.), *Model Driven Engineering Languages and Systems*. Lecture Notes in Computer Science 5795, pp. 665–669. Springer Berlin / Heidelberg, 2009.

[CLSE05] Y. Cheon, G. Leavens, M. Sitaraman, S. Edwards. Model variables: Cleanly Supporting Abstraction in Design By Contract. *Softw. Pract. Exper.* 35:583–599, May 2005.

[DBL06] W. J. Dzidek, L. C. Briand, Y. Labiche. Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005*. LNCS 3844, pp. 10–19. Springer, Berlin, 2006.

[DW09] B. Demuth, C. Wilke. Model and object verification by using Dresden OCL. In *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice*. Pp. 687–690. Ufa, Russia, 2009.

[FGOG07] L. Froihofer, G. Glos, J. Osrael, K. M. Goeschka. Overview and Evaluation of Constraint Validation Approaches in Java. In *Proceedings of the 29th international conference on Software Engineering*. ICSE '07, pp. 313–322. IEEE Computer Society, Washington, DC, USA, 2007.

[GBR07] M. Gogolla, F. Büttner, M. Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming* 69:27–34, 2007.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GHXZ11] M. Gogolla, L. Hamann, J. Xu, J. Zhang. Exploring (Meta-)Model Snapshots by Combining Visual and Textual Techniques. In *Proc. 10th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'2011)*. 2011.

[GM09] S. R. GY. Cheon, C. Avila, C. Munoz. Checking design constraints at run-time using OCL and AspectJ. *International Journal of Software Engineering* 2(3):5–28, 2009.

[GR08] M. Gopinathan, S. K. Rajamani. Runtime Monitoring of Object Invariants with Guarantee. In *Runtime Verification, 8th International Workshop, RV 2008*. LNCS 5289, pp. 158–172. Springer, Berlin, 2008.

[Ham04] A. Hamie. Translating the Object Constraint Language into the Java Modelling Language. In *Proceedings of the 2004 ACM symposium on Applied computing*. SAC '04, pp. 1531–1535. ACM, New York, NY, USA, 2004.

[HG10] L. Hamann, M. Gogolla. Improving Model Quality by Validating Constraints with Model Unit Tests. In *Proc. 7th Int. Workshop on Model-Driven Engineering, Verification, and Validation (MODEVVA'2010)*. 2010.

[LCC+05] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.* 55(1-3):185–208, 2005.

[OMG09] *UML Superstructure 2.2*. Object Management Group (OMG), Feb. 2009. http://www.omg.org/spec/UML/2.2/Superstructure/PDF/

[OMG10] *Object Constraint Language 2.2*. Object Management Group (OMG), Feb. 2010. http://www.omg.org/spec/OCL/2.2/

[Ora11] Oracle. Java™Platform Debugger Architecture - Structure Overview. 2011. http://download.oracle.com/javase/6/docs/technotes/guides/jpda/architecture.html

[RG03] M. Richters, M. Gogolla. Aspect-Oriented Monitoring of UML and OCL Constraints. In Aldawud et al. (eds.), *Proc. UML'2003 Workshop Aspect-Oriented Software Development with UML*. Illinois Institute of Technology, Department of Computer Science, http://www.cs.iit.edu/~oaldawud/AOM/index.htm, 2003.

[SC10] A. Sarcar, Y. Cheon. A new Eclipse-based JML compiler built using AST merging. Technical report 10-08, Department of Computer Science, The University of Texas at El Paso, Mar. 2010.

[SHCS10] H. Song, G. Huang, F. Chauvel, Y. Sun. Applying MDE Tools at Runtime: Experiments upon Runtime Models. In *Models@run.time*. Pp. 25–36. 2010.

[WK03] J. Warmer, A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 2003. 2nd Edition.

Publication A21C

OCL-Based Runtime Monitoring of Applications with Protocol State Machines

Authors: *Lars Hamann, Oliver Hofrichter, and Martin Gogolla*

Proc. 8th European Conference Modelling Foundations and Applications (ECMFA'2012)

The final publication is available at Springer via
`http://dx.doi.org/10.1007/978-3-642-31491-9_29`

OCL-Based Runtime Monitoring of Applications with Protocol State Machines

Lars Hamann, Oliver Hofrichter, Martin Gogolla

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany
{lhamann|hofrichter|gogolla}@informatik.uni-bremen.de

Abstract. This paper presents an approach that enables users to monitor and verify the behavior of an application running on a virtual machine (like the Java virtual machine) at an abstract model level. Models for object-oriented implementations are often used as a foundation for formal verification approaches. Our work allows the developer to verify whether a model corresponds to a concrete implementation by validating assumptions about model structure and behavior. In previous work, we focused on (a) the validation of static model properties by monitoring invariants and (b) basic dynamic properties by specifying pre- and postconditions of an operation. In this paper, we extend our work in order to verify and validate advanced dynamic properties, i. e., properties of sequences of operation calls. This is achieved by integrating support for monitoring UML protocol state machines into our basic validation engine.

1 Introduction

When one faithfully follows the Model-Driven Development (MDD) paradigm, abstract representations of all artifacts, in particular of code, are needed in form of models. Model-like descriptions can be used as central parts in the software development process and are considered to be a promising paradigm for effective software production. Models can be employed in all development phases and for different purposes. Consequently and despite all justified criticism, the Unified Modeling Language (UML) is playing a pivotal role as a modeling language. Nearly every software engineer understands at least the UML core concepts, while other more specialized modeling languages first need to be explained from the scratch. This central role of the UML can also be observed by looking for transformation approaches from UML to more formal and specialized languages or tools such as the Alloy [24] language, SAT [25] or model checkers [19].

When using UML models for abstractions of concrete software systems, model quality is important. It has to be ensured that the developed models correspond to the implementation to be abstracted from. Otherwise formal quality assurance techniques would verify some disconnected abstract model and not the concrete implementation. This is especially true, if the implementation is not fully generated from the model and finalized by a developer. This is currently the most common case.

In [17] simulation of the model is proposed in the overall process of model checking. The process is shown in Fig. 1 which is adapted from [17, p. 8]. Our contribution and extension to the process is shown in the parts having a grey background.

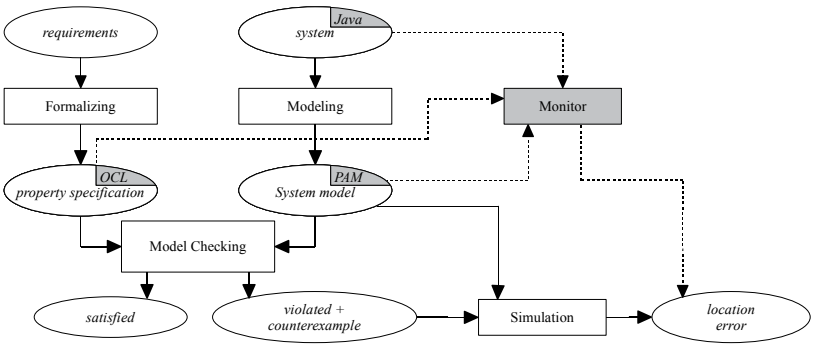


Fig. 1. Monitoring in the context of Model Checking (c.f. [17])

In our approach, we do not only simulate the model. We combine the system model with the implementation (the system) in order to be able to detect mismatches between the implementation, the system model and the property specification. We do so while executing the actual implementation. As systems we consider applications running inside a virtual machine, such as Java application running inside the Java Virtual Machine (JVM). Our system model will be defined as a UML class model extended by UML protocol state machines [20] and augmented with property specifications resp. assumptions formulated as OCL (Object Constraint Language) [21] state invariants and OCL operation pre- and postconditions. Since the elements of this system model need to be identified by our monitor, the system model needs to be aligned to the implementation. We call such a model a platform aligned model (PAM). We connect these components with a monitor in order to verify assumptions about the components at runtime. Our monitor can be started at any time that the concrete system, i. e., the Java application, is running. As an extension to our work presented in [15] and [16], we show how a state machine extension of the employed validation engine can be used without modifying our monitor component. Here, we show how UML protocol state machines (psms, singular psm) can be used to validate the correct sequence of operation calls, i. e., a protocol definition for a given class. We will further discuss some threads to validity which have to be considered when using a monitor approach like ours.

The rest of this paper is structured as follows. In Section 2 we put forward the basic ideas of our proposal for analyzing applications running in the Java virtual machine. Section 3 gives an overview on the integration of protocol state machines into our validation engine USE [11]. Section 4 explains the employment

of protocol state machines in combination with our monitoring approach by means of a middle-sized case study applied in our tool USE. Section 5 discusses related work. The paper ends with a conclusion and ideas for future work.

2 Monitoring

In this section we explain our monitoring approach. A more detailed description can be found in [15]. The main idea of our approach is to monitor a running implementation of a system and to extract a more abstract representation of the current system state into a validation engine. We call this abstract representation a *snapshot* of the system under monitoring (SUM), because in general it is a small subset of the artifacts of the running system. Since we want to focus only on central parts of the implementation we leave out unimportant parts. The basis for this snapshot is a model which is more abstract than the implementation, e. g., by defining associations which are not present in programming languages, but specific enough to be able to find relevant parts inside the SUM, e. g., by specifying concrete package names. Because of this alignment between the most specific platform model, e. g., byte code and platform independent models we call this model level *platform aligned model (PAM)*.

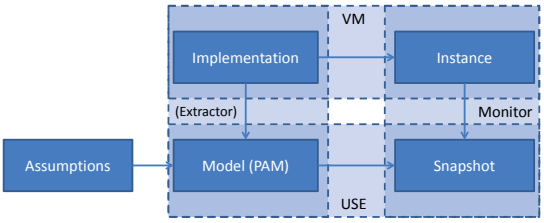


Fig. 2. Overview of the monitoring approach

As shown in Fig. 2 the PAM is enriched with assumptions about the running system. These assumptions are verified during the monitoring process by our validation engine USE. In order to be able to verify assumptions specified in a model USE needs an instance (i. e., objects and links) of it. In the monitoring context we call this instance a snapshot. Figure 2 shows this relation at the bottom. The monitor ensures, that the instance required by USE is a valid snapshot of the monitored instance inside the virtual machine. The virtual machine itself as shown at the top of the figure uses the implementation and an instance, i. e., the (heap) memory, stack, stack pointer, etc. of a running program. The PAM can be defined in several ways. For example, it can be step-wise refined when developing a system or it can be extracted by using reengineering techniques as shown in [16]. Furthermore, it can be generated when using model driven development.

Using modern virtual machine implementations like the JVM or the CLR of Microsoft .NET allows our monitor to use a rich pool of debugging and profiling interfaces. For example, the Java Platform Debugger Architecture[22] enables third party tools to easily access applications running inside a local or remote virtual machine. An important part of this interface is the possibility to retrieve information about instances of a specific type. This is used as an entry point for our monitoring approach described next.

First, the validation engine needs to be configured with the corresponding PAM and the SUM needs to be started. Next, the monitor needs to be connected to the running system. If the startup of a SUM is important, the user can also start the application with specific parameters, so that it suspends directly when started and is resumed only if the monitor signals this to the application. When the monitor is connected after the application is already running, the monitor creates a snapshot of the current system state. The following descriptions of the steps to create this abstract snapshot are explained in more detail in [15].

1. For all classes in the PAM which can be matched to an already loaded class in the VM, all existing instances of them are mapped to newly created instances of the platform aligned model.
2. For each created instance in the previous step the values of the attributes defined in the PAM are read. This step includes a mapping for values of primitive types to built-in OCL types, e. g., String and Real (c. f. [28]). Attribute values with a type of a class defined in the PAM need to be mapped using the mapping created in the first step.
3. For all associations in the PAM, links are created between corresponding instances.
4. By using the current stack-trace of the monitored system the current operation call sequence relevant to the monitored elements can be rebuilt. For this, the deepest operation call to a monitored operation (an operation specified in the PAM) on the call stack acts as an entry point for the following monitored operations on the call stack.

After such a snapshot has been constructed, the monitor needs to register to several events that occur in the VM in order to keep the snapshot synchronized with the running system and to allow a dynamic monitoring of the SUM. Currently our monitor makes use of the following breakpoint and watchpoint locations:

1. At class initialization to allow the registration of all other breakpoints. This ensures, that classes which were not loaded while taking the snapshot are also monitored.
2. At constructors of monitored classes. This allows the monitor to keep track of newly created instances and therefore enables an incremental construction of the system state in contrast to always construct a new snapshot of the running system when needed.
3. At the start of a monitored operation. This enables the monitor to validate preconditions at runtime and to follow the call sequence.

4. Just before the exit of an operation call. This enables the monitor to validate postconditions. The break must occur after the result of the operation is calculated.
5. When a monitored attribute is modified. A monitored attribute might be an attribute or association end inside of the PAM.

Monitoring an application in the presented way in combination with our validation engine USE allows a user to monitor the validity of UML constraints like multiplicities or composition properties, invaraints, pre- and postconditions *without* the need to modify the source code of the application or to use special bytecode injection mechanism. In addition, without changing the monitor component, improvements made to the validation engine can be used. For example after adding support for protocol state machines to USE, as described next, only the PAMs of the monitored systems needed to be extended to allow a more detailed monitoring of call sequences. Without the use of protocol state machines, only a very small part of a call sequence could be validated in one step, because OCL only allows access to the state just before an operation was called. Using protocol state machines it is possible to validate operation call sequences of arbitrary length.

3 Protocol State Machines in USE

The UML specifies two kinds of state machines: behavioral and protocol state machines [20]. As the name suggest, the former kind is used to specify the behavior of UML elements including actions attached to transitions to specify changes inside a system while taking a transition. The latter one specifies the allowed call sequences of a protocol. In USE we added support for protocol state machines in the context of a class. Following the general idea of USE, we start with a small well-defined subset of the many features for UML state machines. In the following we describe this implemented subset and its semantics.

