Rigorous Object Oriented Software Development

Draft

Version 0.3

5.3.2002

Dr. Albert Zündorf

Acknowledgements

This book is a simplified and reorganized version of my Habilitation thesis.

I still have to thank Prof. Wilhem Schäfer for supervising my work and Prof. Gregor Engels, Prof. Carlo Ghezzi and Prof. Manfred Nagl for reviewing my thesis and lots of constructive suggestions. I have to thank R. Depke, M. Gehrke, H. Giese, R. Heckel, J. Jahnke, U. Nickel, J. Niere, J. Wadsack, S. Schrödel for proof reading.

This work would not have been possible without the Fujaba developers (sorted by contributed LOC, descending): Michael Kisker, Lothar Wendehals, Carsten Reckord, Jens Mühlenhoff, Burkhard Ulrich, Robert Wagner, Matthias Tichy, Thomas Klein, Jörg Niere, Stefan Scharnke, Ingo Rockel, Marcus Pallasdies, Felix Wolf, Lars Torunski, Oliver Hager, Andreas Schill, Hans-Josef Köhler, Thorsten Fischer, Kan-Sing Wan, Leif Geiger, Thomas Maier, Nils Bandener, Christian Köpke, Andreas Elsner, Thomas Sander, Stefan Kisker, Kan-Hung Wan, Frank Heimes, Ulrich Nickel, Dominik Küster, Peter Skilinski, Ralf Schuhmacher, Charalampos Chrysikopoulos, Martin Kamneng, Michael Hurst, Lars Nyvik.

Thank you very much.

Table of Contents

1 Introduction

During the last two decades, the object oriented paradigm has improved the software development practice, massively, with respect to software quality, developers' productivity, and manageable project size and complexity. The first contribution is the concept of a class as an encapsulation of data fields and corresponding operations. This encapsulation allows the localization of certain system functionality. Different kinds of class relationships allow the combination of classes to system architectures. This enables the modeling of system structures at a higher level of abstraction. Object oriented component and connector architectures build up flexible systems, easily adaptable for changing requirements. Object oriented CORBA [Emm00] technology provides mechanisms for distributed applications.

As classes allow to organize complex system structures, objects allow to organize complex runtime information. Objects allow to localize certain pieces of information. Different kinds of object relations allow the combination of objects to complex object structures. There are many examples how objects and object relations allow to tackle complex functionality and facilitate the building of intelligent behavior. The design patterns proposed by Gamma et al. [GHJV95] intensively rely on well-organized object structures and the concept of delegation. Certain tasks are forwarded to neighbor objects and thus, changing the current object setting yields behavior changes. This allows to build flexible applications with relatively simple source code. Generally, object structures are able to represent arbitrary complex system states. Well known is the representation of the structure and current state of complex graphical user interfaces. The same holds in principal for all kinds of office applications like text processors, spread sheet calculators, (vector) graphics programs, WWW browsers, and CAxx tools. This is also true for runtime data structures of operating systems, compilers, database systems, or WWW servers.

Other applications with complex state information are e.g. workflow management and groupware systems. Even embedded systems are now moving to a more decentralized organization that employs more intelligent components. A modern car employs several dozen controller nodes, e.g. for operating the windows, the seat, the control devices, the radio, the breaks, and last but not least the car engine. All these elements are connected to implement a cooperative behavior. For example, if a person starts the car using his/her personal car key, seat and mirrors may automatically move to the positions preferred by that person and the radio may recall the preferred radio station.

Another example for a complex embedded system with multiple distributed control nodes (control processes) is a modern production hall with autonomous production cells and with an intelligent, autonomous transportation system. All these applications employ complex object structures to achieve an intelligent behavior and to be able to represent some kind of knowledge about their current state.

However, with state-of-the-art programming languages and techniques, the realization of complex object structures is a tough job. Object structures are build by objects and pointers. Pointers are frequently referred to as the goto of data structures. Object structures are built and changed through creation and removal of objects and through redirecting pointers by assignment statements. With this primitive means, pointers tend to become corrupted. If the object structure reaches a certain complexity, problems like memory leaks, dangling references, and corrupted system states emerge. With current techniques, the developer always fights with a jungle of references and for example the removal of an object may become a real challenge.

Pointer and memory management in C and C++ is a real nightmare. Modern languages like Java have improved the situation a lot, but although the garbage collector facilitates the memory management considerably, it is still a tough job to isolate a certain object within a complex object structure in order

to achieve its removal. Therefore, many developers tend to avoid complex object structures. In safety critical areas and for most embedded systems the creation of objects at runtime or more generally the usage of a memory heap is considered as an unacceptable risk and usually forbidden by the project guidelines. Thus, current programming languages and object oriented techniques are not yet sufficient for building reliable, complex, object oriented applications. However, without advanced object oriented concepts, intelligent system behavior based on complex knowledge cannot be realized.

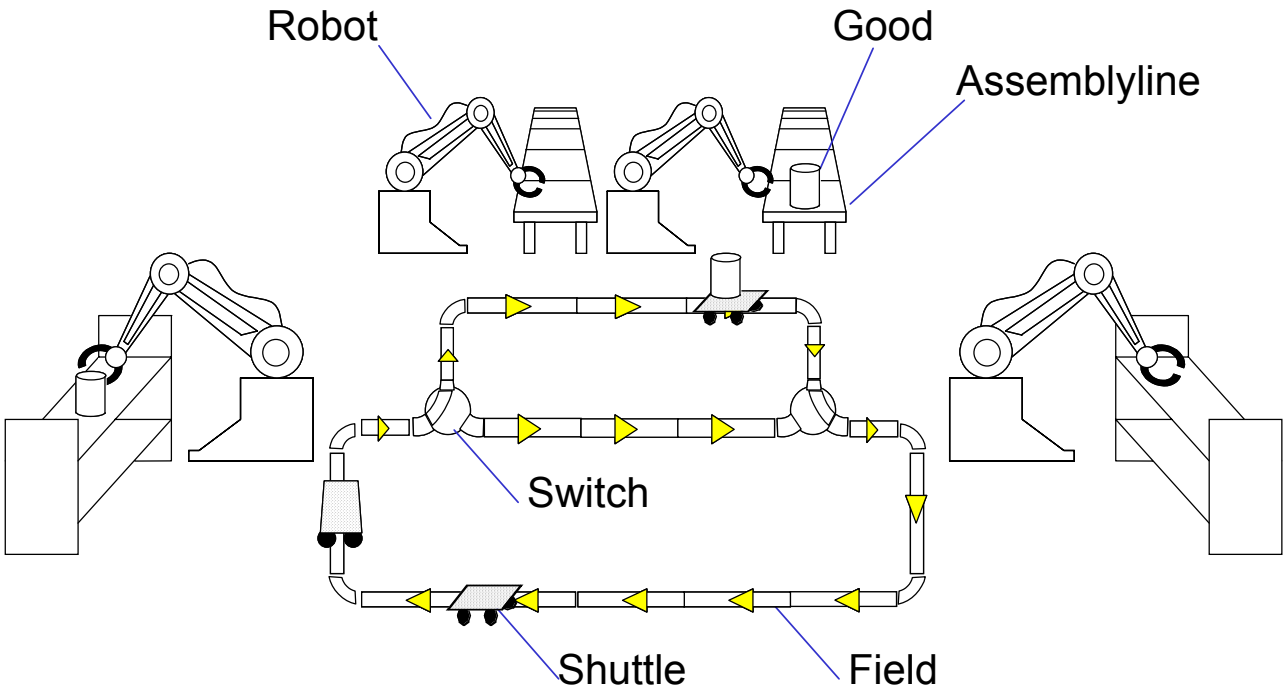
Several modern object oriented modeling methods and languages like Object Oriented Analysis (OOA) and Object Oriented Design (OOD) from Booch [Boo95] and Object Oriented Modeling Technique (OMT) from Rumbaugh et al. [RBPEL91] try to overcome this dilemma by providing a higher level of abstraction for the software development. Today, the Unified Modeling Language (UML) [BRJ99] and the Unified Process [JBR99] are state-of-the-art. These modern software development methods try to provide a systematic approach to the development of reliable object oriented applications and complex object structures. Above modeling languages try to support the analysis, design, and implementation of applications employing complex object structures on a higher level of abstraction. Modern computer aided software engineering (CASE) tools like Rational Rose [RR-RT], Rhapsody [Rhap], or TogetherJ [Toge] have been developed, that try to support the editing of high level modeling languages, that try to support an explicit software development process, that try to provide consistency analysis functionality within each software development phase and across different phases through the whole system model. The forward engineering process shall be supported by generators that try to support the transition from analysis to design and from design to implementation. Tools with reverse engineering functionality try to retrieve design and analysis information from the implementation. Round-Trip functionality try to ensure that changes to either analysis documents, design, or implementation are automatically propagated to predecessor and successor phases.

Unfortunately, current software development methods, current modeling languages, and current CASE tools are not yet mature enough to enable sophisticated development of reliable complex object oriented applications. State-of-the-art are the Unified Process [JBR99] and the unified modeling language UML [BRJ99, RJB99, UML97]. The Unified Process provides general guidelines for the organization of software development projects using an iterative life cycle. The Unified Process organizes software projects in different phases and into workflows. The phases correspond to a time axis, from early project phases via main development phases to maintenance and growing phases. The workflows correspond to different kinds of tasks which are executed for each extension / iteration of the project. This includes requirement engineering tasks, analysis tasks, design tasks, and implementation tasks. Each workflow involves certain kinds of team members like project managers, requirement engineers, analysts, designers, component engineers, software developers, and customers. Each of these roles contributes specific expertise and responsibilities to each of the workflows. Each task employs and contributes to a number of different documents like requirement documents, analysis documents, project plans, design documents, implementation documents, and test reports. Actually, the Unified Process helps a lot in organizing a software project. However, it focuses on project organization. The Unified Process provides only little help for the actual design and modeling tasks. There is no help how analysis scenarios should be created, how class diagrams may be derived from these scenarios, how reactive behavior should be modeled that realize the analysis scenarios, how system behavior and reliable complex object structures are to be specified and how such specifications may be turned into a robust, reliable, and maintainable implementation. A more technical, detailed, and content oriented development method is necessary.

The UML is a general object oriented modeling language that provides different sublanguages for the different phases and tasks of an object oriented software project. This work assumes that the reader is already familiar with the UML. However, due to the importance of the UML in general and for this work, we shortly revisit the common usage of the UML.