First of all, all state machines in USE are flat, i. e., they have only one region and no composite states. They have only a single initial and a single end state. All other states are proper states and no pseudo states, which means that there are no forks or joins. States can have a state invariant which needs to be valid if a given psm instance is in the corresponding state. The context of a psm instance and also for the state invariant (accessed by using `self` in an OCL expression) is the instance of the context class of the psm which owns the psm instance. For the initial state only an unnamed transition or a transition with the event `create` is allowed as an outgoing transition. An initial state has no incoming transitions while an end state has no outgoing transitions. The transitions between states specify the valid call sequences of operations for the context class. As described in the UML the protocol state transitions between states consist of three parts:

1. the referred operation (`op`),
2. an optional guard (`G`), i. e., a precondition and
3. a postcondition (`PC`) which is also optional.

In an state machine diagram the transitions are labeled using the following schema: $\xrightarrow{\texttt{[G] op()/ [PC]}}$. A state can have multiple outgoing transitions that refer to the same operation. To be still able to choose a single transition the guard, post condition and state invariant of the target state for all transitions referring to the same operations are considered by USE. In some situation the usage of all this information still leads to multiple possible transitions. When USE encounters such a situation it reports an error to the user.

When an operation on an object whose class defines at least one psm is called, the selection of the transition to be taken for each psm is done in the following way. First, it is checked if the operation call needs to be ignored, i. e., no transition must be taken. This is the case if

- none of the transitions inside the protocol state machine covers the called operation (see [20, p. 545]) or
- the psm is not in a stable state, i. e., a transition is currently active.

If the operation cannot be ignored it is checked

- if at least one outgoing transition of the current state is enabled, i. e., the state has one or more outgoing transitions which refer to the called operation while having a valid precondition.

All enabled transitions are saved as possible transitions which could be taken after the operation call is completed. When the called operation finishes its execution, for all possible transitions the postcondition and the state invariant of the target state are validated. If only one transition fulfills the postcondition and the state invariant the transition is taken. Otherwise an error is reported which also explains if either no transition could be taken or multiple transitions would be possible.

3.1 State determination

One benefit of our monitoring approach is the possibility to connect to a monitored system at any time. While this allows a SUM to run without overhead until the monitoring starts, this ability leads to some issues to be considered. One major problem is the lack of information of previously called operations, so that all protocol state machine instances are in an undefined state. To allow a correct monitoring of psms it is important to determine the correct states of all psm instances. To be able to determine the states after an initial snapshot has been taken we use state invariants. These state invariants need to be well-defined because otherwise the snapshot would be in an unsound state. For example, all psm instances should be in a given state after the state determination check. In this context well-defined means that the state invariants should be independent of each other, i. e., at any state only one state invariant evaluates to true for every instance referring to the psm.

When using complex state invariants the task of verifying the independence of state invariants can be accomplished by using automatic model finding techniques. These are similar to the one presented in [13] which allows a user to

show the independence of invariants. In [13] the independence of invariants is slightly different form the independence of state invariants we want to achieve. In [13] an invariant is defined as independent if it cannot be removed without loss of information meaning, there exists at least one system state where this single invariant is violated. For the independence of state invariants required for the state determination, we consider state invariants as independent if for all system states only a single state invariant is fulfilled.

Formally, given the set of all possible system states $\sigma(M)$ of a Model M and the invariants $i_1, \ldots, i_n$ the independence of an invariant i_k is defined in [13] as

$$\exists \sigma \in \sigma(M)(\sigma(i_1) \wedge \cdots \wedge \sigma(i_{k-1}) \wedge \sigma(i_{k+1}) \wedge \cdots \wedge \sigma(i_n) \wedge \neg\sigma(i_k))$$

whereas in this work the independence of state invariants $i_1, \ldots, i_n$ for a single psm is defined as

$$\forall \sigma \in \sigma(M)(\sigma(i_k) \Rightarrow \neg\sigma(i_1) \wedge \cdots \wedge \neg\sigma(i_{k-1}) \wedge \neg\sigma(i_{k+1}) \wedge \cdots \wedge \neg\sigma(i_n))$$

However, the same validation techniques apply, but as the universal quantification indicates, a full verification requires a complete search through all possible system states, which implies the well-known state space explosion problem and is therefore not a trivial task and we restrict ourselves to checking occurring test cases.

4 Case Study

In this section we apply our extensions to the monitoring approach to the public available, mid-sized application we used in [15]. The case study will demonstrate the advantages of our approach.

- Assumptions about a running implementation can be validated without the need to modify the source code.
- The state of an implementation can be examined in an abstract way to discover inconsistencies or design decisions.
- Using protocol state machines the correct usage of the defined protocol of a class can be validated.
- Concrete usage scenarios can be visualized by means of a sequence diagram.

This will be exemplified by the following case study using an open source computer game called *Free Colonization*[1] or in short *FreeCol*. It is a modern Java-based implementation of the 1994 published game *Sid Meier's Colonization*[2]. The game itself is a round-based strategy game with the goal to colonize America and finally to achieve independence. The game takes place on a matrix-like map which consists of tiles with different types, e. g., water, mountain, forest.

[1] Project website: `http://www.freecol.org`

[2] The corresponding Wikipedia article gives detailed information about the game play. `http://en.wikipedia.org/wiki/Sid\_Meier\%27s\_Colonization`

Different units operate on this map and can explore unknown territory, build colonies, trade goods, etc. Figure 3 shows an example state transition of a running game. One unit (i. e., a pioneer) is placed in the center of the shown map on the left side and is surrounded by several different tile types. The right map shows the game state after the pioneer has build a new colony called Jamestown. The sketched state machines displayed below the two maps exemplify our new contribution. We want to be able to monitor the transition of the pioneer state from one state before she or he built a colony to another state after she or he joined the colony (note, that this is a single step in the game).

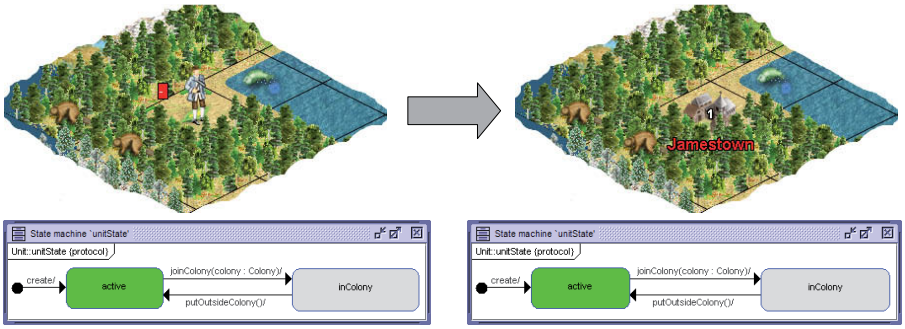


Fig. 3. Sample game situation in FreeCol

To be able to monitor this transition, we extended the PAM presented in [15] in a step-wise manner. First we added the enumeration `UnitState` to our PAM and defined a new attribute `state:UnitState` to the class `Unit` as shown in Fig. 4. The presence of this attribute simplified the definition of the state invariants as we will see later.

For our purpose the class diagram shown in Fig. 4 with an overall of 14 classes is detailed enough. When compared to the 551 classes which are present in version 0.9.2 of FreeCol we used for the monitoring this illustrates that the PAM for an application only needs to represent a small subset of the monitored implementation. Because we focus on state transitions we do not show any constraints defined for the PAM. Examples can also be found in [15].

Except for one case, we modeled attributes of a Java class which use a class present in the PAM as associations. The exception is the attribute `Tile::type` which reduces the number of links in an object diagram, but still allows to directly see that tiles differ in their type. For a first definition of a psm which monitors the entrance and the exit of a colony for a unit, we only need to consider the class `Unit` and its operations `joinColony(aColony:Colony)` and `putOutsideColony()` in combination with the attribute `state:UnitState`. These elements are present in the concrete implementation of the class `Unit` and can directly be monitored. The enumeration `UnitState` defines nine different enumeration literals which express different states of a unit. Since we are only inter-

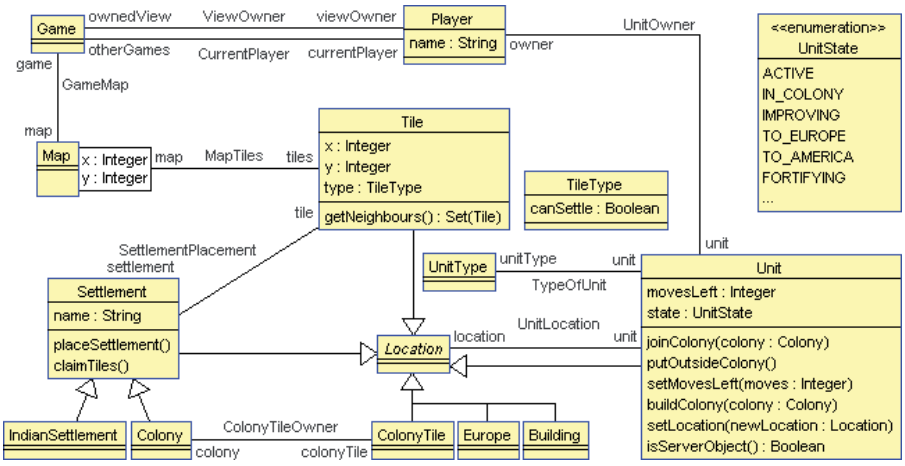


Fig. 4. Platform aligned model

ested in the state `IN_COLONY` and do not consider the other states we can specify a protocol machine with two states. One for the state `IN_COLONY` and one for all other game states. Our assumption about the protocol of the class `Unit` is that an operation call to `putOutsideColony()` is only valid after the operation `joinColony()` has been called on the same object sometime before.

To be able to set the correct state of a monitored instance we need to specify state invariants for these two states. As stated earlier, the presence of the attribute `state` for the class `Unit` simplifies this task, because we only need to check the value of the attribute to determine the current state after a snapshot has been taken. Therefore, the state invariant for the psm state `inColony` is `self.state = UnitState::IN_COLONY` and the other state invariant only changes the comparison from equal to not equal. Given the previously expressed assumptions, this leads to a psm which has two states and two transitions leaving out the transition for the creation. This psm is shown in Fig. 5. A `Unit` object starts in the state `active` after it is created and enters the state `inColony` when the operation `joinColony(colony:Colony)` was executed. If the operation `putOutsideColony()` is called the state changes back to `active`. Any other operation call to a unit instance is ignored as described in the UML specification for operation not mentioned in a psm. This means, the psm only allows a state change when one of the two operations is called.

Figure 6 shows the relevant part of the snapshot after connecting to the running game when it is in the game state shown on the left of Fig. 3 as an object diagram. The overall snapshot consists of nearly 6000 objects and 4000 links which makes it impossible to manually extract an informative object diagram. USE allows a user to select objects which should be shown or hidden in an object diagram by using several features. Two useful ones are the selection by an OCL expression and the selection of related objects by path length (see [12] for more

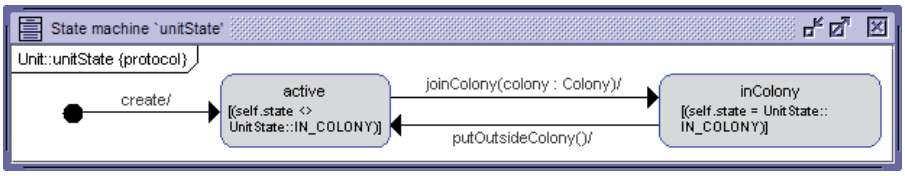


Fig. 5. Protocol state machine for the class Unit

information). The shown part of the snapshot is divided into two parts, which are important while validating the assumptions about the state transitions. Because we monitored a single user game on a single machine the instance of the game contains both, the data used by the game server and the client. By looking at the instances `Tile3466` on the server side and `Tile1583` on the client side one can see that the server part has more information about the game than the client part. Both instances represent the same tile on a map, because their positions are equal, but the client instance does not know of what type the tile is. To be able to determine if an object belongs to the server or client side we also monitored the class game with the association `ViewOwner`. If a game object is not linked to a player by this association it is the server game. The equivalent OCL expression (`self.owner.ownedView->isEmpty()`) is used as a body for the operation `isServerObject()` of the class `Unit`. This operation is marked as a query operation and is therefore ignored by the monitor. The object diagram further shows the owned units of the player named 'ada' and the object for the tile on which we want to build a colony (`Tile4228` resp. `Tile225`).

After taking this initial snapshot, the states of the protocol state machines for the existing unit objects need to be determined. This can be done by a single command in USE which also informs the user about objects for which the psm instance could not be set to a single state. This happens, if no state invariant or multiple state invariants evaluate to true w. r. t. the given snapshot. Because this state determination is a common task after a snapshot has been taken, the monitor plugin can automatically execute the state determination after the construction of a snapshot. After the states have been determined the states of the relevant units of the snapshot are as expected (`active`). After resuming the game and building the new colony *Jamestown* we get a valid sequence of operation calls which can be seen in the monitored sequence diagram shown in Fig. 7. We observed, that the execution of the operation `joinColony()` indeed leads to the attribute value `IN_COLONY` of the attribute `Unit::state`, because no violation of a transition is reported.

To get further information about our assumptions we can instruct USE to validate the current state invariants of all psm instances. After the validation of our current snapshot USE reports an error for the psm instance of the client object of the unit which has built the colony. This is due to the fact that the operation `buildColony` is only called on the server object and only the new values are transfered to the client object. Therefore, USE did not execute a

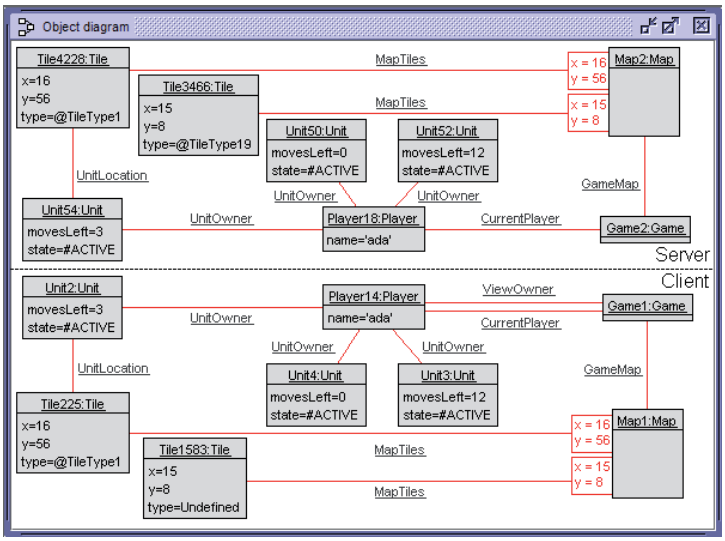


Fig. 6. Parts of the snapshot taken at runtime

transition from the source state `active` to the target state `inColony` for the client unit but monitored the change of the attribute `state` to `IN_COLONY`. Now, the new attribute value violates the state invariant of the state `active`.

Because the separation of the client and server objects seems to be a valid design decision we can ignore these violations and continue the monitoring process to retrieve further information about the validity of our assumptions. To test the defined protocol we use another unit and let it join and exit the colony. While executing this scenario another issue arises because entering an existing colony, i. e., a unit only enters a colony without building it before, does not lead to an operation call to `joinColony()`. Instead, only `setLocation()` is called which is not handled by the psm and therefore does not execute a transition keeping the psm instance in the state `active`, but the attribute value of the runtime instance is set to `IN_COLONY` which violates the state invariant of the state `active`.

Using this information a user of the monitor needs to decide where the error is located: in the implementation or in the PAM. For our example, we assume that the PAM needs to be modified although it seems to be an unsound usage of the `Unit` class. This assumption is backed by the fact, that the developers of FreeCol refactored this part of the game in newer releases. If we want to adapt our psm to the last discovered facts, we need to handle the client server separation and the additional operation calls. The modified psm is shown in Fig. 8. The additional operation `setLocation(newLocation:Location)` leads to two new transitions in the psm. Both transitions have as their source state the state `active` but differ in their target and guard. If the new location is of type `ColonyTile`, which represents special tiles related to a colony, the new state after the execution is

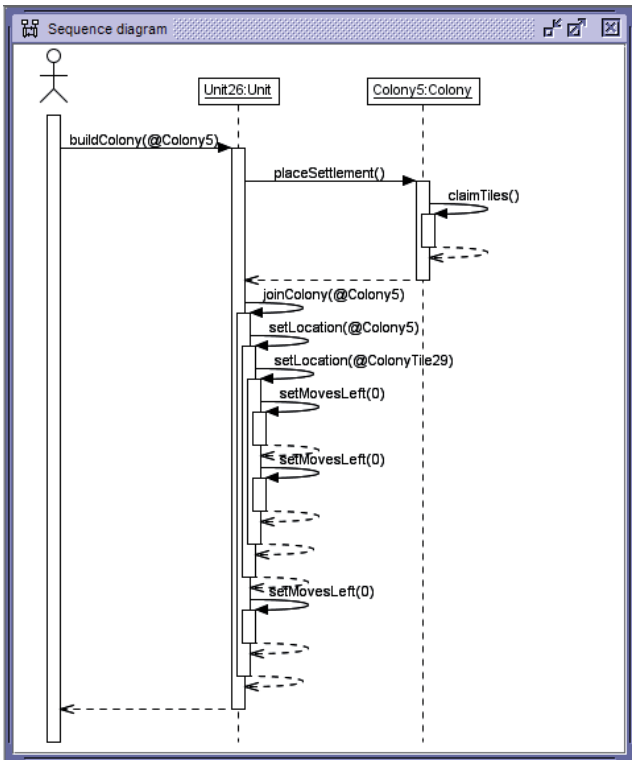


Fig. 7. Sequence diagram of the monitored execution

`inColony` otherwise the state does not change. Interestingly, when a `Unit` object leaves a colony this leads always to a call to `putOutsideColony()`.

However, the problem how to differentiate the server and client objects still exists. When taking a snapshot all state invariants should be independent to allow a valid determination of the current state. If we would introduce a new state for client objects with the state invariant `not self.isServerObject()` and add a new conjunction `self.isServerObject()` to the two existing state invariants the state determination would work, because each instance can be mapped to a single state. Either it is a client object or it is a server object and the two server states are independent because of the different comparison operators. The problem with this approach is the dynamic monitoring after existing instances are read. A new instance would be in the state `active` which is now only defined for a server object due to the new conjunction. If a new client instance is created this would violate the state invariant. Further, the new client instance would never change the state because none of the monitored operations is called for client objects as described before. Using an implies condition would

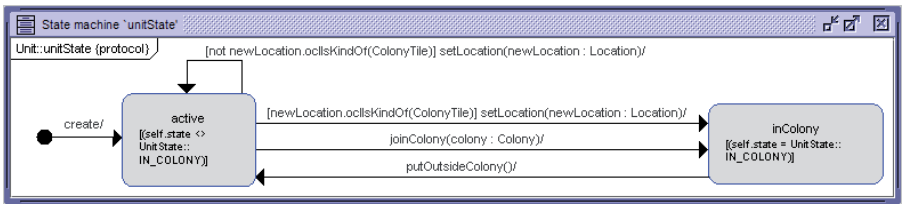


Fig. 8. Extended PSM for the class Unit

violate the independence of the two server states because for client objects both invariants would be fulfilled.

A simple solution would be to add a transition which covers the operation that specifies the new instance to be a client instance. However, FreeCol does this inside of the constructor which is represented as the `create` transition and the UML explicitly forbids multiple outgoing transitions for the initial state and also no post condition on it. If it was allowed the distinction could be done using postconditions on two different create transitions. Another solution for this is to use a change event on a transition. By specifying the change expression `not self.isServerObject()` a psm instance can move to a specify client state. A drawback of this solution is the relative high calculation cost of such change events. Because a change expression can generally access every object or property of the current snapshot all currently valid transitions with change events would need to be checked after every change in the snapshot. The costs can be reduced by using special analysis algorithms which calculate the change expressions that need to be checked after a change as presented in [7]. Currently, such a mechanism is not present in USE, but could be integrated in the future. For now, we need to ignore state violations of the client objects. Note, that the validation of the correct transitions for the server objects still works.

When using this modified psm all scenarios described above lead to the expected changes of the psm states. Beside the manual execution of observed game situations the presence of computer controlled players in the game can be used as a test driver. As with the manual play all analyzed operations are also used by computer controlled players. We used this to strengthen our PAM.

5 Related Work

In our previous work we focused on the runtime verification of static properties (like multiplicity constraints and invariants) of an application running on a virtual machine [15]. Different approaches for checking information extracted from a running system for certain properties exist. [10] and [2] make a comparison between these approaches. According to [10] most constraint validation techniques for Java are based on the design-by-contract-principle introduced by the Eiffel programming language. In contrast to our approach, the approaches compared to each other in [2] require a full access to the source code of the system under

monitoring. The Java Modeling Language (JML) is appropriate both for formal verification and runtime assertion checking [18].

In this paper, we extended our validation engine by support for UML protocol state machines in order to be able to verify and validate dynamic sequences of operation calls. Our approach applying protocol state machines differentiates from approaches which are based on the usage of regular expressions. Such an approach is presented in [5]. It enables programmers to define parameterized runtime monitors. For this purpose a temporal ordering over breakpoints, which are used for debugging purposes by programmers, is introduced. The temporal ordering is defined by regular expressions. Another approach uses tracematches for runtime verification [6]. As the previous approaches, this one is also based on regular expressions.

A UML protocol state machine as used in our approach is different from regular expressions through the information of transitions: protocol state machines provide the possibility to specify an initial condition (guard) under which an operation can be called. This possibility makes protocol state machines more powerful than regular expressions. The authors of [23] present an approach which applies UML protocol state machines to produce class contracts. For this purpose they define the structure and the semantics of UML protocol state machines.

With 'ocl2j' a tool exists which allows to enforce OCL constraints in Java through translating OCL expressions into Java code [9]. An analog approach is presented e. g. in [14]. From the authors runtime verification approach the tool 'INVCOP' has arised. The Dresden OCL toolkit makes available two distinctive approaches for OCL-based runtime verification [8]. While the 'generative' approach is based on the generation of AspectJ code, the 'interpretative' approach integrates the Dresden OCL2 Interpreter into a runtime environment in order to interpret OCL constraints.

In [4] the monitoring of state machines is focused while the usage of OCL is relinguished. With the 'aspect oriented approach', the 'listener approach' and the 'debugging approach', the authors describe three possibilities to extract runtime models.

To synchronize a running system with a runtime model the authors of [26] use 'synchronizers'. Thus the system can be changed immediately when the model has been updated and the model can be immediately adapted if the system progresses.

Java PathFinder (JPF) is a runtime verification and testing environment for Java developed at NASA Ames Research Center [27]. JPF is based upon a special Java Virtual Machine which is called from a model checking engine included in JPF. The authors of [1] present JPF-SE, a symbolic execution extension to JPF. The framework Polyglot has been integrated with the Java PathFinder [3]. Polyglot enables the execution of multiple variants of statecharts including UML statecharts and the verification of their models against properties. It uses an intermediate representation which is translated from a range of modeling tools. The intermediate representation is used to generate Java code representing the structure of a statechart which is analyzed by applying JPF.

6 Conclusion

We have presented an extension to our approach for monitoring assumed properties in form of OCL constraints for a running Java application. Based on this approach, which takes advantage of the powerful features of the Java virtual machine, we have added support for protocol state machines to the underlying validation engine. This allows us to specify assumptions not only formulated as class invariants or operation contracts, but also as state invariants. By using a protocol state machine, more knowledge about the history of an object is available because of the recording of states. We have shown that the definition of state invariants is important for our approach in order to determine the correct states of an object when connecting to a running system without the information about previous operation calls. We explained our work by a non-trivial example of an open-source game.

As future work we want to extend the support for protocol state machines within our validation engine. One major improvement would be the support for change events. To be applicable in practice, an efficient implementation is needed which considers only the transitions with an effective change event. A more detailed study of similar approaches, for example, based on aspect-orientation or approaches considering the Java Modeling Language (JML) as a target language, might introduce alternative features and our monitor could be improved in various directions. For example, one could consider abstract model breakpoints, which are configurable by the user or by extended information about elements that are only present within the running system. Last, but not least, comprehensive case studies must give more feedback about the applicability of our work.

References

1. Anand, S., Pasareanu, C.S., Visser, W.: JPF-SE: A Symbolic Execution Extension to Java PathFinder. In: Grumberg, O., Huth, M. (eds.) TACAS. LNCS, vol. 4424, pp. 134–138. Springer (2007)
2. Avila, C., Sarcar, A., Cheon, Y., Yeep, C.: Runtime Constraint Checking Approaches for OCL, A Critical Comparison. In: SEKE (2010)
3. Balasubramanian, D., Pasareanu, C.S., Whalen, M.W., Karsai, G., Lowry, M.R.: Polyglot: modeling and analysis for multiple Statechart formalisms. In: Dwyer, M.B., Tip, F. (eds.) ISSTA. pp. 45–55. ACM (2011)
4. Balz, M., Striewe, M., Goedicke, M.: Monitoring Model Specifications in Program Code Patterns. In: Proc. of the 5th Int. WS Models@run.time. pp. 60–71 (2010)
5. Bodden, E.: Stateful breakpoints: a practical approach to defining parameterized runtime monitors. ESEC/FSE '11, ACM, New York, NY, USA (2011)
6. Bodden, E., Hendren, L.J., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative Runtime Verification with Tracematches. J. Log. Comput. 20(3), 707–723 (2010)
7. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. Journal of Systems and Software 82(9), 1459 – 1478 (2009)
8. Demuth, B., Wilke, C.: Model and object verification by using Dresden OCL. In: Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice. pp. 687–690. Ufa, Russia (2009)