As an application example for our short trip through the UML, we use a manufacturing hall employing a decentral autonomous transportation system, cf. Example 1.1. This example is part of the ISILEIT project funded by the German Research Foundation (DFG) as part of a nation wide research program for the integration of software specification methods in engineering disciplines (in German: Schwerpunkt-programm zur Integration von Techniken der Softwarespezifikation in ingenieurwissenschaftlichen Anwendungen). The manufacturing site hosts a number of assembly robots and storages. The transportation system employs autonomous shuttles that travel on a track system. The shuttles receive transportation and production tasks from a central planning unit. Each shuttle executes its transportation and production tasks, autonomously. The shuttle employs knowledge of the factory and track layout and of the position and capabilities of the employed construction robots. Each shuttle negotiates with the different assembly robots and analyses the work load of different system parts in order to compute an optimal travel and production plan. The shuttles have to deal with traffic jams, blocked tracks, and non-functioning assembly robots. Therefore, the shuttle employs a complex object structure representing the shuttles' "world". This object structure serves as basis for planning and executing the shuttles' tasks.

Example 1.1: The ISILEIT case study, an autonomous transportation system

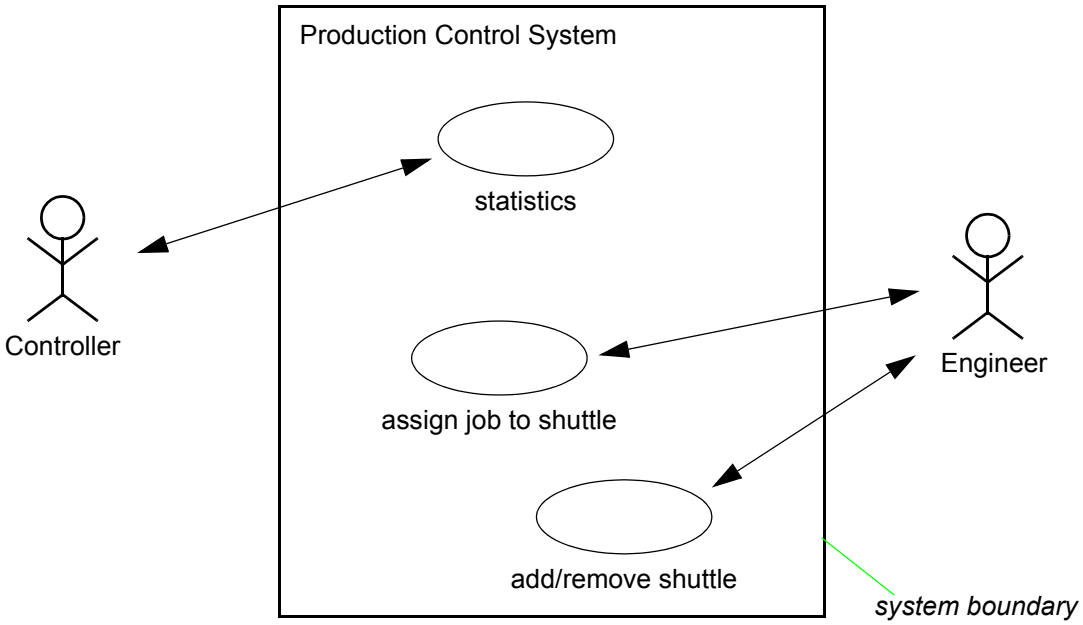


Our responsibility within the ISILEIT project was the development of methods that allow the realization of the shuttles' control software in a reliable way. Note, this kind of embedded systems has very high (software) quality requirements. The system is expected to work 24 hours a day throughout the year. Downtimes due to software failures or regular reboots in order to deal with memory leaks are not acceptable. On the other hand, a flexible and "intelligent" behavior is desired that allows to produce different kinds of products in small lot sizes in parallel. The latter requirement implies the need for complex object structures representing the shuttles knowledge and planning data.

Employing the UML, we would probably start to develop our application using Use-Case diagrams to enumerate the requirements, cf. Example 1.2. Basically, a Use-Case diagram consists of a number of stickmen and ovals. The stickmen represent different kinds of users or user roles. The ovals represent functionalities of the planned software system. Arrows represent which users may access which system functionalities. The ovals are enclosed by a large rectangle separating them from the stickmen. This rectangle represents the system border. Everything outside the rectangle does not belong to the

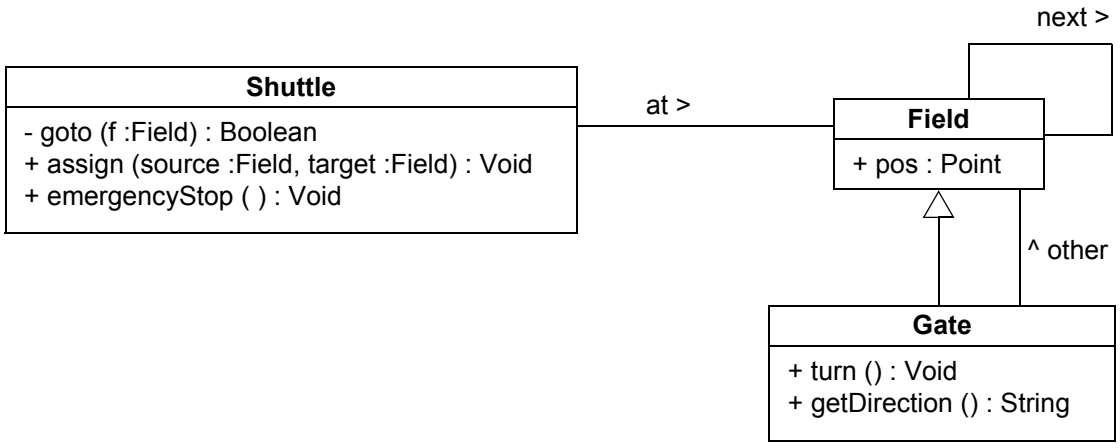
planned software system and needs not to be implemented. Use-case diagrams are a valuable means for requirements analysis. They facilitate the communication with the customer and the elicitation of functional requirements and their separation from surrounding system components.

Example 1.2: Use-case diagrams



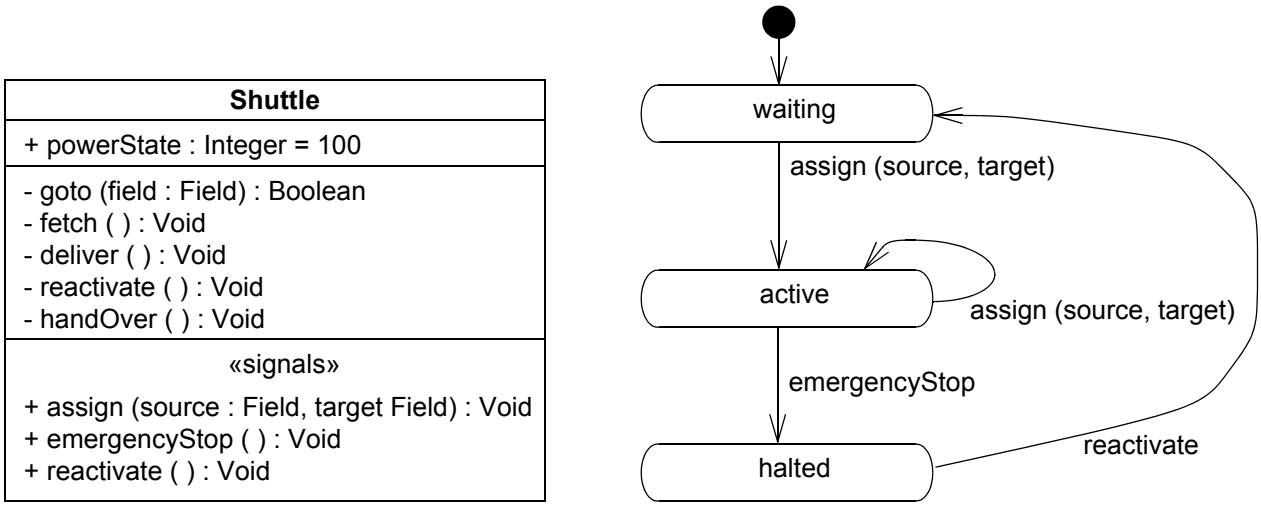
Usually, use-cases are accompanied with additional descriptions in natural language. In order to derive a domain level class diagram, common object oriented methods recommend to analyse these descriptions and to mark all substantives. "Relevant" substantives are turned into classes. Then one analyses verbs and simple phrases in order to "derive" attributes, methods, and associations. The result is a domain level class diagram as shown in Example 1.3.

Example 1.3: Cutout of a domain level class diagram



During the object oriented analysis phase, use-cases are revisited and refined using different kinds of behavior and scenario diagrams. In our transportation system, we employ different kinds of autonomous, active objects. In the UML, the reactive behavior of active objects may be modeled with UML statecharts, cf. Example 1.4.

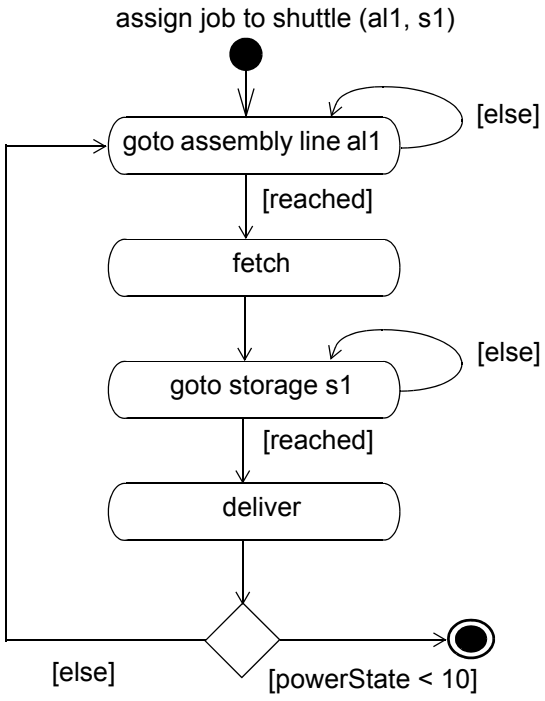
Example 1.4: Modeling active objects with statecharts



In Example 1.4, the methods of class Shuttle have been grouped into usual methods and signals. The reaction of shuttle objects to received signals is defined by the given statechart. Initially, a shuttle is in state waiting. On the receiption of an assign event the shuttle switches to state active. While in state active, new assign signals may be received. On an emergencyStop signal, the shuttle switches to state halted. In state halted, the shuttle does not respond to signals other than reactivate. A reactivate signal switches the shuttle back into state waiting, where it is able to react on a new assign signal.

Alternatively, a use-case or a complex action may be refined by an activity diagram, cf. Example 1.5. An activity diagram consists of bonbon shaped boxes representing activities that are connected by control flow arcs. Activity boxes contain short descriptions in natural language or pseudo code. A filled circle marks the start activity, a bulls-eye the terminal activity. Diamond shaped activities may be used to emphasize branching control flow. The control flow arcs may be labeled with guard conditions in square brackets.

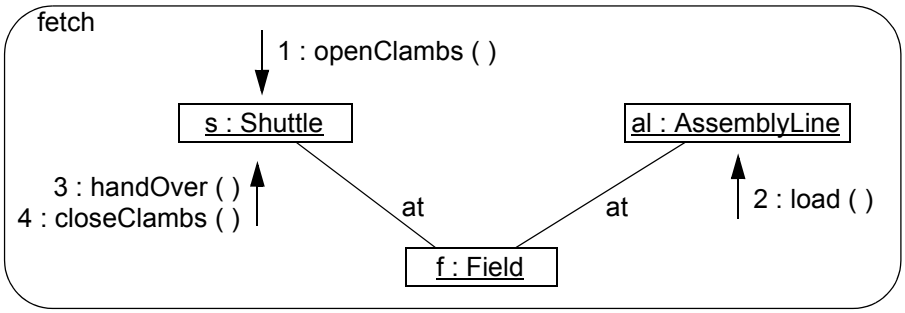
Example 1.5: Refinement of use-cases with activity diagrams



The activity diagram of Example 1.5 refines the use-case "assign job to shuttle". If this message is send to a shuttle, the shuttle first travels to the assembly line a1, passed as first parameter of the assign message. If the assembly line is reached, the shuttle fetches the good that shall be transported. Otherwise, the goto step is repeated. After fetching the transport good, the shuttle goes towards the storage s1 passed as second parameter to the assign message. When the shuttle reaches the storage, it delivers the transported good. Then a branch is reached. If the battery power of the shuttle has fallen below a given threshold, the execution terminates. Otherwise, a new transportation cycle starts.

Alternatively, complex actions may be modeled using collaboration diagrams. A collaboration shows a possible cut-out of the runtime object structure and a possible message flow between the collaborating objects. Example 1.6 shows a refinement of activity fetch employed in Example 1.4. Three objects are employed, a shuttle s, a field f, and an assembly line al. In step 1, an openClambs message is sent to the shuttle in order to prepare the shuttle for being loaded. In step 2, a load message sent to the assembly line causes the loading of a good onto the shuttle. After the loading, in step 3 the shuttle receives a handOver message that transfers the responsibility for the loaded good from the assembly line to the shuttle. Finally, a closeClambs message to the shuttle fixates the transport good and the shuttle is ready to go on to the target storage, cf. step 4.

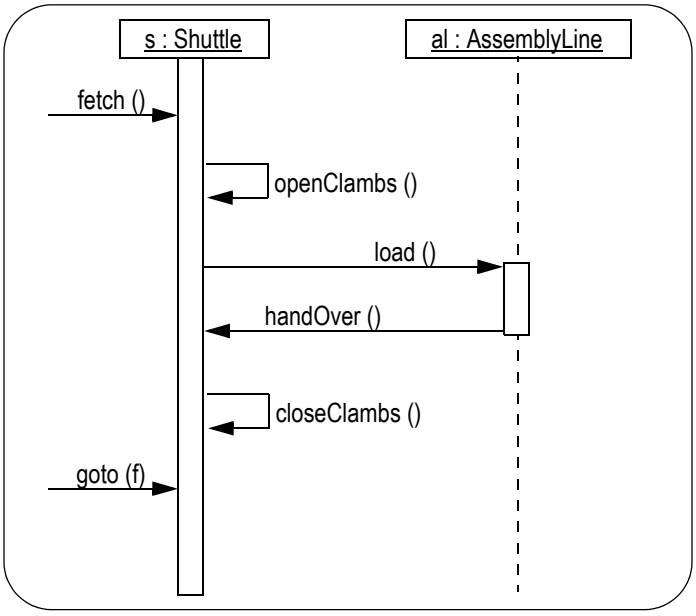
Example 1.6: Modeling scenarios with collaboration diagrams



An UML collaboration diagram may always be turned into a semantically equivalent UML sequence diagram (and vice versa), cf. Example 1.7. A collaboration diagram emphasizes an object structure centered view on a collaboration. The cutout of the object structure that is of interest is depicted together with all import object relations and possibly with attribute values. On the other hand, the timely order of messages is shown with sequence numbers, only. If the message flow becomes a little bit complex, collaboration diagrams easily become crowded and hard to read.

In contrast, sequence diagrams emphasize the timely order of messages but neglect object relationships and attribute values. In a sequence diagram, the collaborating objects are shown on the top row of the diagram. Under each object a vertical life line is depicted. The vertical axis corresponds to a time axis, time floating from top to bottom. Horizontal arcs represent message exchanges at certain points in time. While an object is executing some method, its life line is shown as a bar. While the object is passive, its life line is just a dashed line. Example 1.7 shows exactly the same activity refinement as Example 1.6 but from a time centered perspective. The shuttle receives a fetch message. It sends itself an openClambs message and then it sends a load message to the assembly line. The assembly line responds with a handOver message. Next, the shuttle sends itself a closeClambs message and then it receives a goto message.

Example 1.7: Modeling scenarios with sequence diagrams



This was a very short introduction to the UML. However, this kind of introduction corresponds to what one may expect from a typical UML course or what one will find in a typical UML book. Of course, the different diagram kinds are presented in more detail and much more language elements are discussed. In addition, our introduction skipped a number of UML diagrams. Nevertheless, the UML is typically just presented as an assembly of loosely coupled diagrams. For the different diagrams, only the syntactical elements are presented, a sound formal semantics is not provided. In addition, a single aspect of the modeled system may be described in different ways using different kinds of diagrams. Only little help is provided, how a certain aspect should be modeled using which kind of diagram in which way. As shown for collaboration and sequence diagrams, different diagrams of one UML specification may overlap, i.e. they may model similar aspects. However, such a connection becomes not clear from the language. In general, there exists no or only little formal relationships between the different diagrams modeling a system. Accordingly, in the UML no notion of completeness and consistency exists. It is never clear, whether all use-cases and all complex activities have been refined somewhere. Similarly, for two given diagrams it is not clear whether they model completely separate, overlapping, or exactly the same aspect. Even worse, if two diagrams actually model overlapping aspects, there is no formal possibility to check, whether the different descriptions are compatible and consistent. While this is acceptable for the analysis phase, for the design and implementation phase a complete and consistent and formal system specification covering the general case is required. The UML is not appropriate for system specification since most of the UML behavior diagrams model only example scenarios and incomplete, possible behavior. They do not provide a complete behavior specification for the general case. Finally, the semantics of UML behavior diagrams is not yet formally defined. Thus, even if a complete and consistent system specification could be created, it would not be clear what it means and how it should be implemented. Thus, the UML is not yet usable for the reliable specification of complex object oriented applications.

As current object oriented software development methods like the Unified Process lack "how-to" support for modeling activities and as current modeling languages like the UML lack (formal) semantics and a sufficient notion of completeness and consistency, current CASE tools like Rational Rose, Rhapsody, and TogetherJ provide only limited support for the development of reliable complex object oriented applications. Only for class diagrams and statecharts consistency analysis and code generation are reasonably supported. In addition, TogetherJ supports sequence diagrams. Only, TogetherJ realizes reliable reverse engineering functionality.

Altogether, neither current object oriented software development methods nor current object oriented modeling languages nor current CASE tools are mature enough for the development of high quality, reliable, (safety critical,) complex, object oriented applications.

However, there are large market demands for such complex applications that meet high quality standards allowing their usage for safety critical areas or for embedded systems. Such applications need to become more intelligent and therefore they need appropriate object oriented techniques and tools.

Generally, the development of high quality, object oriented applications requires:

- A rigorous object oriented software development method providing the "how-to" for the actual development tasks.
- A complete and consistent, formal modeling language with precise (execution) semantics.
- Sophisticated tools supporting the method and the language within all phases of software development.

This work describes such a method, such a language, and such a tool. Chapter 2 introduces Story Driven Modeling (SDM), a new rigorous software development method dedicated to the development of complex, graph-like object structures. SDM focuses on the "how-to" of modeling, specification, design, and implementation tasks. SDM is easily embedded in modern object-oriented software development processes like the Unified Process. To the Unified Process, SDM contributes the "how-to" for the actual development tasks. System specification with SDM is based on a formal semantics and rigorous usage of the UML. To provide a first informal idea of the execution semantics of SDM diagrams, chapter 3 introduces sophisticated code generation concepts covering code generation from class diagrams, statecharts, activity diagrams, and finally from collaboration and sequence diagrams. The detailed formal semantics of SDM diagrams is discussed in Appendix A. Provided with a formal specification language and code generation concepts, chapter 4 revisits the analysis phase and a systematic transition from the analysis phase to the design. Chapter 5 evaluates a number of currently available UML CASE tools with respect to their support for a rigorous software development method like SDM and with respect to their code generation capabilities. One of the evaluated CASE tools is the Fujaba environment, cf. www.fujaba.de. The Fujaba environment has been built during the last three years in order to provide sophisticated tool support for SDM. It implements our code generation concepts and provides reasonable support for creating complete and consistent specifications. It is public domain and open source and may be obtained from www.fujaba.de. Chapter 6 provides conclusions and discusses some future work.

SDM and our rigorous UML and the Fujaba environment are suited for all kinds of object oriented applications from office applications via internet applications via software development tools to safety critical and embedded systems. Our techniques are especially suited for the development of applications employing complex, graph-like object structures. Our approach integrates widely-used and emerging object oriented modeling methods and languages like the Unified Process and the UML. We use the UML as a well-defined, visual programming language allowing the specification of structure and behavior and runtime object structures of an object oriented application at a very high level of abstraction. Our code generation and round-trip engineering concepts free the developer from the tedious and error-prone implementation tasks. Our approach raises software development productivity and quality and the manageable project size and complexity. This way modern object oriented methods and tools become really usable for the development of high-quality, reliable, safety critical, intelligent, complex applications.

2 Story Driven Modeling (The Rigorous Software Development Process)

2.1 Overview

Current object-oriented modeling methods like UML [UML97], OMT [RBP+91], OOAD [Boo94], OOSE [CJJÖ93] etc. focus on the specification of the static structure of software objects. A major deficiency of these methods is that they provide no or only little help on how requirements are refined, who scenarios are developed, how class diagrams may be derived from scenarios, and how the behavior of active components and of central system operations is specified and how such a specification may be turned into a robust and reliable implementation. In this work, we introduce *story driven modeling* (SDM) to support analysis, design and implementation of systems with complex, graph-like, dynamic object structures. We will show that this approach bridges an important gap between rather informal, high-level analysis techniques like *use-cases* [CJJÖ93] and the formal specification of a software system.