9. Dzidek, W.J., Briand, L.C., Labiche, Y.: Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In: Satellite Events at the MoDELS 2005 Conference. LNCS, vol. 3844, pp. 10–19. Springer, Berlin (2006)
10. Froihofer, L., Glos, G., Osrael, J., Goeschka, K.M.: Overview and Evaluation of Constraint Validation Approaches in Java. In: Proc. of ICSE '07. pp. 313–322. IEEE Computer Society, Washington, DC, USA (2007)
11. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69, 27–34 (2007)
12. Gogolla, M., Hamann, L., Xu, J., Zhang, J.: Exploring (Meta-)Model Snapshots by Combining Visual and Textual Techniques. In: Proc. 10th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'2011) (2011)
13. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, Independence and Consequences in UML and OCL Models. In: Dubois, C. (ed.) Proc. 3rd Int. Conf. Test and Proof (TAP'2009). pp. 90–104. Springer, Berlin, LNCS 5668 (2009)
14. Gopinathan, M., Rajamani, S.K.: Runtime monitoring of object invariants with guarantee. In: Runtime Verification, 8th International Workshop, RV 2008. LNCS, vol. 5289, pp. 158–172. Springer, Berlin (2008)
15. Hamann, L., Gogolla, M., Kuhlmann, M.: OCL-Based Runtime Monitoring of JVM Hosted Applications. In: Proc. WS OCL and Textual Modelling. ECEASST (2011)
16. Hamann, L., Vidács, L., Gogolla, M., Kuhlmann, M.: Abstract Runtime Monitoring with USE. In: Proc. CSMR 2012. pp. 549–552 (2012)
17. Katoen, J.P., Baier, C.: Principles of Model Checking. MIT Press (2008)
18. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. Sci. Comput. Program. 55(1-3), 185–208 (2005)
19. Moffett, Y., Beaulieu, A., Dingel, J.: Verifying UML-RT Protocol Conformance Using Model Checking. In: Whittle, J., Clark, T., Kühne, T. (eds.) MoDELS. LNCS, vol. 6981, pp. 410–424. Springer (2011)
20. UML Superstructure 2.2. Object Management Group (OMG) (Feb 2009), `http://www.omg.org/spec/UML/2.2/Superstructure/PDF/`
21. Object Constraint Language 2.2. Object Management Group (OMG) (Feb 2010), `http://www.omg.org/spec/OCL/2.2/`
22. Oracle: Java^{TM}Platform Debugger Architecture - Structure Overview (2011), `http://download.oracle.com/javase/6/docs/technotes/guides/jpda/architecture.html`
23. Porres, I., Rauf, I.: From Nondeterministic UML Protocol Statemachines to Class Contracts. In: Int. Conf. on Software Testing, Verification, and Validation. pp. 107–116. IEEE Computer Society, Los Alamitos, CA, USA (2010)
24. Shah, S., Anastasakis, K., Bordbar, B.: From UML to Alloy and Back Again. In: Ghosh, S. (ed.) Models in Software Engineering, LNCS, vol. 6002, pp. 158–171. Springer Berlin / Heidelberg (2010)
25. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In: Gogolla, M., Wolff, B. (eds.) TAP. LNCS, vol. 6706, pp. 152–170. Springer (2011)
26. Song, H., Huang, G., Chauvel, F., Sun, Y.: Applying MDE Tools at Runtime: Experiments upon Runtime Models. In: Models@run.time. pp. 25–36 (2010)
27. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model Checking Programs. Autom. Softw. Eng. 10(2), 203–232 (2003)
28. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley (2003), 2nd Edition

Publication A22C

On Integrating Structure and Behavior Modeling with OCL

Authors: *Lars Hamann, Oliver Hofrichter, and Martin Gogolla*

Proc. 15th International Conference Model Driven Engineering Languages and Systems (MoDELS'2012)

The final publication is available at Springer via
`http://dx.doi.org/10.1007/978-3-642-33666-9_16`

On Integrating Structure and Behavior Modeling with OCL

Lars Hamann, Oliver Hofrichter, and Martin Gogolla

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany
{lhamann,hofrichter,gogolla}@informatik.uni-bremen.de
http://www.db.informatik.uni-bremen.de

Abstract. Precise modeling with UML and OCL traditionally focuses on structural model features like class invariants. OCL also allows the developer to handle behavioral aspects in form of operation pre- and postconditions. However, behavioral UML models like statecharts have rarely been integrated into UML and OCL modeling tools. This paper discusses an approach that combines precise structure and behavior modeling: Class diagrams together with class invariants restrict the model structure and protocol state machines constrain the model behavior. Protocol state machines can take advantage of OCL in form of OCL state invariants and OCL guards and postconditions for state transitions. Protocol state machines can cover complete object lifecycles in contrast to operation pre- and postconditions which only affect single operation calls. The paper reports on the chosen UML language features and their implementation in a UML and OCL validation and verification tool.

Keywords: Structure modeling, Behavior modeling, UML, OCL, Protocol state machine, State invariant, Guard, Transition postcondition

1 Introduction

Executable UML [23] is designed to specify a system at a high level of abstraction, independent from specific programming languages and decisions about the implementation. Executable UML follows the ideas of the Shlaer-Mellor methodology, which separated concerns about the structure [34] and the behavior [33] of a system to be developed. It is defined as a profile of the Unified Modeling Language (UML) [26]. Executable UML models are testable, and can be compiled into less abstract programming languages to target a specific implementation. Executable UML supports model-driven development (MDD) through specification of platform-independent models. The approach proposed in this paper follows these ideas.

When using Executable UML, a system is decomposed into multiple modeling sub-languages: A class diagram defines the system structure in terms of the classes and associations; a state machine defines the states, events, and state

transitions for a class instance; an action language defines the actions or operations that perform processing on model elements; the system behavior is determined by the state machines and the operations realized in the action language.

Our tool USE (UML-based Specification Environment) supports the development of class diagrams by validating OCL class invariants and operation pre- and postconditions [7, 8, 19]. Recently, the tool was extended with an action language [3] which is based on the Object Constraint Language (OCL) [27, 36]. The present contribution explains our support for state machines in order to complete the description of behavior. Within our tool, we integrate class diagram validation with UML protocol machine validation on the basis of OCL state invariants and OCL guards and postconditions for transitions. In contrast to Executable UML, our approach extends OCL in order to express actions and operation implementations, but does not need to define a separate action language.

The need for integrating structure and behavior modeling in the OCL context arose from monitoring running Java applications in terms of UML class diagrams and OCL constraints and our state machine approach. In [12] we describe the monitoring of a non-trivial Java application with constraints. Other applications of our state machine implementation include middle-sized example models.

The rest of this paper is organized as follows. Section 2 introduces with a running example the main state machine features which we employ on the type level (at design time). Section 3 puts the state machine features which we handle in the context of UML and our implementation. In Sect. 4, model validation of state machines in connection with class diagrams is discussed on the instance level (at runtime). Section 5 connects our contribution with related work, before we conclude in Sect. 6.

2 Structure and Behavior at Design Time by Example

Our running example describes a digital support system for a library. The structural system requirements are shown in form of a UML class diagram in the top of Fig. 1. The system supports the administration of users, book copies, and books represented by respective classes and appropriate attributes. Two associations can establish object connections: the association `Borrows` between the classes `User` and `Copy` is meant to express that a `User` object has currently borrowed a `Copy` object, and the association `BelongsTo` between the classes `Copy` and `Book` expresses that a `Copy` object is an exemplar of a particular `Book` object. Further properties are specified by restricting multiplicities, role names (in the example, class names with lower first letter) and invariants (e.g., uniqueness requirements for the attributes `name`, `signature`, and `title`, as well as a range restriction for the attribute `year`). All classes possess operations for initializing objects. The association `Borrows` can be manipulated from both participating classes through the operations `borrow` and `return`. In order to support easy recognition of operation names, the first letter of the respective class has been added to these names (`borrowU`, `returnU`, `borrowC`, `returnC`). The `return` operations also modify the attribute `numReturns`.

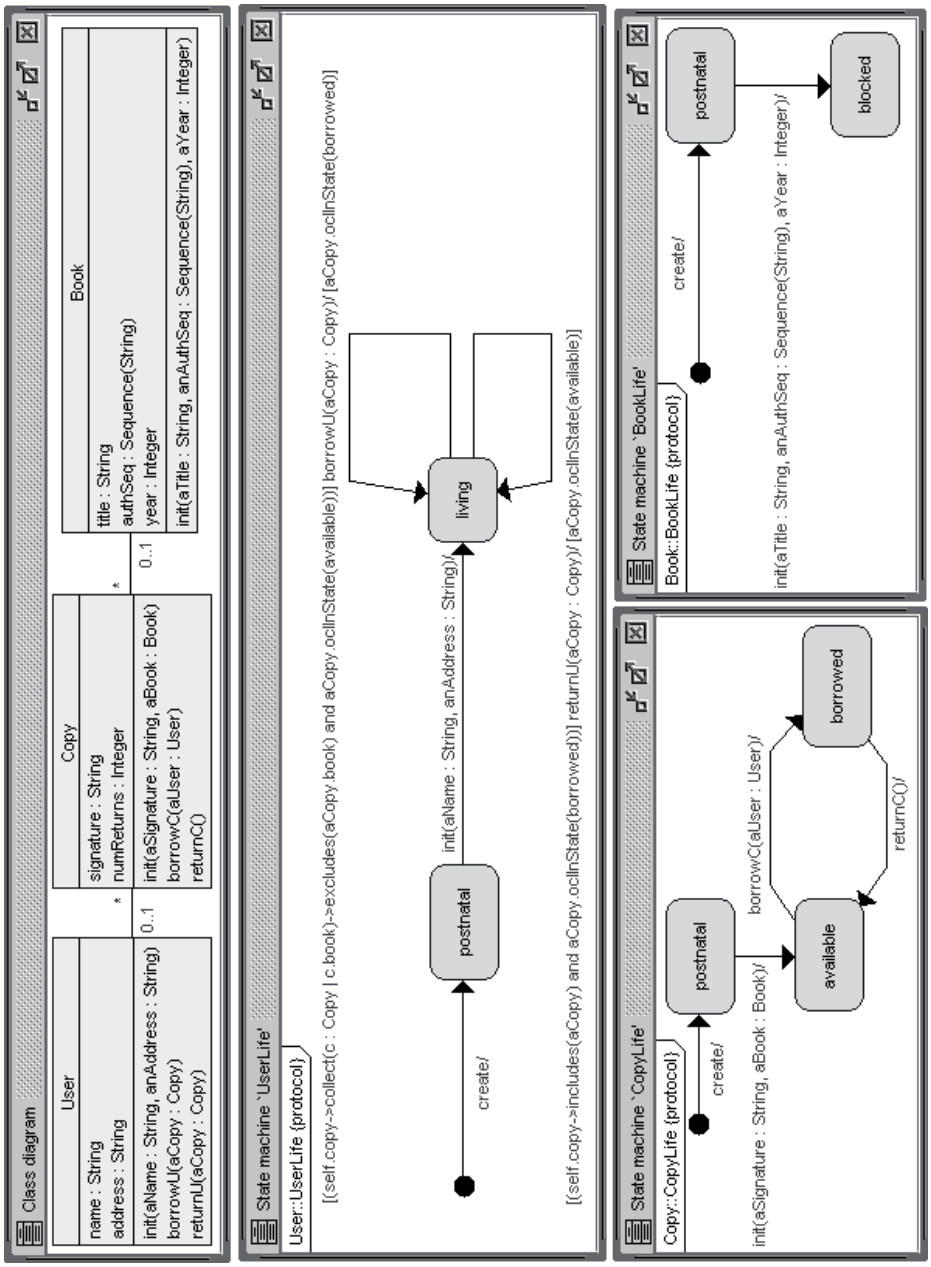


Fig. 1. Example System Requirements for Structure and Behavior (Design Time)

The behavioral system requirements are shown in the bottom of Fig. 1 as UML protocol state machines possessing states and transitions. For every class, the valid object lifecycles are depicted, which restrict the order of creation events and operation calls. As a central means to make the model precise, OCL is used in various places: States are described by state names and state invariants in form of boolean OCL expressions; transitions include (a) the triggering create or call event, (b) a guard in form of a boolean OCL expression asserting that the transition only takes places when the guard holds, and (c) a postcondition in form of a boolean OCL expression asserting that the transition only takes place in the case that after the transition the postcondition holds. Traditionally, the notion guard is used in connection with state machines; however, because of the symmetric behavior of the guard and postcondition, the guard may also be called transition precondition.

The state invariants may optionally be shown in the protocol state machine diagrams, however, we have suppressed them here. For example for the class `Book`, the two proper, non-pseudo states possess the following state invariants.

```
postnatal [title.isUndefined and authSeq->isUndefined and
           year.isUndefined and copy->isEmpty()]
blocked   [title.isDefined and authSeq->isDefined and year.isDefined]
```

In state `postnatal` (after `create`), all attributes must be undefined and the book must not be linked to any copy. In state `blocked` (after a call to the initialization operation `init`), all attributes are defined, but note that no statement about the linked copies is made, because there may or may not be copies for that book in the library (either `copy->notEmpty()` or `copy->isEmpty()` may hold).

The transitions are either labeled with the `create` event which brings the respective object into life or with an event which calls an operation of the object. The protocol state machine for the class `Book` asserts a finite lifecycle demanding that after object creation only the operation `init` may be called once. The state machine for class `Copy` guarantees that after creation and initialization, the `borrowC` and `returnC` operations switch between the states `available` and `borrowed`. The state machine for the class `User` is the only one employing OCL for transition guards and postconditions. But please be aware of the fact that all states are accompanied by OCL state invariants. Both operations, `borrowU` and `returnU` in class `User` are allowed in state `living`, however, OCL restrictions via transition guards and postconditions apply. The guard (precondition) for `borrowU` guarantees that a user cannot borrow two copies of the same book, for fairness reasons. And the guard asserts that only available, not borrowed copies can be handled with the operation `borrowU`. The postcondition of `borrowU` checks that the copy, which was available before the transition took place, is now unavailable. Conversely, the guard for `returnU` asserts that the copy to be returned belongs to the current user and is indeed a copy in state `borrowed`. The postcondition checks that the parameter copy is indeed `available` after the `returnU` call. Note that these simple example restrictions do not guarantee unproblematic behavior in all possible implementations. The state invariants, guards, and postconditions have been chosen for demonstration purposes.