Like common use-case driven approaches, story driven modeling starts with usual requirements analysis based on use-case diagrams, cf. Figure 1. Next, each use-case is described by a set of sample sce-

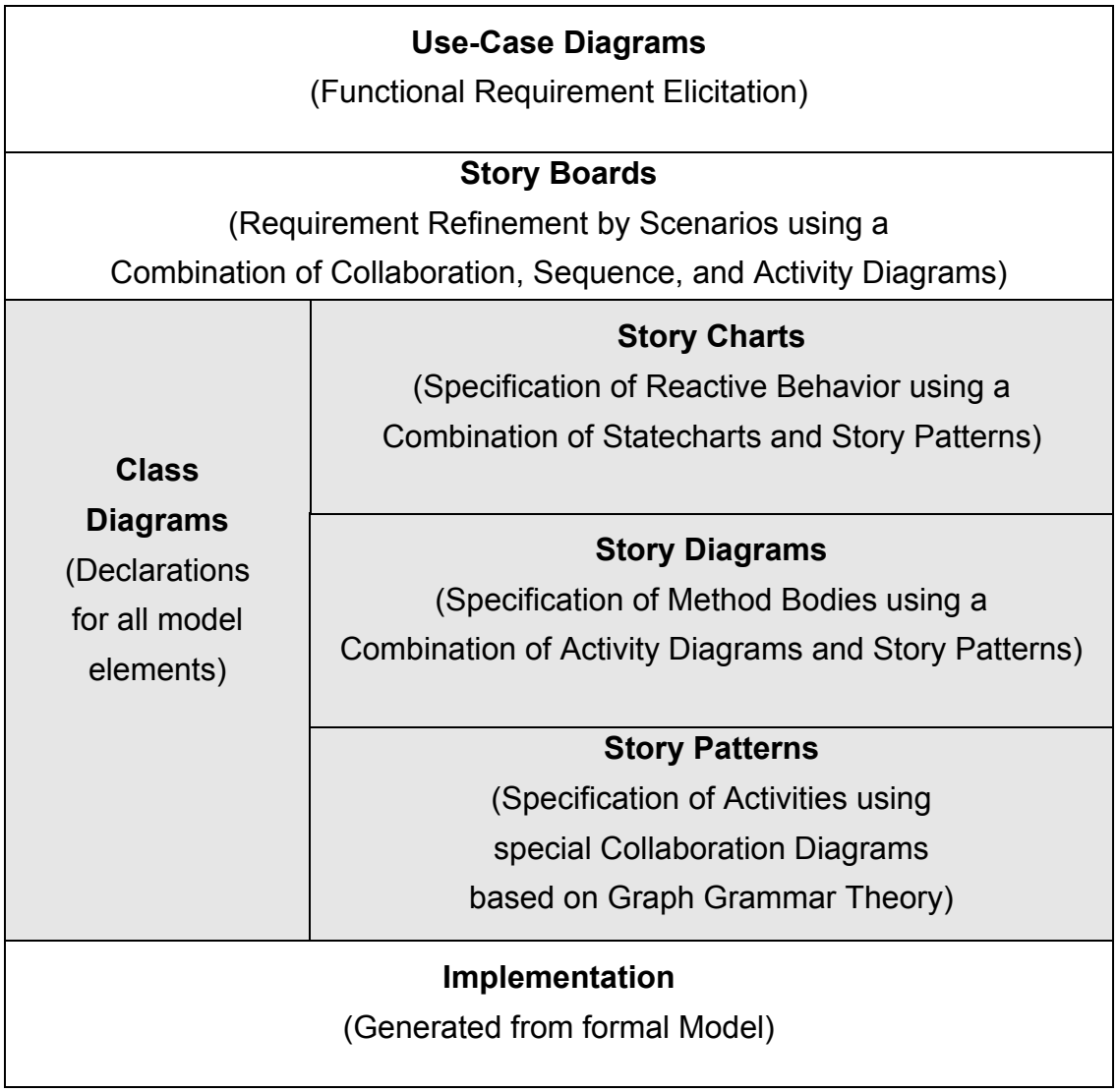


Figure 1 Story Driven Modeling

narios, so called *stories*. Such stories are defined by a new modeling technique called *story boarding*. Story boarding facilitates the specification of the dynamics of graph-like object structures as a sequence of single *snap shots*. In addition, it overcomes major deficiencies of notations like e.g. UML collaboration diagrams that allow to model the control flow of methods but are not sufficient for modeling the evolution of graph-like object structures. Due to our experiences with several case studies, story boards are an excellent vehicle to communicate ideas and concepts to team members and even to skilled customers due to their simple visual notation. On the other hand, story boards support a systematic derivation of subsequent specifications like e.g. class diagrams and method body specifications. In the story driven modeling (SDM) approach method bodies are formally specified in a high-level, graphical formalism, so-called *story diagrams*. While story boards describe example scenarios, story diagrams are operational specifications of the general case. Story diagrams are a combination of UML activity diagrams and so-called story patterns. The activity diagram part specifies high-level control flow. *Story patterns* specify the actual operations, e.g. modifications of the current object structure. Story patterns are a simplified version of UML collaboration diagrams that yield a well-defined execution semantics based on graph grammar theory. In addition, SDM provides so-called *story charts* for the specification of reactive systems. Story charts are a combination of statecharts and story patterns. All employed specification (sub)languages have a well defined execution semantics. This enables our *Fujaba* environment to generate the (Java) implementation of the modeled system from the class diagrams, story charts, and story diagrams, automatically. In addition, the Fujaba environment provides a dedicated interactive graphical browser and simulation environment for the generated Java code that allows a direct validation of the specified system. The generated Java code becomes the basis for the final implementation of the application. Fujaba is an acronym for From Uml to Java And Back Again. The "back again" part reflects that the Fujaba environment provides round-trip engineering support for the final implementation phase.

SDM employs major parts of the UML. However, in contrast to the UML, in SDM the different diagrams do not stand on their own as unrelated description parts. SDM integrates all its sublanguages and subdescriptions to a complete and consistent overall model. Each use-case is refined either by another use-case diagram or by one or more story boards. All elements employed in story boards like object(kind)s, attributes, and messages are finally declared in the class diagrams. The behavior of each active class is modeled by exactly one story chart covering exactly the events declared in the signal department of that class. Each object, attribute, and action employed in the story chart is declared in the class diagram. Each method declared in the class diagram is formally modeled by exactly one story diagram. Again, all story diagrams employ only objects, attributes, and methods provided by the class diagrams. These constraints allow many context sensitive compile time checks enforcing the consistency and completeness of the overall model.

SDM has been taught with great success in several courses at Paderborn University and on some courses for industrial participants. It has been adapted for computer science courses at various high-schools in Paderborn. SDM is used in research projects at several universities, and it has been applied to develop several sample applications which deal with complex and highly dynamic object structures, e.g. a distributed planning and information system for courses at Paderborn University and an information system for the local public transportation system of Paderborn. The Fujaba environment comprises about 380 000 lines of Java code. It is public domain and open source and can be downloaded from www.fujaba.de. Currently, we have recorded about 5900 downloads.

Within this chapter, we exemplify SDM on the public transportation system example which is simple enough to be described in a few pages and complex enough to show the strength of our approach. In the next section we will outline the main SDM activities. In section 2.3 we demonstrate the use of SDM by applying it to the development of the bus route information system of Paderborn. Section 2.4

concludes and gives some hints on current and future work. The consistency constraints of our model are revisited in chapter 5.

2.2 Process Definition

The aim of SDM is to ease analysis, design, and implementation of software components that deal with dynamic and highly complex graph-like data structures. Although, this is a very important class of software components, an entire software system probably includes other components with different characteristics (e.g. graphical user interfaces, numerical methods, file-io, etc.), that should be developed with other techniques. Note, SDM is heavily influenced by UML notation [BRJ99] and process and its predecessors. We try to stick to this new standard as close as possible and to extend it for the handling of graph-like data structure.

In the following, we will give a concise overview on 7 activities involved in SDM, which will be discussed in more detail with a case study in section 2.3.

Activity 1: Gathering Informal Requirements

We propose SDM as an extension and refinement of use-case driven analysis and design as proposed by the unified process [CJJÖ93, JBR99]. Thus, the first activity in SDM is to elicit and analyse informal requirements of a software system by defining a so-called use-case model. There are two main concepts in a use-case model, namely *actors* and *use-cases*. At this, an actor represents a particular role users can play, while a use-case is defined as a sequence of transactions in a dialogue with the system.

Activity 2: Story Boarding

The term *story boarding* goes back to the very beginnings of cinema. It got acquainted by Walt Disney who used a board with subsequent pictures to outline the story of his animated movies. Recently, this idea has been adopted in the software engineering domain as a paper and pencil technique to design user interfaces by sample scenarios or define classes with their properties by e.g. using CRC cards [Boo94 p. 239]. In [BRJ99] event trace diagrams, collaboration diagrams, statecharts, and activity diagrams allow to outline sample interactions within a given object structure. [CD94] proposes to illustrate intermediate states of ‘event scenarios’ by object diagrams. In our approach, we pick up these ideas and extend them with respect to the definition of evolutionary aspects of complex object structures. Moreover, we provide story boards as an intermediate language that supports the derivation of class diagrams and method specifications in the next activities.

In our approach a number of story boards is defined for every identified use-case. Each story board consists of a series of *snap shots*, which describe the evolution of the system’s object structure for a particular sample scenario. At this, it is important to note that story boarding is designed as an activity where developers only describe the idea how the system will work. They do not yet have to worry about algorithmic details and all the pitfalls to be regarded in a complete problem specification. Furthermore, story boards are excellent subject for discussions of the outlined ideas between different developers (and even skilled customers): “*Having additional team members participate in story boarding is a highly effective way of teaching junior developers, and communicating the architectural vision.*” [Boo94]

Activity 3: Deriving the Static Class Structure

Given a number of story boards it is easy to derive informations about the static class structure of a software system. Here, we reuse many ideas from [RG92]. In a first step we identify classes for

objects with the same properties. Then we define inheritance hierarchies by searching for classes with common properties. Furthermore, we may derive associations and aggregations and even some cardinality constraints from the sample situations defined. In addition, first methods are derived, e.g. from the use-case diagrams and by assigning the responsibility for certain (sub)steps to certain object (kinds). Note, in general activity 2 and 3 are highly intertwined and executed in parallel.

In terms of [JBR99], story boards and this first class diagrams define an *analysis model* for our application.

Activity 4: Providing a Design Model

In this activity we revisit the story boards of activity 2 and develop ideas how the system will work. Usually, we will have to rework existing object structures towards algorithmic needs and add facilitating and temporary data structures. This will result in more detailed story boards that give a glance on the intermediate steps and on the data structures of the employed algorithms. In addition, one derives extended and refined class diagrams that comprise the additional information and methods. Thus, this activity results in a *design model* of the desired application, cf. [JBR99]. Of course, there is no sharp separation of activities 2, 3, and 4 and in practice these activities will be intertwined. However, these activities address different aspects in the overall system analysis and design phase which are worse to be mentioned individually.

Activity 5: Deriving Dynamic Operations

In this activity the ideas outlined in the story boards and class diagrams of the previous activities are elaborated. Now we look at the concrete object structures and algorithms that realize the desired system. In SDM this is done using so-called *story charts* and *story diagrams*. These languages are an adaption and enhancement of programmed graph rewriting systems [SWZ95] to the UML notation and the object oriented data model. Story charts combine UML statecharts and collaboration diagrams to a powerful visual programming language for state based systems and for the specification of event handling in active processes. Story diagrams combine UML activity diagrams and collaboration diagrams for the specification of methods that deal with complex graph-like object structures. In addition, the story chart and story diagram notations are very similar to story boards, thus facilitating the transfer of analysis and design results to these specification and implementation activities. However, while story boards describe particular sample situations, story charts and story diagrams specify the general case. In other words, story charts and story diagrams specify a general system that implements the sample scenarios described in story boards.

Story charts and story diagrams are a major strength of SDM compared to most other approaches. Most other approaches employ plain programming languages for the implementation activities and thus face a severe semantical gap between analysis & design and the implementation activities.

Activity 6: Validating the Model

Having finished the specification of the class structure and operations implemented by a set of story diagrams the model should be validated. For this purpose our *Fujaba environment* provides code generators that translate class diagrams and story charts into a standard Java implementation. In addition, the Fujaba environment provides a generic object structure browser with automatic layout capabilities that enables the developer to invoke methods on objects interactively and to keep track of the effects of called operations by depicting the changed object structure, directly.

Activity 7: Developing non-graph-like system parts

If the functionality of the specified system is satisfying, the not-yet mentioned task is to realize the non-graph-like system parts. These parts may employ other development approaches and tools, e.g. component technologies and modern user interface builders. For simple applications, the Fujaba environment provides a small library that allows to create a HTML based user interface, easily.

Disclaimer: This chapter describes Story Driven Modeling as a relatively strict, water-fall-model like sequence of activities. Our intention was to identify the central SDM activities and their different characteristics. However, these activities are closely related and intertwined. They may be executed in a more explorative life-cycle, too, e.g. [Boe88]. In fact, the SDM activities are easily embedded into the unified process [JBR99] extending or replacing the more technical activities e.g. in the responsibility of the component engineer.

The central contribution of SDM is the deployment of story boards for analysis and design and of story charts and story diagrams for specification and implementation purposes. Story boards extend the usual OO analysis by considering not just nouns and central terms and single object configurations (or collaborations) but by analyzing the changes to the object-structures in sequences of execution steps. Story charts and story diagrams allow the implementation of complex manipulations of graph-like object structures on a high-level of abstraction.

2.3 Case Study: Paderborn bus route information system

Activity 1: Gathering Informal Requirements

As already mentioned in section 2.2, we adopted a use-case driven approach as described in [CJJÖ93, JBR99] to analyse and denote requirements of a software system.

The running example for this paper is a small program we have developed for the bus service provider of Paderborn. Our bus service provider wanted an on-line bus route information system BusRoute. BusRoute had to provide easy-to-use query facilities to show bus time-tables and to compute bus routes through the town. In addition, our bus service provider had some special requirements on the routing algorithm that were not met by available standard tools. They required a platform independent system and access via World-Wide-Web.

Figure 2 shows a (simplified) use-case diagram for the BusRoute system. BusRoute supports two kinds of users, customers and operators. These users with their associated roles (so-called actors) are depicted as stickmen in Figure 2. Each actor may use BusRoute to perform a particular set of actions (so-called use-cases) which are represented by ovals. Note, that use-case diagrams allow refinement and extension of use-cases which is not used in our example, cf. [CJJÖ93, BRJ99].

The operator performs two major activities. First, he is allowed to edit, load, and store (new versions of) the bus plan. The bus service provider of Paderborn maintains the bus plan using a standard spread sheet calculator. This format is close to the result format of SQL queries on standard relational databases. BusRoute is able to parse bus plan data provided in this way. Later on we want to be able to access bus plan data via a JDBC interface. The use-case "optimize bus plan" depicts activities, when the operator adds or adjusts informations on transfer possibilities or groups co-located bus stops thus permitting additional transfers at these locations. Customers use BusRoute mainly to query for bus time-tables and for bus connections from one stop to another in certain time ranges. This is done via a simple WWW based user interface that allows to select departure and arrival stops and a desired time slot, cf. Figure 12. Another frame shows the query result in a scrollable text area, cf. Figure 13.

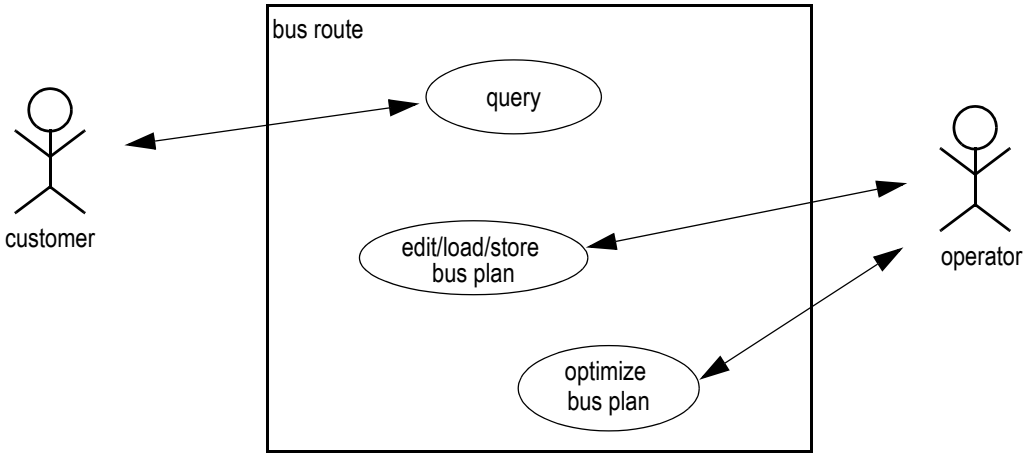


Figure 2 BusRoute use-case diagram

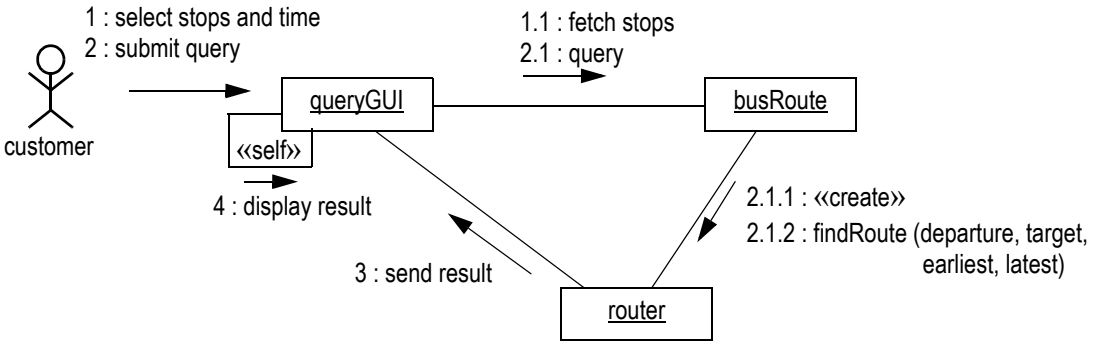


Figure 3 Collaboration diagram for use-case query

Activity 2: Story Boarding

Usually, after requirements elicitation the next step in OO analysis is the derivation of an initial conceptual class diagram, cf. [JBR99]. This can be done by inspecting the different use-cases and by identifying the participants of these use-cases and their corresponding roles and relations. Later on the conceptual class diagram is complemented by describing the different use-cases in more detail using collaboration diagrams. However, this is an extremely critical and challenging activity. Due to our experiences in using and teaching UML, the direct identification of classes and their roles and their relations and the direct derivation of a class diagrams is very difficult for most people. The problem is that class diagrams describe a system at a structural level or more precise at type level. Most people find it much easier to think in concrete objects and links and examples i.e. to think at the instance level. Thus, we propose to analyse use-cases using (extended) interaction diagrams, first, and to postpone the derivation of the conceptual class diagram to the second step.

In our example, we start to refine use-case query with a collaboration diagram, cf. Figure 3. In Figure 3, the customer first selects departure and target stops and the desired time via a queryGUI object that represents the graphical user-interface. As a sub-step of step 1, in step 1.1 the queryGUI fetches all available stops from the busRoute object. Once the customer has selected stops and time, he or she submits his query, cf. step 2. In reaction to step 2, the queryGUI forwards the query to the busRoute object, cf. step 2.1. In step 2.1.1, the busRoute object creates a router object for this query and in step 2.1.2 it asks the new router to find an appropriate route. Once the route is computed, the router submits the result to the queryGUI object, step 3, and the queryGUI displays it to the customer, step 4.

Collaboration diagrams are an excellent means to analyse details of object structures and to model the flow of messages between the depicted objects and to some extend to show creation, deletion and modification of participating objects. However, collaboration diagrams tend to become crowded if to many operations or different cases are to be shown. More complex computations may require several logical steps where each logical step operates on a certain detail of the current object structure. Sometimes the computation may involve branches and iterations on such logical steps. Thus, one may want to employ several collaboration diagrams to model a more complex computation and he or she needs additional control structures to guide the execution of these collaboration diagrams.

In SDM we call such a series of collaboration diagrams that refines some use-case, a *story board*. A story board illustrates the evolution of details of the system's object structure as sequence of *snap shots*, i.e. collaboration diagrams. Each snap shot or collaboration diagram shows a detail of the object structure that represents a current situation and how this object structure will evolve and how messages are passed between the objects. For representing control structures on snap-shots we adapt UML statecharts and activity diagrams. In UML, the actions performed by states and activities are shown as pseudo code, only. In SDM we replace this pseudo code with collaboration diagrams, i.e. we draw collaboration diagrams within the state and activity shapes of statecharts and activity diagrams, cf. Figure 4.