An implementation on the modeling level of the operations can be realized in our language SOIL (Simple OCL-based Imperative Language) [3]. Such an implementation is indispensable for animating and validating the model. SOIL allows the developer to make system state manipulations with attribute assignments, object and link creation and destruction, and control flow using conditionals, loops, and operation calls. As an example, we show implementations for the operations of the classes `User` and `Copy`.

```
class User -- pre- and postconditions not shown
operations
  init(aName:String,anAddress:String)
    begin self.name := aName;
    self.address := anAddress; end

  borrowU(aCopy:Copy)
    begin aCopy.borrowC(self); end

  returnU(aCopy:Copy)
    begin aCopy.returnC(); end
end

class Copy
operations
  init(aSignature:String, aBook:Book)
    begin self.signature := aSignature; self.numReturns := 0;
    insert (self, aBook) into BelongsTo; end

  borrowC(aUser:User)
    begin insert(aUser, self) into Borrows; end

  returnC()
    begin delete(self.user, self) from Borrows;
    self.numReturns := self.numReturns+1; end
end
```

These operation implementations allow the developer to build up simple or complex test states and scenarios with call sequences easily. Consequently, model properties like consistency or the reachability of protocol states can be checked with scenarios constructed with SOIL statements. The SOIL command sequence in the upper right side of the forthcoming Fig. 3 is an example for such a test scenario. The validity of model properties formulated in OCL as class invariants, operation pre- and postconditions, state invariants, and transition pre- and postconditions is checked against these scenarios and by this also against the SOIL implementation given for the operations. When writing down a particular test scenario, the developer will have expectations on particular (class or state) invariants and (operation and transition) pre- and postconditions. These informal expectations are formally checked by the tool USE, and the validation results give detailed feedback to the developer about the possible discrepancy between

her expectations and the actual facts: *What you write down doesn't mean exactly what you think it means. And when it does, it doesn't have the consequences you expected.* [15, p. XIII]

3 Behavior Modeling with Protocol State Machines

3.1 Protocol State Machines in UML

The UML defines two different kinds of state machines: *Behavioral state machines* and *protocol state machines* [26, p. 535]. As the name suggests, the former can model the behavior of a model element by specifying actions which are linked to state transitions, whereas the latter focus on the specification of correct usage protocols, leaving out concrete actions associated with transitions [26, p. 547]. These protocols can be specified for any model element of type *Classifier* [26, p. 544]. The metamodel for state machines provided by the UML allows to model highly structured state machines composed of, for example, composite states, multiple regions and substate machines. At the current stage, our approach supports only a well-defined subset of these features leaving out mainly concepts to structure state machines, but allowing nearly the same expressiveness. Issues arising from the high structuring possibilities can for example be found in [21]. Next we describe the protocol state machine language as implemented in our work. Starting with the syntactical and semantical rules defined in the UML, we continue by showing the current features supported in our approach and how they are interpreted at runtime.

As other languages for (finite) state machines the core part of the state machines defined by the UML are states and transitions. The UML distinguishes between concrete and pseudo-states [26, p. 536, 549, 559]. A state machine instance cannot have a pseudo-state as its current state after a transition has been completed. Pseudo-states are only traversed during the execution of a transition. One example of such pseudo-states are choice points for a transition. Both kinds of states are derived from the metatype *Vertex* for which directed transitions are defined. Behavioral state machines consist of transitions which need a source and target vertex. In addition, transitions can specify a trigger (e.g., a call event), a guard and an effect, i. e., a behavior [26, p. 536].

As we will see, several parts of state machines can be enriched with additional boolean OCL expressions in order to add additional constraints. States can be enriched with a OCL state invariant which characterizes the state in more detail. The state invariant for a given state must be true, if a state machine is in this state. An OCL guard of a transition must be true to be able to execute this transition. For example, this allows to separate two outgoing transitions from one state with the same trigger. In protocol state machines it is also allowed to specify a boolean OCL expression which describes the system state after a protocol transition has been taken. This expression is called a postcondition of the protocol transition.

The initial pseudo-state together with a single outgoing transition marks a concrete state as the default state of the state machine. The transition from

the initial state to the default state can only define a behavior and no trigger or guard [26, p. 550]. Furthermore, the initial state, as all other pseudo-states, cannot specify a state invariant, whereas concrete states can.

Transitions inside a protocol state machine are defined by the metaclass *ProtocolTransition* [26, p. 546]. This class extends the transition class of the behavioral state machine and makes some extensions and restrictions. The main restriction for protocol transitions is that they cannot specify an effect, because they specify the usage of a protocol of a class and not its behavior. An effect of a transition is instead specified in a declarative way by means of a postcondition which cannot be specified for ordinary transitions. The trigger of a protocol transition is usually an operation call, but it can also be an event.

When a protocol state machine defines at least one transition, which refers to an operation, a call to this operation is only valid, if there exists a currently valid transition for this call event. If an operation of the owning class is not referred by a protocol state machine, a call to this operation is valid for any state of the state machine [26, p. 549]. The specification of events other than call events inside a protocol state machine defines requirements for the environment using the owning class, stating that the event can only be sent to an instance of the owning class under the current conditions specified by the protocol state machine [26, p. 549]. An additional constraint specified for a transition is usually called a guard, but for protocol transitions the naming is aligned to the area of operations, calling this constraint a precondition.

3.2 Supported Concepts for Behavior Validation

Our approach supports protocol state machines which allows to specify valid call sequences for lifecycles of an instance. A protocol state machine is defined in the context of a class. The concrete syntax of such definitions is shown below.

```
class A
 attributes
   ...
 operations
   ...
 statemachines
    psm ALife -- psm: Protocol State Machine
     states
      s_i:initial
      s_k [ state_invariant_k ]
      ...
      s_n:final
    transitions
      s_src -> { [ pre_cond ] call_event [ post_cond ] } s_trg
      ...
    end
end
```

First, more than one state machine (in the following we use the term state machine to refer to protocol state machines) can be specified for a class. Beside a name, each state machine defines two sections: `states` and `transitions`. The state section contains the definition of the pseudo- and the concrete states. A state machine must define exactly one pseudo-state of type *initial* acting as the entry point of the state machine. As already mentioned, the initial state cannot define any information except a name for the state. Concrete states are defined by their names and an optional state invariant expressed as a boolean OCL expression in the context of the owning class. State invariants will be discussed in detail during the description of the runtime behavior of state machines. Beside the concrete states and the initial pseudo-state, multiple final states can be defined.

The transition section specifies the structure of valid call sequences to the owning class. The textual syntax is aligned to the graphical representation in the state machine diagrams. For transitions, the source (`s_src`) and target state (`s_trg`) separated by an arrow (`->`) are mandatory. Except for the outgoing transition from the initial state, a `call_event` is also mandatory. These call events refer to an operation of the owning class. The call event for the outgoing transition of the initial state can either be left out or must be named `create` because a newly created object in our approach is immediately initialized with instances of all defined state machines for its class. The `call_event` can be surrounded by a pre- and postcondition given as a boolean OCL expression. Like pre- and postconditions for operations they can access the context object (the instance receiving the call event) and the parameter values of the call event. The postcondition can additionally make use of the OCL `@pre` keyword to access the values which were valid when the call event was triggered.

When a USE model containing state machines is loaded, static checks are made. These include checking the uniqueness of state names inside a single state machine and the well-formedness of transitions, i.e., checking that state names and transitions do refer to existing states and operations.

3.3 Protocol State Machines at Runtime

To validate a specified model, our approach allows the developer to instantiate it and observe its behavior. The instantiation can be done in several ways, e.g., by manually manipulating the system state using the graphical user interface or shell commands or by specifying statements in SOIL [3]. If an object of a class is created, which contains state machines[1], it is linked to the corresponding state machine instances. These state machine instances are initialized with the default state, i.e., the state reached by the outgoing transition of the initial state, as their current state.

If an operation is called on an object, all state machines, which specify a transition referring to the operation call, are checked for enabled transitions. A

[1] In the following we refer to objects of classes with defined state machines when using the word object.

transition is called *enabled*, if it is an outgoing transition leaving the current state of a considered state machine instance, if it refers to the called operation and if it has a currently valid precondition [26, p. 584]. If at least one enabled transition for each state machine under consideration exists, the operation call is valid. The transition to take is determined after the operation has been executed. This is done by evaluating for each previously enabled transition the postcondition and the state invariant of the target state. For each considered state machine instance there must be exactly one transition fulfilling both conditions. By using this mechanism, we (currently) disregard non-deterministic state machines and executions which are however generally allowed in UML. Otherwise, the operation execution is invalid. The concrete error situation is reported to the user stating that either there exists no valid transition or multiple transitions are currently valid. When a state machine instance is currently in an unstable state, i.e., it is executing a transition, all nested operation call events need to be ignored. Otherwise, a call to another operation on the same object by a called operation could for example change the current state making the previously enabled transition invalid. The modeler can turn on a notification mechanism for such situations.

The explained runtime behavior of state machines lead to valid call sequences respecting state invariants, transition pre- and postconditions, if the state of an object is only modified by operations specified by protocol state machines. However, as we described earlier, a protocol state machine can leave out operations, making them callable at any time. Because these unconsidered operations could also modify the state of an object, it is not guaranteed that a state invariant stays valid while a state machine instance remains in a certain state. Therefore, our approach is able to validate state invariants after any change to the system state, e.g., attribute assignments or link creations. A violation of state invariants is immediately reported to the user, who can then react to the error.

Another unique feature of our approach is the possibility to determine the current state of the state machines by the specified state invariants [11]. For this, the validation of transitions and state invariants can be suppressed. After a system state is constructed without the validation of state machines, the user can invoke the state determination command. The command tries to determine the current state for each state machine instance by evaluating its state invariants. If exactly one state invariant of a state machine instance evaluates to true, the state of this instance is modified. This can, for example, be used, if a given system state needs to be constructed without the execution of operations and afterwards an operation call sequence has to be validated. An application of this mechanism is the USE monitor [10, 12] which allows to connect to a running Java application and to retrieve a snapshot of the current application state. When connecting to the application, not all information about previously called operations is available, and therefore the current states must be calculated to obtain the valid state machine configuration.

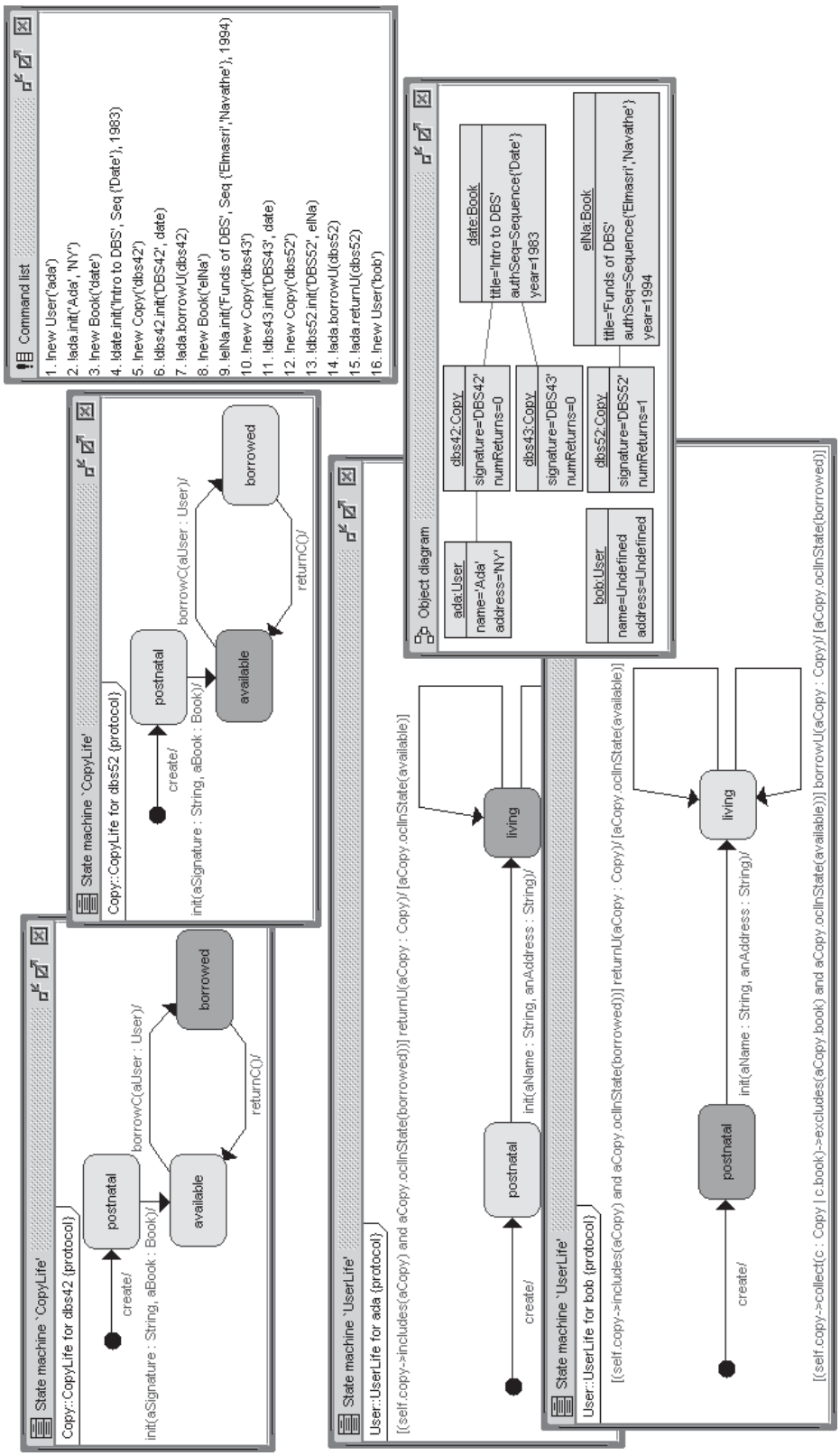


Fig. 2. Example Scenario for Structure and Behavior (Runtime)

4 Structure and Behavior at Runtime by Example

This section will explain how to apply the proposed concepts for the example. Whereas Fig. 1 pictures structure and behavior of the library system on a type level (design time), Fig. 2 displays structure and behavior of one system test scenario on the instance level (runtime). The object diagram in the lower right

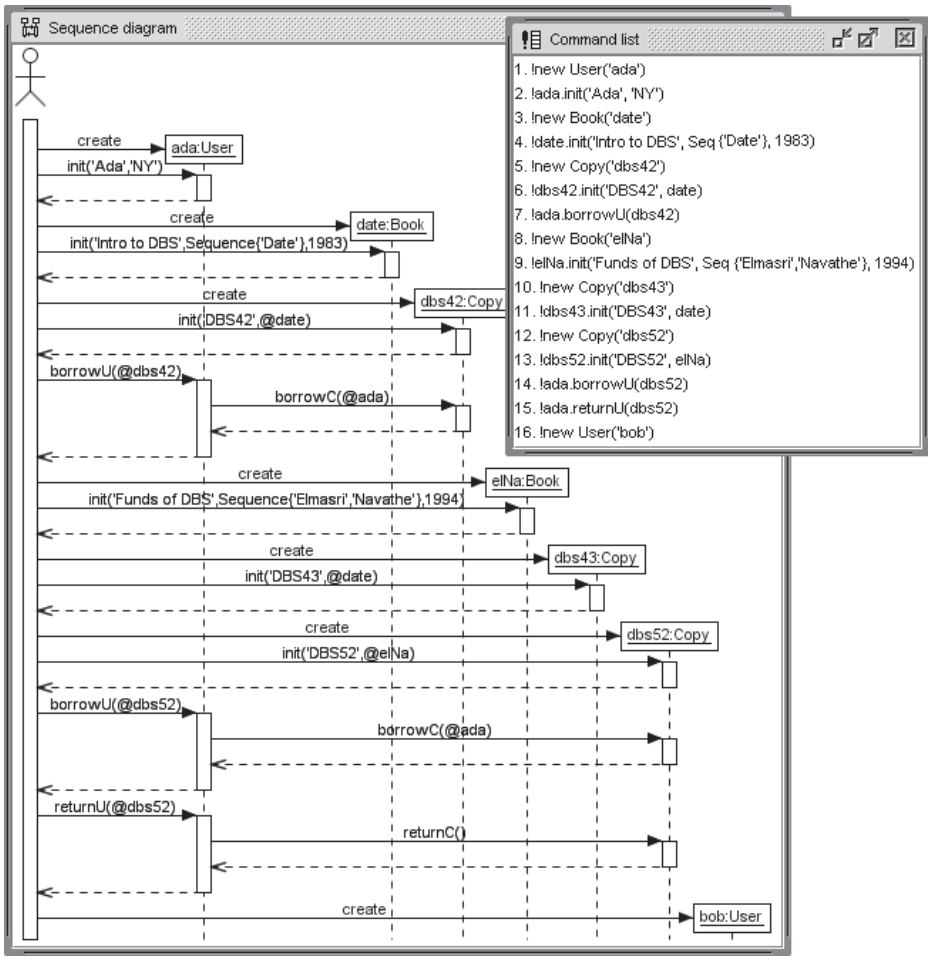


Fig. 3. Sequence Diagram and SOIL Commands for Example Scenario

represents the objects, their attribute values and links after the SOIL command sequence in the upper right part of Fig. 3 has been executed. In the left of Fig. 2, the upper two state machine instances show the current protocol state for the `Copy` objects `dbs42` and `dbs52`, respectively. Also in the left, the lower two state

machine instances display the current protocol state for the `User` objects `ada` and `bob` in dark grey. Please note, that the state of both `Copy` objects and the state of both `User` objects are different. The state sequence which the `Copy` object `dbs52` went through was `postnatal`, `available`, `borrowed` and again `available`. We can conclude this from the executed operation sequence and from the attribute value `1` for attribute `numReturns`. In the shown operation sequence, all OCL restrictions have been checked and no violation occurs: all class invariants, state invariants and transition pre- and postconditions have been evaluated to `true`. Please note, that full OCL support in our approach means that we can relate OCL queries concerning structure with behavioral descriptions, for example, the OCL query in Fig. 2 checks relevant `Copy` properties and these can be compared with the current protocol state and the value of the state invariants.

This scenario can be extended by further operation calls. For example, the `User` object `ada` could try to borrow the `Copy` object `dbs43`. In this situation, the guard for the `borrowU` call on the transition from `living` to `living` would prevent the transition to take place: User `ada` has already borrowed another copy of the `Book` object `date`. On the USE shell, a message will inform about the violation and the fact that the transition should not and will not occur. The following message will be shown.

```
!ada.borrowU(dbs43)
 >> Error: No valid transition available in protocol state machine
 >> 'User::UserLife [current state: living]' for operation call
 >> User::ada.borrowU(dbs43) due to failing transition guard.
```

Analogous error messages would be displayed on the shell, if the transition postcondition or the state invariant of the next state would be violated. Summarizing we can say that taking a transition may be aborted due to four possible reasons:

- a failing transition guard (precondition),
- a failing transition postcondition,
- a failing state invariant in the resulting state, and
- non-deterministic transitions, e.g., multiple transitions for the same trigger.

In Fig. 4, another example explains the usage of state invariants and the state determination option. For a `TrafficLight` class with three boolean attributes representing the red, yellow, and green bulbs, a protocol state machine allows the traffic light to step through four phases, where each phase is represented by a single state and a state invariant in form of an OCL expression characterizing the signal in terms of the bulbs.[2] The object diagram shows four test traffic lights equipped with randomly determined attribute values for the bulbs, not all representing valid signal configurations. The attribute values have been modified not by operations, but with direct attribute assignments.

[2] The phases are the phases used in Germany, whereas in other countries, e.g., in Italy, the phases are different.

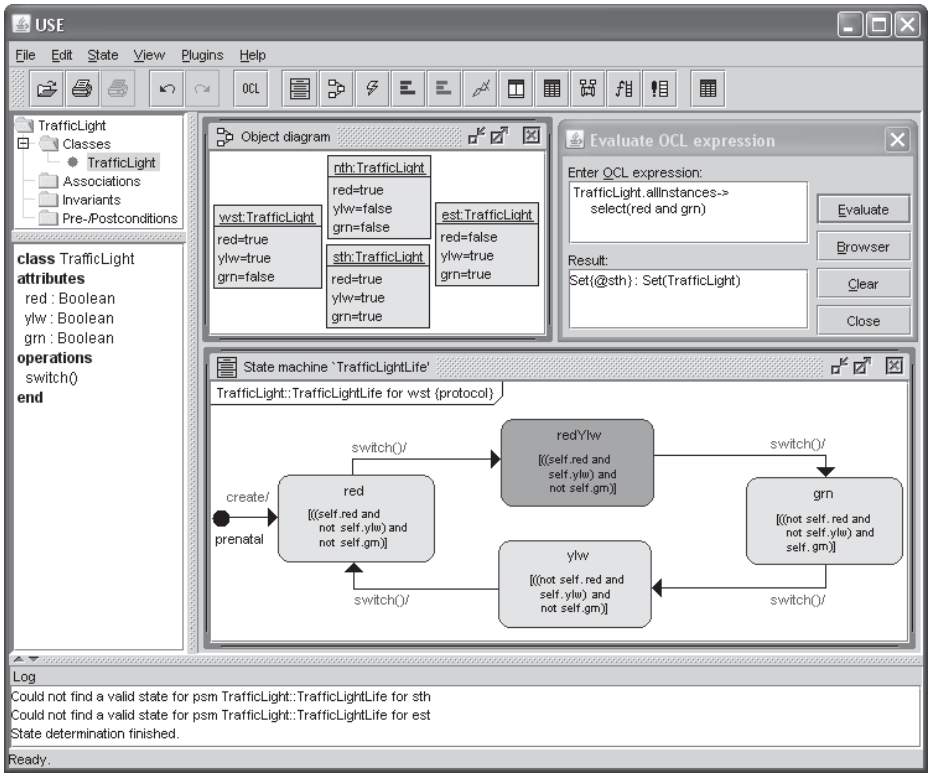


Fig. 4. Example for Usage of State Invariants and State Determination Option

In the log window at the bottom, the result of executing the state determination command is given. This command aims to bring the state machine instances into the state corresponding to their state invariants, if possible. The command can be issued through an entry in the 'State' menu. For two traffic lights (`sth` and `est`), a valid state fitting one of the four state invariants could not be found; the other state machine instances are moved into a state determined by a state invariant. The displayed state machine instance in the middle belongs to the `TrafficLight` object `wst` and shows that the attribute values (`wst.red=true` and `wst.ylw=true` and `wst.grn=false`) fit to the OCL state invariant expression (`self.red and self.ylw and not(self.grn)`) belonging to the current state `redYlw` shown in dark grey. As our approach supports OCL during all development phases, the complete system state can be inspected with OCL expressions at any point in time. The OCL query expression in the upper right retrieves all present traffic light objects which currently show both `red` and `grn`. The state determination together with OCL querying allows to check positive and negative test cases with respect to structure (objects and attributes) and behavior (operations and state machines).

5 Related Work

Specifying behavior in OCL OCL not only allows for specifying structural model features but also constraints on the behavior of objects by means of pre- and postconditions. In order that pre- and postconditions can be interpreted unambiguously, a detailed semantics of operation specifications is needed. The approach in [14] addresses this. However, according to [16], pre- and postconditions describe static aspects of the system, as they compare states of a system, which are static entities. Therefore in [16, 17] the so-called action clause is introduced to the Object Constraint Language and is provided with a semantics.

Semantics of state machines In our approach we use UML protocol state machines to constrain the model behavior. The structure and the semantics of protocol state machines are discussed in [28]. The authors present an approach which applies protocol state machines to produce class contracts. The semantics of behavioral state machines is discussed in [20]. The authors apply the semantics for validity proofs of refinement transformations on behavior state machines. A formal semantics for the integration of UML statecharts into OCL, which makes it possible to formulate expressions over states in UML statecharts is presented in [5]. However the authors refer to an older UML version, whereby postconditions of protocol state machine transitions are not handled. The dynamic semantics of state machines is discussed in [2].

Usage of state machines Different approaches for the usage of state machines in the software testing context exist. Model-based testing (MBT) tools often use UML state machines as a basis for automatic test case generation. The approach in [38] makes it possible to automatically generate state machine diagrams from use cases. This approach is also implemented in a tool and evaluated in different case studies. The approach in [31] applies behavioral state machines for modeling reactive systems and automatic generation of test cases. Based on this, the input-output conformance of the systems is tested. The presented test approach is implemented by the so-called TEAGER tool suite. In [37], the authors report on an industrial cooperation for model-based testing applying UML state machines with a German rail engineering company. Based on a given UML state machine this approach makes it possible to automatically generate unit tests. The use of UML state machines for requirements validation is described in [25]. The authors apply Formal Concept Analysis (FCA) in analyzing the association between a set of test scenarios with a set of transitions specified in a UML state machine model. The authors of [35] use protocol state machines in the field of network security. They introduce Veritas, a tool which uses applications network traces to automatically generate protocol state machines. The generated state machines are able to represent incomplete knowledge about a protocol and are labeled as probabilistic protocol state machines (P-PSM). K-statecharts are an extension of UML statecharts which allow the use of knowledge-logic formulae in the statechart transition guard and are used for runtime verification of system behavior [4].

Tools In [32] a tool set which supports static and dynamic validation of UML models is presented. The tool mOdCL is based on Maude, an executable formal specification language and is able to validate invariants and pre- and postconditions during the execution of a system [29]. In contrast to our approach and like the tool set presented in [32], mOdCL leaves out handling and runtime validation of protocol state machines. In [29], the authors report on the experiences with the development of a tool for dynamic enforcement of OCL constraints. Applying aspect-oriented programming (AOP), ocl2j automatically instruments OCL constraints in Java programs. In [24] a prototype of a tool being able to check the conformance of components within the UML extension for real-time (UML-RT) to the respective protocol state machines, which specify the legal communication between components, is described. Rhapsody is a verification environment for UML models. The tool implements an own semantics of statecharts, as discussed in [13]. The tool TABU allows for verification of reactive systems behavior [9]. For this purpose the behavior is modeled by state machines and automatically transformed into the used formal specification SMV (Symbolic Model Verifier). Additionally a number of CASE tools suchs as [6] allow for modeling statecharts, but are not able to validate state machines at runtime. In contrast to our approach, [1] and [22] don't provide full OCL support. Epsilon [18] is a platform which allows for model validation. However handling for state machines is not integrated.

Our contribution profits from these related works. It is however the only one which combines state machine validation with full OCL support for structural modeling and validation.

6 Conclusion

We have made a proposal for integrated structure and behavior modeling and validation. Full OCL support for (class and state) invariants and (operation and transition) pre- and postconditions guarantees that the underlying graphical models become precise. We combine descriptive requirements with an OCL-like imperative language. The models are validated and verified by test scenarios.

We plan to extend the supported UML state machine features, in particular, we will care for structuring mechanism like nested states. A number of improvements on the user interface can be realized, for example, an optional indication of protocol state machine states on object lifelines in sequence diagrams. Features of the behavior models like state reachability and other dynamic properties like liveness could be supported in a (semi-)automatic way. Consistency, redundancy and other relationships between the structural and behavioral model features should be investigated. Methodological questions about the usage of (class and state) invariants, and (operation and transition) pre- and postconditions must be discussed. Last but not least, larger case studies must give further feedback about the applicability and efficiency of the approach.