We now use Figure 4 to exemplify the use of story boards and to introduce their notation in more detail. Figures 4 shows the story board *findRoute* that refines message 2.1.2 of Figure 3. Snapshot 1 shows 4 objects, the stops airport and uni and the busRoute and router objects taken over from Figure 3. Note, to facilitate recognition, one may use additional icons to depict the type of some objects. In snapshot 1, the just created router creates rfrom and rto links to its departure and target stop, respectively. Note, we have omitted the numbering and the arcs of the «create» messages for simplification. In addition, the router stores the earliest and latest departure time in corresponding attributes. This is done using the assign operator := in the attribute compartment. Note, to emphasize object-structure modifications, we use light gray or green color for created elements and attribute assignments.

In our example the router has the task to compute a tour from the departure to the target bus stop. This computation may start by considering appropriate busses leaving the departure stop within the allowed time range. Therefore, Snapshot 2 of Figure 4 shows three move objects that model busses leaving the airport. Each move shows its departure and arrival time. Thus, in our example only moves move2 and move3 fall into our time range. Snapshot 2 has now the task to mark the appropriate moves somehow. At the first iteration, we used simple links between the chosen moves and the router for this purpose. However, later on it turned out that explicit marker objects, like hopA and hopB, do a better job.

Snapshot 3 of Figure 4 is close to the maximum reasonable size of a single snap shot / collaboration diagram. It models the central search step of our routing approach. A simple routing algorithm would just do a breadth first search through the moves and stops object structure. We could do so by considering one hop object after the other and by visiting the target stop of the corresponding move and by looking for appropriate departures and by marking these departures with new hop objects and so on, cf. Figure 5. However, this approach turned out to be too inefficient. At each stop the simple algorithm has to search through the set of departures to find appropriate moves within a certain time frame. But, usually there is only one option, i.e. to stay in the bus that just has arrived. Only some stops really offer a transfer possibility to other busses. In order to exploit this domain knowledge, we introduce the notion of *drives*. A drive object collects a list of moves visited by a bus during a drive from its start stop (depot) to its terminal stop. In addition, we mark stops that offer transfer possibilities with a transfer flag. Equipped with this additional information, in the example snap shot 3 of Figure 4 we just look up the successors of move3, i.e. move31 and move32, until our example drive reaches a transfer stop, i.e.

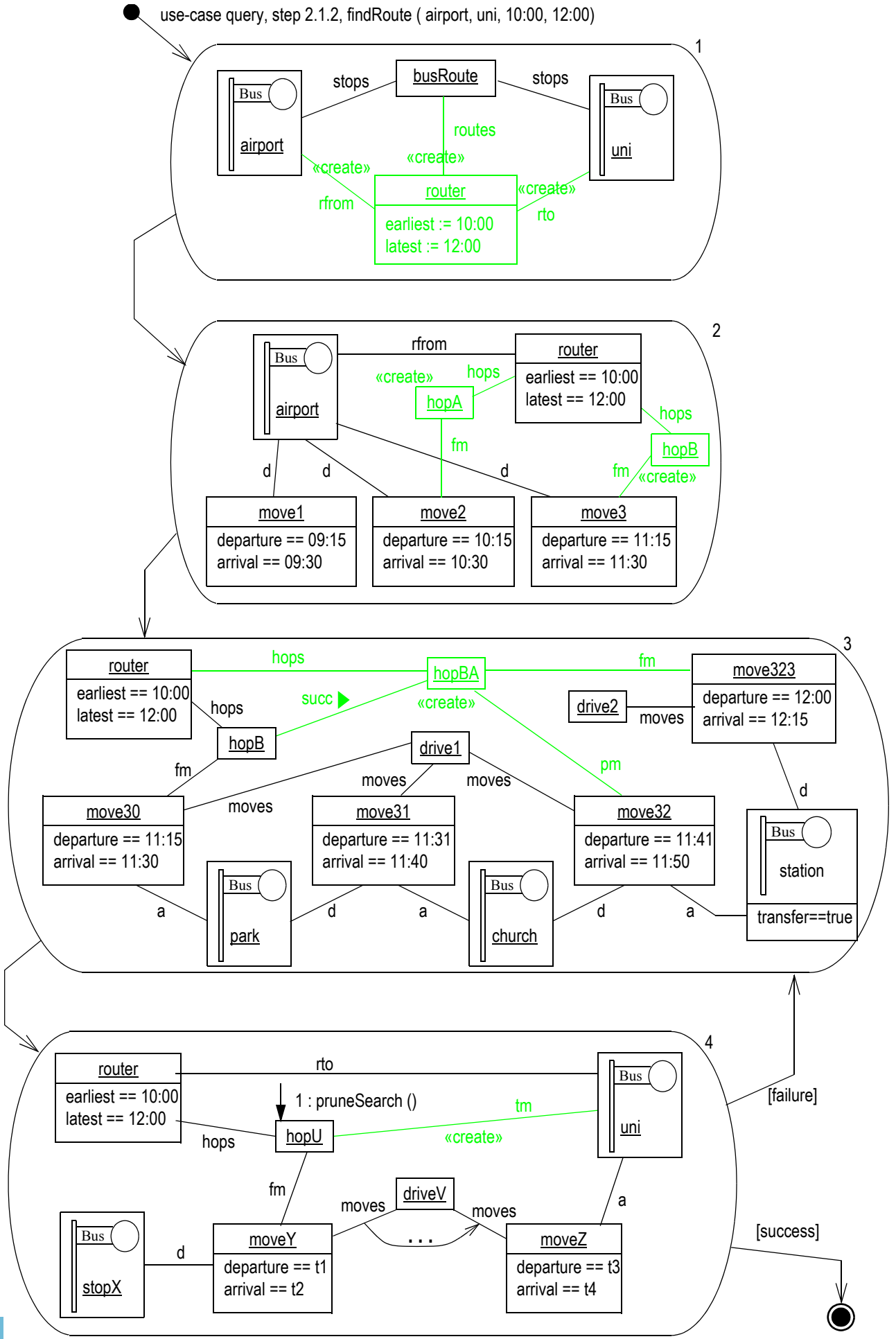


Figure 4 Story board findRoute

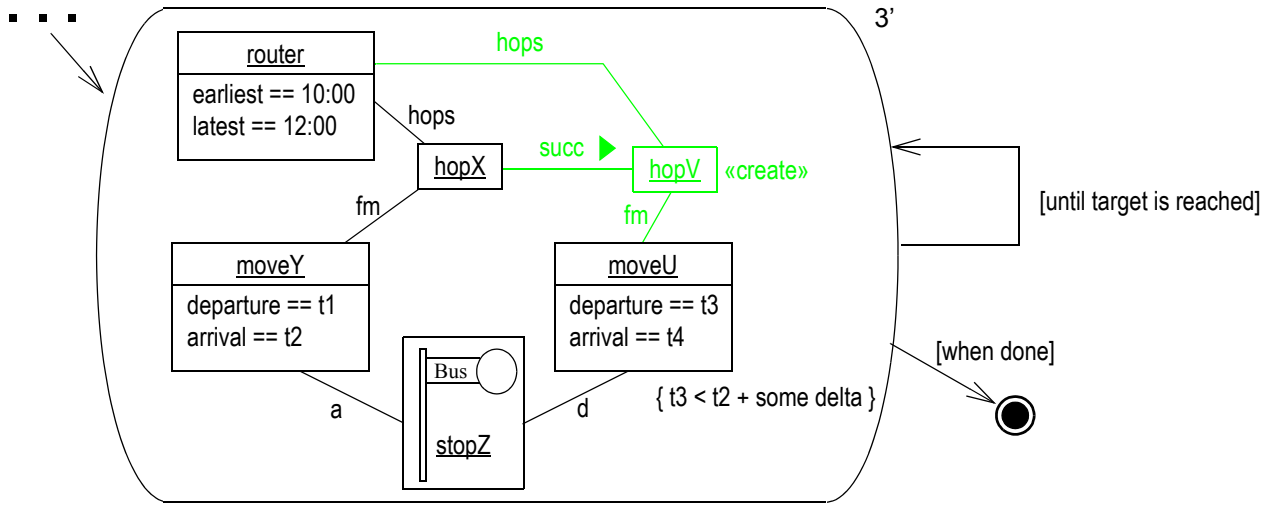


Figure 5 Outline of a simple routing idea

the station. At such a transfer stop we look for departures belonging to other drives (in our example drive2) and we mark those drives via a new hop object, e.g. hopBA.

Snap shot 4 of Figure 4 shows how our routing algorithm terminates. The target stop is marked by an rto link from the router object, in our example stop uni. Once a drive reaches this target stop, we create a tm (**t**arget **m**ove) link to the corresponding hop, e.g. hopU, and the routing may terminate. If the current drive does not reach the target stop, we go back to snap shot 3 and look for alternative transfer possibilities.

We have discussed Figure 4 very detailed in order to show how story boards may be used to outline the ideas of an algorithm. This discussion has shown, that a single collaboration diagram often does not suffice to illustrate complex computations that include several object structure evolution steps. Story boards solve this problem by combing several collaboration diagrams with a control flow description borrowed from activity diagrams. Each step may show a certain detail of the object structure under investigation and certain modifications and certain message invocations. Note, Figure 4 shows only one message invocation, i.e. message pruneSearch send to hopU in snap shot 4. Figure 4 shows an untypically low number of messages, since our example emphasizes the object structure evolution through several logical steps. However, some operations or logical steps may deal with complex message flows between a small number of objects with only little object structure modifications. In such situations the snap shots will show more messages and it may also be more appropriate to use a usual collaboration diagram or a sequence diagram to refine an operation. Note, one may also use a sequence diagram as a snap shot within a story board.

To summarize, story driven modeling proposes that one should start the OO analysis work with behavior diagrams. One may start using statecharts to model interactions between active processes and using collaboration diagrams and sequence diagrams to model complex message flows. Story boards complement this set of UML behavior diagrams. Story boards are especially suited to model complex object structure evolutions within several logical steps. They are very well suited for discussions on white boards in order to develop ideas for algorithms working with complex object structures and may easily be combined with the other UML behavior diagrams. The behavior diagrams that model example situations may then serve as input for the development of the analysis model class diagram.

Activity 3: Deriving the Static Class Structure

In story driven modeling, a analysis model class diagram is easily derived from objects and links used in the behavior analysis activities. Basically, one steps through the story boards and through the other

behavior diagrams and provides classes, attribute declaration, method declarations, and associations for all used objects, attributes, messages, and links, respectively. For each element one may either add a new declaration to the class diagram or one may identify an existing class diagram element that already describes the current behavior diagram element.