References

1. Abstract Solutions Ltd: Executable UML (xUML). Internet, http://www.kc.com/XUML/
2. Börger, E., Cavarra, A., Riccobene, E.: Modeling the Dynamics of UML State Machines. In: Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.) Abstract State Machines. LNCS, vol. 1912, pp. 223–241. Springer (2000)
3. Büttner, F., Gogolla, M.: Modular Embedding of the Object Constraint Language into a Programming Language. In: Simao, A., Morgan, C. (eds.) Proc. 14th Brazilian Symposium on Formal Methods (SBMF'2011). pp. 124–139. Springer, Berlin, LNCS 7021 (2011)
4. Drusinsky, D., tak Shing, M.: Using UML Statecharts with Knowledge Logic Guards. In: Schürr and Selic [30], pp. 586–590
5. Flake, S., Müller, W.: Formal semantics of static and temporal state-oriented OCL constraints. Software and System Modeling 2(3), 164–186 (2003)
6. Geiger, L., Zündorf, A.: Statechart Modeling with Fujaba. Electr. Notes Theor. Comput. Sci. 127(1), 37–49 (2005)
7. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Journal on Software and System Modeling 4(4), 386–398 (2005)
8. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69, 27–34 (2007)
9. Gutiérrez, M.E.B., Barrio-Solórzano, M., Quintero, C.E.C., de la Fuente, P.: UML Automatic Verification Tool with Formal Methods. Electr. Notes Theor. Comput. Sci. 127(4), 3–16 (2005)
10. Hamann, L., Gogolla, M., Kuhlmann, M.: OCL-Based Runtime Monitoring of JVM Hosted Applications. In: Cabot, J., Clariso, R., Gogolla, M., Wolff, B. (eds.) Proc. Workshop OCL and Textual Modelling (OCL'2011). ECEASST, Electronic Communications, journal.ub.tu-berlin.de/eceasst/issue/view/56 (2011)
11. Hamann, L., Hofrichter, O., Gogolla, M.: OCL-Based Runtime Monitoring of Applications with Protocol State Machines. In: Vallecillo, A., Tolvanen, J.P., Kindler, E., Strrle, H., Kolovos, D. (eds.) Modelling Foundations and Applications. LNCS, vol. 7349, pp. 384–399. Springer Berlin / Heidelberg (2012)
12. Hamann, L., Vidács, L., Gogolla, M., Kuhlmann, M.: Abstract Runtime Monitoring with USE. In: Ferenc, R., Mens, T., Cleve, A. (eds.) Proc. CSMR'2012 (2012)
13. Harel, D., Kugler, H.: The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML) - Preliminary Version. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) SoftSpez Final Report. LNCS, vol. 3147, pp. 325–354. Springer (2004)
14. Hennicker, R., Knapp, A., Baumeister, H.: Semantics of OCL Operation Specifications. Electr. Notes Theor. Comput. Sci. 102, 111–132 (2004)
15. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
16. Kleppe, A., Warmer, J.: Extending OCL to include Actions. In: Evans, A., Kent, S., Selic, B. (eds.) UML. LNCS, vol. 1939, pp. 440–450. Springer (2000)
17. Kleppe, A., Warmer, J.: The Semantics of the OCL Action Clause. In: Clark, T., Warmer, J. (eds.) Object Modeling with the OCL. LNCS, vol. 2263, pp. 213–227. Springer (2002)

18. Kolovos, D., Rose, L., Paige, R.: The Epsilon Book. Internet, http://www.eclipse.org/epsilon/doc/book/
19. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: Bishop, J., Vallecillo, A. (eds.) Proc. 49th Int. Conf. Objects, Models, Components, and Patterns (TOOLS'2011). pp. 289–305. Springer, Berlin, LNCS 6705 (2011)
20. Lano, K., Clark, D.: Semantics and Refinement of Behavior State Machines. In: Cordeiro, J., Filipe, J. (eds.) ICEIS (3-1). pp. 42–49 (2008)
21. Lano, K., Clark, D.: Axiomatic Semantics of State Machines, pp. 179–203. John Wiley & Sons, Inc. (2009)
22. Lano, K., Kolahdouz-Rahimi, S.: UML RSDS Model Transformation and Model-Driven Development Tools. Internet, http://www.dcs.kcl.ac.uk/staff/kcl/uml2web/
23. Mellor, S.J., Balcer, M.: Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley (2002)
24. Moffett, Y., Beaulieu, A., Dingel, J.: Verifying UML-RT Protocol Conformance Using Model Checking. In: Whittle, J., Clark, T., Kühne, T. (eds.) MoDELS. LNCS, vol. 6981, pp. 410–424. Springer (2011)
25. Ng, P.: A Concept Lattice Approach for Requirements Validation with UML State Machine Model. In: SERA. pp. 393–400. IEEE Computer Society (2007)
26. OMG (ed.): UML Superstructure 2.4.1. Object Management Group (OMG) (Aug 2011), http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF
27. OMG (ed.): Object Constraint Language 2.3.1. Object Management Group (OMG) (Jan 2012), http://www.omg.org/spec/OCL/2.3.1/
28. Porres, I., Rauf, I.: From Nondeterministic UML Protocol Statemachines to Class Contracts. In: ICST. pp. 107–116. IEEE Computer Society (2010)
29. Roldán, M., Durán, F.: Dynamic Validation of OCL Constraints with mOdCL. ECEASST 44 (2011)
30. Schürr, A., Selic, B. (eds.): Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings, LNCS, vol. 5795. Springer (2009)
31. Seifert, D.: Conformance Testing Based on UML State Machines. In: Liu, S., Maibaum, T.S.E., Araki, K. (eds.) ICFEM. LNCS, vol. 5256, pp. 45–65. Springer (2008)
32. Shen, W., Compton, K.J., Huggins, J.: A UML Validation Toolset Based on Abstract State Machines. In: ASE. pp. 315–318. IEEE Computer Society (2001)
33. Shlaer, S., Mellor, S.J.: Object Lifecycles: Modeling the World in States. Yourdon Press, EngleWood Cliffs, NJ (1992)
34. Shlaer, S., Mellor, S.J.: Object-Oriented Systems Analysis: Modelling the World in Data. Yourdon Press, EngleWood Cliffs, NJ (1992)
35. Wang, Y., Zhang, Z., Yao, D.D., Qu, B., Guo, L.: Inferring Protocol State Machine from Network Traces: A Probabilistic Approach. In: Lopez, J., Tsudik, G. (eds.) ACNS. LNCS, vol. 6715, pp. 1–18 (2011)
36. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Object Technology Series, Addison-Wesley, Reading/MA (2003)
37. Weißleder, S.: Influencing Factors in Model-Based Testing with UML State Machines: Report on an Industrial Cooperation. In: Schürr and Selic [30], pp. 211–225
38. Yue, T., Ali, S., Briand, L.C.: Automated Transition from Use Cases to UML State Machines to Support State-Based Testing. In: France, R.B., Küster, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA. LNCS, vol. 6698, pp. 115–131. Springer (2011)

Publication A24C

Abstract Runtime Monitoring with USE

Authors: *Lars Hamann, Laszlo Vidacs, Martin Gogolla, and Mirco Kuhlmann*

Proc. European Conference Software Maintenance and Reengineering (CSMR'2012)

The final publication is available at IEEE via
`http://dx.doi.org/10.1109/CSMR.2012.73`

Abstract Runtime Monitoring with USE

Lars Hamann
University of Bremen
Germany
lhamann@informatik.uni-bremen.de

László Vidács
University of Szeged
Hungary
lac@inf.u-szeged.hu

Martin Gogolla
University of Bremen
Germany
gogolla@informatik.uni-bremen.de

Mirco Kuhlmann
University of Bremen
Germany
mk@informatik.uni-bremen.de

***Abstract*—We present a tool that permits developers to monitor and verify assumptions at an abstract level about an application running on a virtual machine. On the implementation level, a so-called platform aligned model (PAM) described in the UML (Unified Modeling Language) and enriched by OCL (Object Constraint Language) requirements is used to formalize these assumptions. Our solution allows a developer to concentrate on verifying core parts of an implementation while ignoring major parts of peripheral technical details. In order to easily detect a PAM which characterizes the central requirements, we propose a semi-automatic approach. First, a complete program model is generated by analyzing the source code. Afterwards, this model is reduced by the user to central classes and associations. This reduced model is enriched by the assumptions about the expected behavior of the system. The monitor connects to the running system at a particular point in time and builds up an abstract snapshot, i.e., an instance of the PAM, which corresponds to the current state. When the application is further executed this snapshot is synchronized by listening to changes in the running system. During monitoring the stated assumptions are validated and possible violations are reported to the user.**

I. Introduction

Formal models are playing an important role in several areas of software engineering. For example in model checking a formal model of a system has to be specified to verify properties of a system. This formal model needs to be a valid abstraction of the concrete system to be able to exclude errors which are introduced by the modeling task. In this paper we present a tool-chain which is able to simulate an abstract model by executing and monitoring its related implementation. The model is automatically built from the source code of an implementation. To be able to focus on central parts of the application we present filtering techniques to reduce the overall size of the system model. This reduced model is enriched by the user with system assumptions which are formulated as OCL [1] invariants, pre- and postconditions. The last model is called a platform aligned model (PAM). If the monitor detects a violation of a formulated assumption at runtime the user is informed. She can now explore the monitored system in an abstract way to identify the cause of the violation.

Several approaches on runtime verification exist. A detailed comparison of runtime verification approaches using OCL can be found in [2]. None of them uses the events provided by a virtual machine to react on changes in the monitored system. A runtime monitoring approach using other formal languages is for example JavaMOP [3].

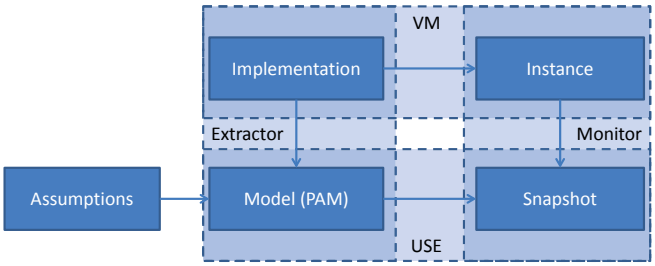


Fig. 1. Tools and artifacts

II. Monitoring approach in USE

The tools and artifacts used in our approach are shown in Fig. 1. The artifacts are shown as solid rectangles. The dashed rectangles are the tools and cover the artifacts which are required by them.

A central part of our monitoring process is the UML and OCL tool USE [4] placed at the bottom of the figure. It uses a model and an instance of such a model to validate constraints included in the model. In the context of our monitoring approach the model instance is called a snapshot. In the upper part of the figure a virtual machine (VM) is shown, which executes an implementation. The runtime data of the application (heap space, stack frames, etc.) is labeled as the 'instance' of the implementation. The required parts of an instance are read by the monitor and transformed into a snapshot, i. e., an instance of the model provided to USE. This model is generated by examining the source code of the implementation and reducing it to central aspects of the system. The central aspects depend on the assumptions a user wants to check. Therefore, the generation task includes several steps. Some of them can be done automatically while others need user interaction. The result of this extraction process is a model which concentrates on properties of the system by ignoring irrelevant parts. After the extraction step the model is enriched with assumptions that should be checked at runtime.

A. USE (UML-Based Specification Environment)

The main task for USE originally was to support the design of systems in an early stage of development. A developer can specify a model of a system with a subset of UML and extend is with constraints formulated in OCL (Object Constraint Language, c. f., [1]). The formulated constraints can be validated by creating system states also called object

diagrams and examining the evaluation result of the constraints against the specified system states. These system states can be built manually as scenarios to validate if the specified model behaves as expected. This manual checking is similar to unit tests on the source level. Further, formal verifications can be done to a certain degree by using a built-in system state generator [5]. The most simple verification is to check if an instance of a model exists, i. e., if the model is consistent. To verify this, a user can search within predefined bounds for a valid system state. If such a state is found, the model is consistent which means no constraint contradiction occurs.

B. USE Monitor

The runtime monitor in USE [6] is realized as a plugin. It currently supports the monitoring of applications running inside a Java virtual machine (JVM). The monitor requires a so-called platform aligned model (PAM) which specifies central aspects of a system to monitor. It is called platform aligned, because information about the implementation is needed within the model, e. g., package names or attribute names for association ends. The monitor can be attached to a running system at any time. After it is connected it takes a snapshot of the running system and maps instances inside the virtual machine to instances of classes of the PAM. The snapshot only contains instances of modeled classes, attributes and associations. Therefore, a snapshot can be seen as a subset of the central data of the running system. After this initial snapshot has been taken a user can examine static aspects of the system by checking structural constraints, e. g., specified multiplicities or invariants. Dynamic validation can be done by resuming the system which is monitored. After the application has been resumed by the monitor, the monitor reacts on several events coming from the virtual machine to keep track of changes and to be able to synchronize the snapshot with the running instance. When monitored operations (operations specified in the PAM) are called inside the running system, the monitor pauses the running system and validates the specified preconditions for the operation. If a precondition fails the user is notified and she can react on this violation by examining the failed precondition and the current system state. Analogously to normal testing, she has to decide whether the specified constraint is erroneous or a feature of the implementation. If no violation is encountered the execution is continued until the operation is returning or other monitored operations are called. When the operation is returning, the monitor pauses the execution again and checks the specified postconditions of the operation. At this point, a central benefit of reusing USE as an execution environment for models gets visible. USE records the system state before an operation was called which allows the validation of postconditions including the usage of the OCL-keyword @pre. In our previously published work [6] the monitor was controlled by simple shell commands inside of the USE shell. The approach was extended to a more intuitive user interface (see Fig. 4). This also allows finer controlled messages to the user and an elegant way to integrate model breakpoints into the monitor in the future.