This step can be supported by a tool that highlights and/or lists behavior diagram elements that have not yet been assigned to a class diagram element. In addition, a tool may check the consistent use of attributes, methods, and associations. For example, a tool could identify that in some diagram a certain link is attached to an object with a type other than usual. Such situations might indicate an inconsistency or they may be resolved by introducing an appropriate inheritance relationship. In addition, a tool may derive the types of certain behavior diagram elements, semi-automatically, as soon as some type information has been provided. For example, from snap shot 2 of Figure 4 one may derive a class Move with attributes departure and arrival and an association d connecting one stop with multiple (departing) moves. Equipped with this information, a tool may identify the d links of snap shot 3 as instances of this association and derive that the connected objects are moves and stops, respectively. Recognizing the attributes of move3, move31, and move32, these objects seem to be instances of class Move and in turn station, park, and church seem to belong to class Stop. The black parts of Figure 6 show the class diagram derived from Figure 3 and Figure 4. The gray parts stem from later refinements of the story boards and from the behavior specification activities.

Within common development processes, e.g. the unified process, the first step of the analysis phase is the derivation of an initial class diagram from the requirements specification. A common approach is to underline nouns in natural language descriptions in order to derive 'real world' classes. In addition one looks for simple phrases indicating relationships between such classes. Once an initial class diagram has been derived it becomes the starting point for the analysis of scenarios with the help of UML behavior diagrams. Our experiences in using and teaching UML indicate that starting this way

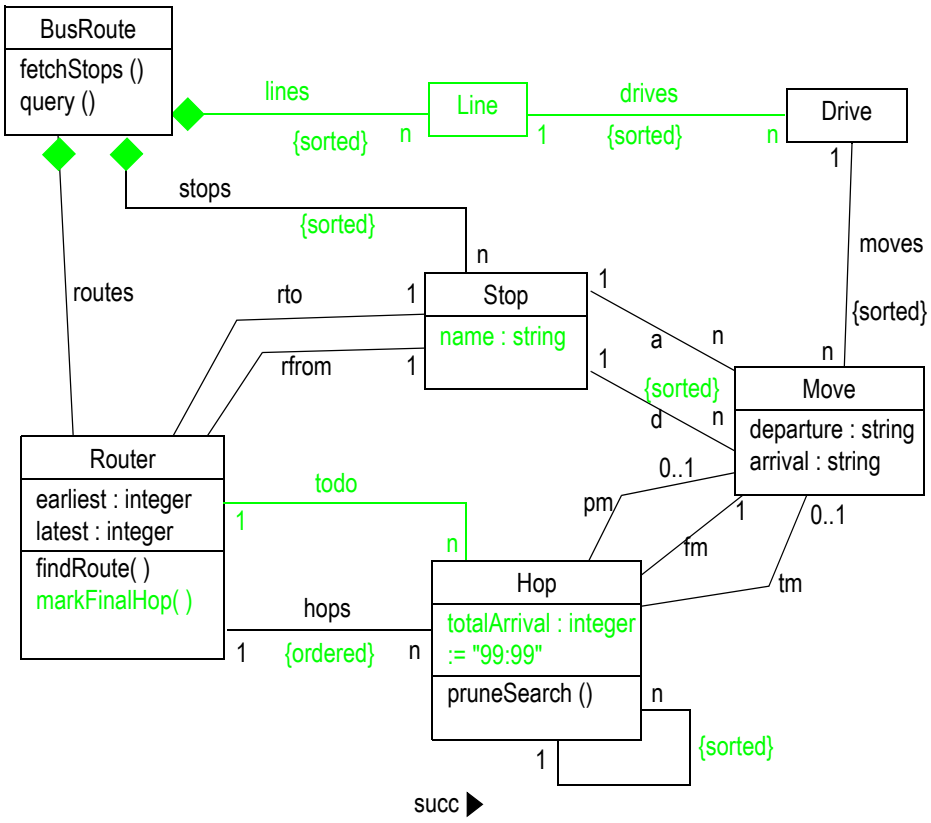


Figure 6 BusRoute core class structure

is wrong. It is quite hard to come up with a reasonable class diagram derived from requirements specification, directly, since class diagrams describe the static structure of the application but not the dynamic object structures employed at run-time. Thus, we propose to analyse requirements specifications by studying scenarios with the help of UML behavior diagrams, *first,* and to derive the class diagram in a second step. Due to our experiences this approach works much smoother and people proceed much faster and develop more reasonable class diagrams.

This paper splits scenario analysis via UML behavior diagrams and the derivation of class diagrams into two subsequent activities. This emphasizes the importance of doing instance-level scenario analysis, first, and class diagram derivation, second. However, these activities are usually heavily intertwined and one may work on the class diagrams, earlier. For example, one may derive the first parts of a class diagram once he or she has refined the first use-case by some scenarios. Having such a partial class diagram at hand facilitates to keep up the consistency of old and new behavior diagrams during the analysis of other use-cases. Such an approach results in incremental development of new behavior diagrams in parallel with the maintenance and extension of a common class diagram.

Activity 4: Providing a design model

So far, we have developed an analysis model of our system. That means, we have identified objects and associations and typical behavior, that models the systems view of the outside world. In more complex systems, one probably still faces a big gap between this analysis model and an executable specification of the target system. Activity 4 aims to bridge this gap by looking on possible algorithms and auxiliary object structures. Thus, we now develop ideas on *how* the system could work.

Therefore, we revisit the story boards developed in activity 2 and refine them by adding additional intermediate steps, i.e. snap shots, that clarify how the described effects are achieved. Usually, we will extend our object structure by auxiliary structures that store intermediate results and guide the algorithmic execution. This involves the corresponding extension of the class diagram derived in activity 3. The result of this activity are refined story boards and class diagrams that outline the algorithms and object structures our system employs to achieve the functionality described in the analysis model.

In addition, we break down complex story boards into smaller methods and assign these methods to suitable objects or classes, respectively. Note, that the assignment of methods to classes is an important design step which needs to be executed with care. One has to ensure that the resulting methods have easy access to all necessary information and suboperations and are easy to access by their callers. We want to achieve maintainable and reusable classes with logically correlated methods where each class carries responsibility for a distinct aspect of system functionality.

Note, one needs some experience in story boarding and applying the story driven modeling approach until he or she is able to decide whether a story board is detailed enough in order to proceed with the next development activity or whether it still needs more analysis. Basically, a behavior diagram is detailed enough as soon as for each elementary step it is clear how that step works in the current example situation. One has to answer questions like: Which object is responsible for this step? Are there enough links such that the responsible object is able to determine all other objects it is collaborating with? Is all necessary information at hand, in order to provide a message invocation with its parameters? Is it possible to evaluate the control flow conditions? Answering these questions often creates intensive team discussions and triggers a lot of reasoning about design trade-offs. This process frequently causes new design decisions and extensions of the employed object structures and changes of the corresponding behavior diagrams and of the class diagrams. Actually, within such steps most of the (analysis and) design work is done. From our experiences, it has turned out that story boards are an ideal basis for such design discussions and for the documentation of design rationals. Of course,

the final outcome of this process is a class diagram that specifies the static structure of the program to be build. However, the story boards tell the developers which functionality the methods they are going to realize should have.

Of course, the borders between analysis and design model are not sharp. For example, one may consider the story boards of Figure 4 and the *Hop* objects we introduced to outline the idea of our routing algorithm as part of the design model. Since the idea of our algorithm is already clear, within this paper we skip a more detailed elaboration of design story boards.

Activity 5: Specifying Dynamic Operations

In the recent activities, we used story boarding as a technique to analyse the object structure of our example system and to describe how this structure might evolve in typical situations. In this section we introduce *story charts* and *story diagrams* as a high-level formalism to specify system behavior, operationally. Story charts and story diagrams consist of extended UML statecharts and activity diagrams, respectively, where the contained actions are specified by so-called *story patterns*. Story patterns define tests and transformations on the system's object structure. Their notation is very similar to the notation of snap shots in story boards. However, while story boards describe the evolution of an object structure in a particular sample situation, story charts and story diagrams are used to specify the general case. Note, the Fujaba environment is able to generate executable Java code from story charts and story diagrams, automatically. Thus, one may view the specification with story charts and story diagrams already as the implementation of the application since no manual implementation activity is required, anymore. Before we proceed specifying our example, the next two chapters introduce the semantics of story patterns and story charts as they are employed in story driven modeling.

Using UML collaboration diagrams for visual programming

The basic element of formal behavior specification in SDM are so-called *story patterns*. Story patterns allow to model on a high level of abstraction operations that modify complex object structures. Story patterns will be used to specify the basic actions embedded in statecharts and activity diagrams used to describe the behavior of active objects and method bodies, respectively.

Story patterns are based on UML collaboration diagrams. Originally, collaboration diagrams are intended to model scenarios of complex message flows between a group of collaborating objects. In this context, collaboration diagrams do not have a precise execution semantics. In order to use collaboration diagrams as a visual programming language one has to add a lot of details on how the participating objects are found and how object structure modifications are executed. As an example, Figure 9 shows a detailed collaboration diagram for the first activity of the story diagram Router::findRoute, cf. Figure 10.

The collaboration diagram of Figure 9 employs 5 objects that are connected via various links. The collaboration messages 1 to 3 look up associations attached to the this object and fill variables s, set1, and m with appropriate values. Note, according to the class diagram d is a to-many association. Thus, looking up the d associations results in a set of objects represented by the multi-object set1. Step 3 loops through this set of objects and assigns them to variable m, one after the other. At this point, candidates for all participating objects are found. However, in our example we look for moves that depart within a certain time range. This is checked in steps 3.1 and 3.2. If these conditions are fulfilled, the actual operations can be executed. Step 3.3 creates a new Hop object h. Substeps 3.3.1 to 3.3.3 create the attached todo, hops, and fm links, respectively. Finally, step 3.3.4 changes the value of the totalArrival attribute of h from x to this.latest+120.