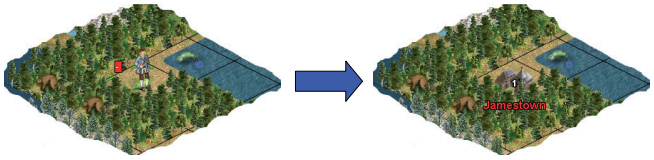
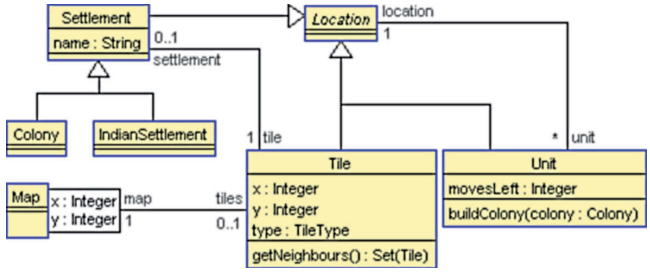


Fig. 2. Parts of a running FreeCol game




```
context Unit::buildColony(colony:Colony)
  pre movesLeft: self.movesLeft > 0
  pre noSurroundingColonies:
    self.location.oclIsKindOf(Tile)and
    self.location.oclAsType(Tile).
      getNeighbours()->forAll(t |
        t.settlement.isUndefined())
```

Fig. 3. PAM for FreeColonization and two assumptions

C. Sample monitoring process

To be able to compare the results of the semi-automatic PAM extraction process to a hand written PAM we reuse the example shown in [6]. In the example we build a PAM for the open source computer game *Free Colonization*. We concentrate on monitoring the execution of one central functionality of the game: the founding of colonies. The example game situation is shown in Fig. 2. The left part is the state before the founding of a colony, whereas the right part shows the state after founding the colony Jamestown. A PAM for the game with assumptions about the behavior formulated as pre- and postconditions is shown in Fig. 3. In addition to the pre- and postconditions introduced to the model, also a query operation `getNeighbours()` was added to the PAM to be able to reuse this expression. The presented preconditions in Fig. 3 state, that an operation call to `buildColony` is only valid, if the unit has moves left and there are no surrounding colonies. The screenshot of the USE system presented in Fig. 4 shows the result of attaching the monitor to FreeCol when the game is in the left state of Fig. 2 and monitoring the execution of the founding of Jamestown which results in the right state of Fig. 2. Parts of this state are shown as an object diagram in USE on the right upper side of the screenshot. The execution flow is shown in the center of the screenshot. Please note, we used a more detailed PAM including additional operations for the screenshot to make it more meaningful but we show only a fragment in Fig. 3.

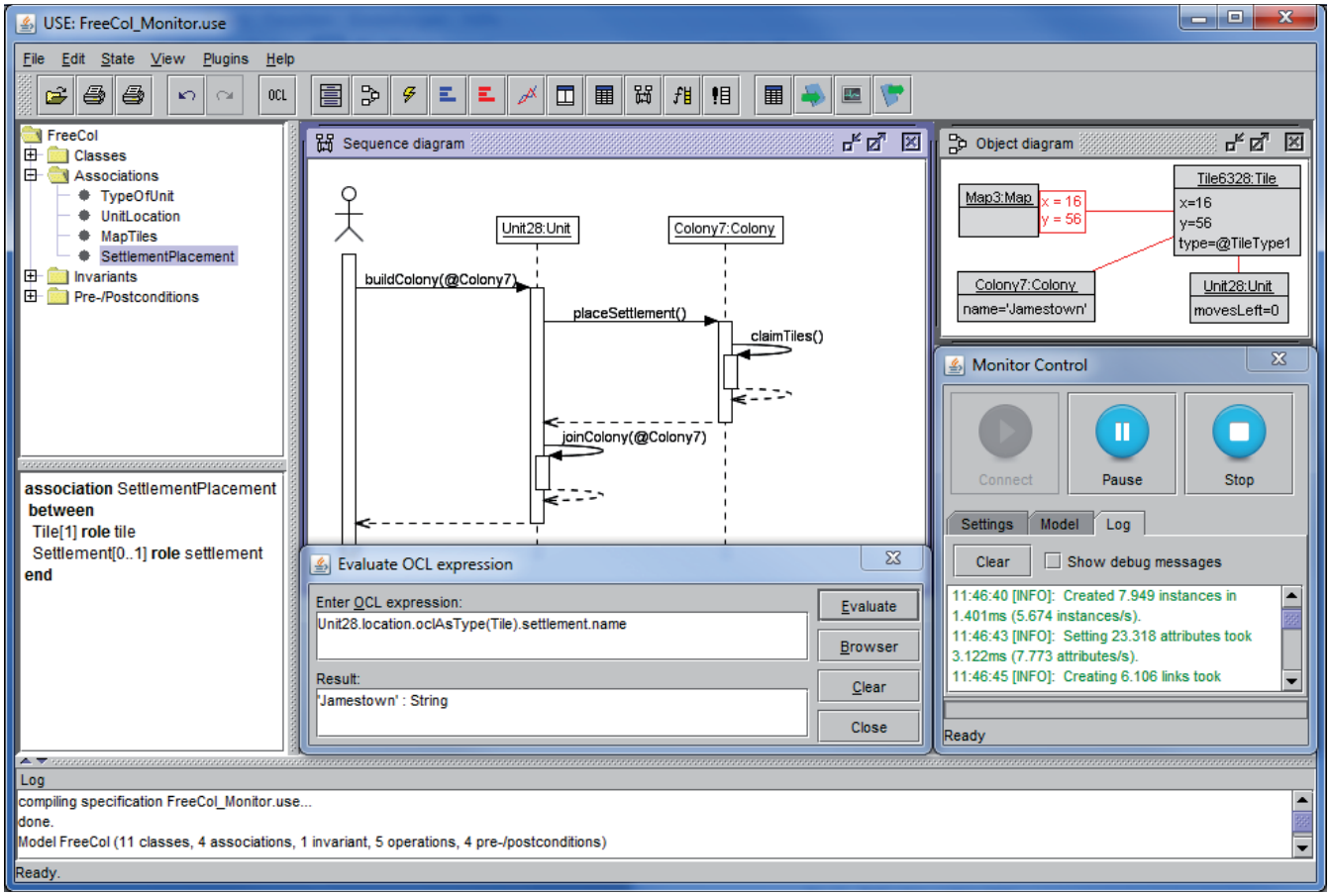


Fig. 4. System state presented in USE while monitoring

D. Support for PAM extraction

In our previous work the platform aligned model was created manually by exploring the source code as shown in Fig. 3. Although finding the appropriate part of a large program remains a human task, it can be fairly supported by reverse engineering techniques. The main problem with an automatically extracted model is its size. Coping with huge models at runtime is challenging: they cause severe performance issues for modeling tools; and they hinder the visual observation and program understanding of the developer.

We approximate the PAM model by a reverse engineered class diagram. We extract the PAM of the whole application from the source code and export it as a USE model. The essential, important part of the model is achieved using further filtering. Thus we employ filtering at two levels:

- pre-filtering - unnecessary details are filtered out during the model building phase
- USE filtering - search/select the important part of the model in USE

Figure 5 contains the overview of the tools used for automatic extraction of PAM. Static analysis of source code is done by the Columbus Java analyzer [7]. The obtained program model is converted to a higher level, language independent object-oriented model. The existing Columbus tool-chain is extended with a new module, which computes the PAM and exports it to a USE model.

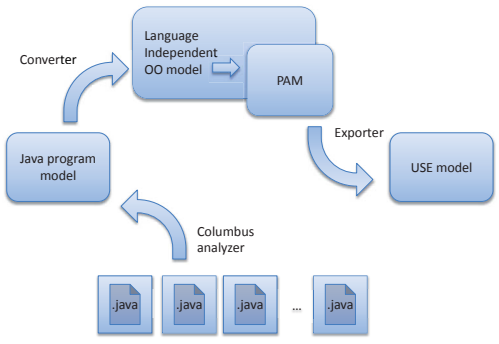


Fig. 5. Tool architecture of the PAM Extractor

During PAM extraction we obtain facts from the source code and convert them to a valid USE model. First, a base class diagram is built consisting of classes, attributes, methods, inheritance relationships and associations. Associations are extracted as suggested by Kollmann et al. [8]. The final model has to conform to several rules like source code traceability,

concise and consistent naming of elements and unique navigability of associations. Source code elements at some points break the well-formedness rules of USE models, e.g. when an attribute is defined both in a base and in a descendant class. To overcome these problems, the names in the model are changed at several points - shortened or made unique by appending unique identifiers to names. The source code traceability of modified names is assured by name annotations in the USE model. Pre-filtering currently consists of dropping out attributes taking part in associations and filtering out Java library classes and their references (attributes, methods with library class parameters).

We validated our reverse engineering solution by extracting the model of the FreeCol program. The extracted USE model was filtered to be comparable to the model made previously by hand. In the USE system there are several possibilities to search classes and their neighbours; and to crop and hide appropriate classes to get a reduced model showing essential part of the application. A typical filtering step can be seen in Fig. 6: the immediate neighbours of selected class *Tile* are shown, while others are hidden.

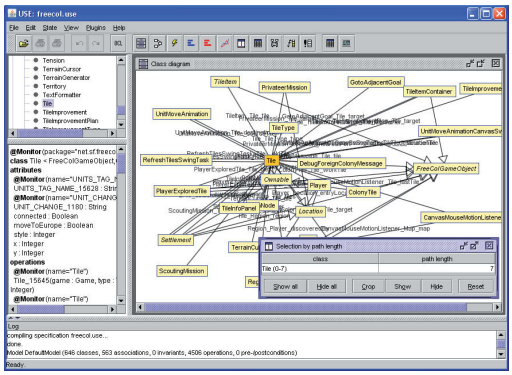


Fig. 6. Examining neighbours of class *Tile*

Figure 7 shows the automatically extracted model after filtering in USE. The main difference compared to Fig. 3 lies in the discovered associations. *Settlement* and *Tile* are associated in both directions, but there is also an additional association from the direction of *Tile* pointing to the owning settlement. Similar observations can be made with *Unit* and *Location* as well. Furthermore, *type : TileType* is an attribute of class *Tile* in the manual model, while it is generated as an association according to the rules of automatic model extraction.

Finally, using the generated model we successfully reproduced the same condition checking procedure as done previously on manually the created models.

III. Conclusions

We have presented a tool-chain that allows developers to monitor a Java application in form of a platform aligned model (PAM) enriched by OCL requirements. With an example we have shown that the tool-chain is capable of handling non-trivial applications with several hundred classes. As future work, larger case studies have to be carried out. In order to find key classes and to support the PAM discovery, concept location techniques could be applied. Furthermore, we think of (what we would call) 'model breakpoints' which permit a developer to force the application to pause at a certain point in the model, not on a specified line in the code. Model breakpoints may be employed in connection with particular conditions. Another line of research would be to incorporate traces in the approach so that certain operation call sequences can be monitored. Last but not least we could check to what extent the approach is applicable to other virtual machines like CLR (Common Language Runtime) for .NET languages.

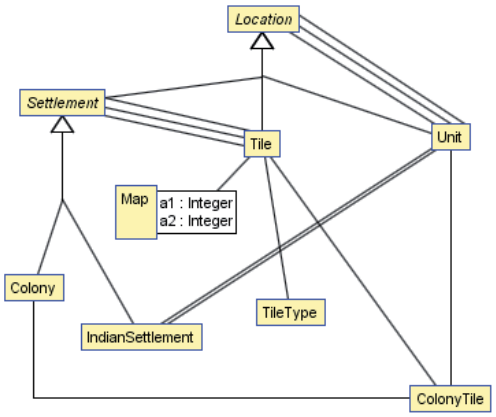


Fig. 7. Extracted USE model of the Freecol example

References

[1] *Object Constraint Language Specification Version 2.2*, OMG - Object Management Group, Feb. 2010. [Online]. Available: http://www.omg.org/spec/OCL/2.2

[2] C. Avila, A. Sarcar, Y. Cheon, and C. Yeep, "Runtime Constraint Checking Approaches for OCL, A Critical Comparison," in *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010)*. Knowledge Systems Institute Graduate School, 2010, pp. 393–398.

[3] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Ros, u, "An Overview of the MOP Runtime Verification Framework," *International Journal on Software Techniques for Technology Transfer*, 2011.

[4] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-Based Specification Environment for Validating UML and OCL," *Science of Computer Programming*, vol. 69, pp. 27–34, 2007.

[5] M. Gogolla, M. Kuhlmann, and L. Hamann, "Consistency, Independence and Consequences in UML and OCL Models," in *Proc. 3rd Int. Conf. Test and Proof (TAP'2009)*, C. Dubois, Ed. Springer, Berlin, LNCS 5668, 2009, pp. 90–104.

[6] L. Hamann, M. Gogolla, and M. Kuhlmann, "OCL-Based Runtime Monitoring of JVM Hosted Applications," in *Proc. Workshop OCL and Textual Modelling (OCL'2011)*, J. Cabot, R. Clariso, M. Gogolla, and B. Wolff, Eds., ECEASST. Electronic Communications, journal.ub.tu-berlin.de/eceasst/issue/view/56, 2011.

[7] L. Schrettner, L. J. Fülöp, R. Ferenc, and T. Gyimóthy, "Visualization of software architecture graphs of java systems: managing propagated low level dependencies," in *Proceedings of the 8th International Conference on Principles and Practice of Programming in Java, PPPJ 2010, Vienna, Austria*, 2010, pp. 148–157.

[8] R. Kollmann and M. Gogolla, "Application of the UML Associations and Their Adornments in Design Recovery," in *Proc. 8th Working Conference on Reverse Engineering (WCRE'2001)*, P. Aiken and E. Burd, Eds. IEEE, Los Alamitos, 2001.

Acknowledgment: The work of László Vidács was supported by the DAAD.