Provided with such detailed collaboration messages, code generation becomes very simple. However, this usage of collaboration diagrams adds little abstraction compared to usual (pseudo) code. We raise

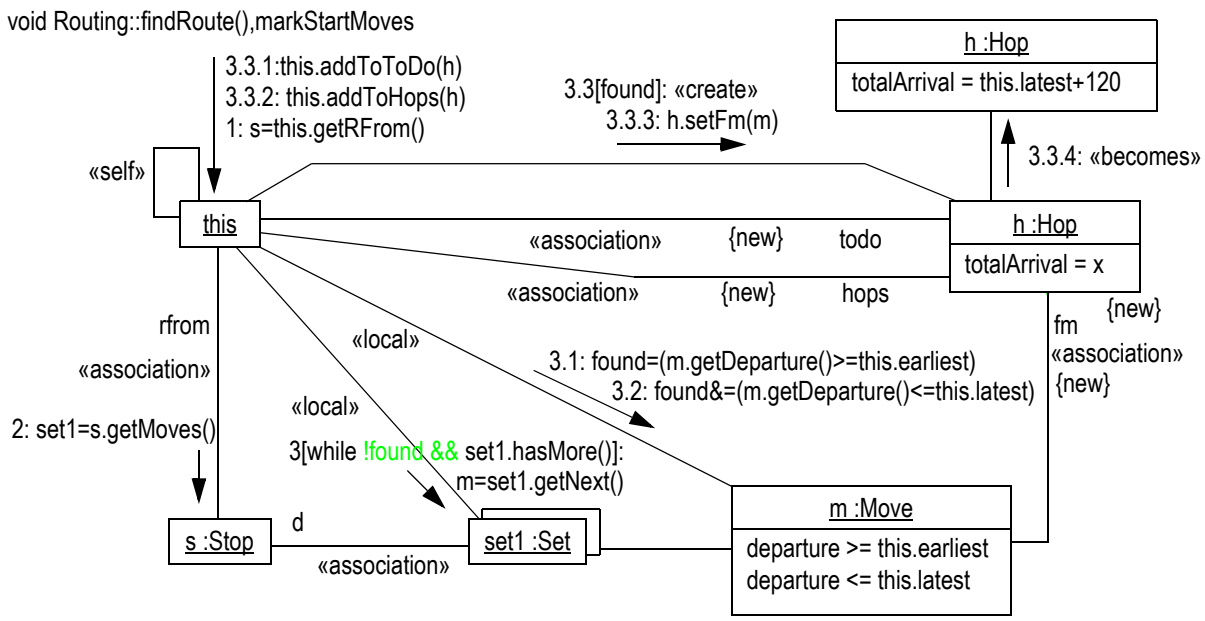


Figure 9 A collaboration diagram showing sufficient details for code generation

the level of abstraction by assigning a standard semantics to certain graphical elements of collaboration diagrams and by a systematic simplification of required elements. For the collaboration diagram of Figure 9 this will result in the simplified collaboration diagram shown as do action of activity 1 at the top of Figure 10.

In our context, a collaboration diagram specifies the body of a certain method (part). At this, all collaboration messages originate from the this object. All boxes depict local variables of the current method or globals declared in the class diagram. Thus, it is clear which kind of objects are denoted and all «self», «global», and «local» links may be omitted. Finally, only «association» links will remain. Thus, the «association» stereotype is superfluous, too. Next, it is clear, that the method variables have to be filled using the depicted links between the objects. The Fujaba code generator is able to compute the necessary look-up operations, automatically, cf. Activity 5 1/2 and [Zün96b, FNTZ98]. Thus, one may omit the association look-up operations (step 1 to 3). In Figure 9, the multi-object set1 is used to look up the d association and to loop through its elements. Our code generator derives these two steps, automatically. Thus, a single d link from s to m suffices. Next, the depicted {new} constraints indicate the creation of the corresponding elements, clearly. Thus, step 3.3 and its substeps 3.3.1 to 3.3.3 can be generated by our code generator and may be omitted. Additionally, we simplify the frequent case of attribute assignments by allowing inscriptions like totalArrival:=this.latest+120 in the attribute compartment of an object. Thus, the two occurrences of the Hop object h and their connecting «becomes» link may be combined to a single box.

The top activity of Figure 10 shows the collaboration diagram resulting from the systematic simplification of Figure 9. Note, in our example all collaboration messages have been replaced by assigning a standard semantics to the corresponding graphical elements. However, story patterns may still contain collaboration messages, e.g. in order to invoke certain methods on identified objects or to send a signal to an active object. Story patterns just drop the superfluous collaboration messages that are used to identify the participating objects and to modify the identified object structures since this is already specified by the graphical elements.

In addition to the discussed simplifications, Figure 10 uses alternative annotations to tag items that are to be created or destroyed. In UML collaboration diagrams these values are tagged with {new} or {destroyed} constraints, respectively. However, experiences in applying story driven modeling and in teaching it to students and industrial users have shown, that these constraints are easily overlooked.

Thus, in previous papers we used series of + symbols and cancelling by two parallel lines to indicate creation and deletion, respectively. In addition, we use light grey (or green) color for to be created elements and dark grey (or red) color for the destroy marker in order to facilitate the recognition of the before and after situation. Although these tags were easily recognized, our users complained that their semantics is not very intuitive and that they are not UML conform. In several interviews the majority of our users voted to use «create» and «destroy» tags instead. They think that these statement like tags are still easy to recognize and that their semantics is very intuitive.

In our usage, collaboration diagrams depict the effects of operations in terms of changed attribute values and created and destroyed objects and links. Thus, the initial situation modeled by a collaboration diagram corresponds to the left-hand side of a graph rewrite rule. Accordingly, the situation resulting from the execution of the collaboration diagram corresponds to the right-hand side of that graph rewrite rule. This view allows the execution and translation of collaboration diagrams using code generation techniques known from the graph grammar field, cf. [SWZ95, Zün96b, FNTZ98]. In addition, the rich graph grammar theory facilitates the proof of complex system properties, cf. [Roz97] for an overview of graph grammar theory and [JZ99] for an application of this theory to the database re-engineering field.

Using story charts for the specification of interacting processes

Figure 7 and Figure 8 show the specification of the reactive behavior of classes, BusRoute, and Router, respectively. We have designed our example to employ one central server process, i.e. one active object of type BusRoute. A customer connects to our BusRoute server via an usual Web-Browser implicitly sending a getStops request. The BusRoute server will respond with a HTML query page containing available stops and fields for the desired time range. The customer selects a start and a target stop and the desired time range within the query-page and submits his or her query to the BusRoute server. Since the routing algorithm itself may need some time, we have decided that the BusRoute server does not process the queries itself. Instead, for each query the BusRoute server forks a new active subobject of type Router and forwards the query to that router. The different routers process their queries concurrently and return the answer to the corresponding customer, autonomously.

In story driven modeling the behavior of such active objects is specified by an UML statechart attached to their class. This statechart models the internal states of the active object and how it reacts to received events in terms of state changes and in terms of triggered actions and in terms of events send to other active objects. Note, our approach assumes a set of concurrent active objects where each single object is controlled by its own statechart. The different active objects run independently and possibly with different pace, depending e.g. on computational demand and on processor speed and on CPU access granted by the responsible operating system. Thus, it may happen that one process generates multiple events while the receiving process still handles some other event. However, such events should not just be lost only because the receiver is too slow.

The code generation of Fujaba deals with this problem according to the UML semantics, cf. [RJB99, UML97]. For each active object, we employ an event queue where new arriving events are stored, intermediately. The active object listens to its event queue and consumes the events from this queue one after the other. The class of an active object declares all kinds of signals the active object is able to understand. The code generator of the Fujaba environment translates such signal declarations into usual methods of the corresponding class. The body of these methods creates an appropriate event and pushes the event to the internal event queue and notifies the event consuming thread and then the signal method terminates. Thus, sending an event is done via an usual method call on the active object. Note, this implies that in our approach all events are explicitly targeted to their receivers. Event broadcasting is not supported, directly. However, our runtime library provides a generic event broadcaster.

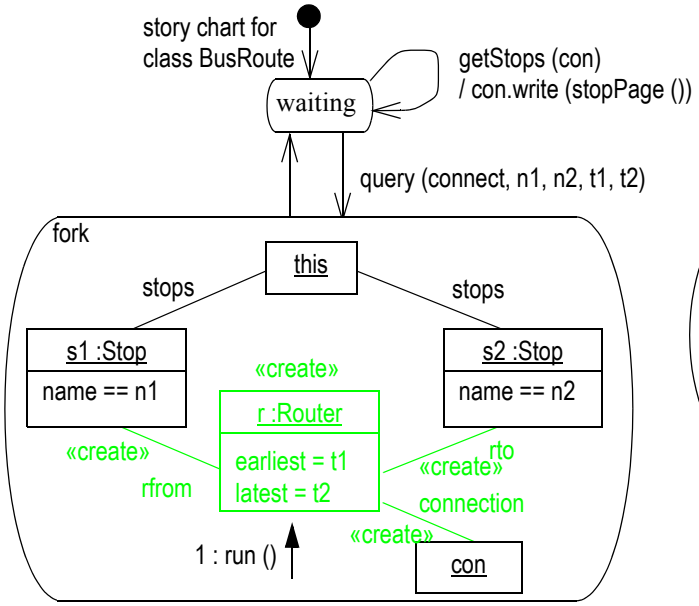


Figure 7 Story chart for class BusRoute

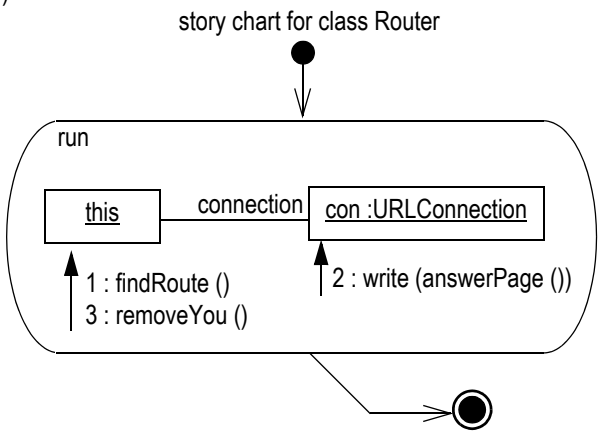


Figure 8 Story chart for class Router

Events submitted to this broadcaster will be copied and forwarded to all processes that have subscribed themselves to this service.

By definition, usual statecharts deal with abstract states, only. They specify how an active object reacts to received signals in terms of changes from one abstract state to another and in terms of transition actions and in terms of exit, entry, and do actions attached to the involved abstract states. These actions are described using pseudo code or some standard programming language. Story driven modeling replaces this pseudo or programming language code with story patterns, thus allowing the specification of operations on concrete object structures on a high level of abstraction. This combination of statecharts and story patterns results in so-called *story charts*. Story charts employ not only abstract states but allow to deal with concrete object structures. Due to the precise semantics of either statecharts and story patterns, our code generator is able to translate story charts in executable Java code, automatically, thus turning story charts into a high level visual programming language for active objects.

In our example, the story chart of Figure 7 employs two states. Usually, the server is in state waiting. When a getStops event is received from some customer our server just computes an html page showing the available bus stops and writes it to the URL connection con, passed as event parameter. Next the customer might choose a start and target stop and an appropriate time range and issue a query. Since the routing algorithm itself might need considerable computation time, our server forks a new thread that processes the query and returns to state waiting, immediately. This is modeled by the collaboration diagram shown in the body of the fork state of Figure 7. This collaboration diagram creates a Routing object r, stores the desired time range within the earliest and latest attributes, connects r to appropriate stops s1 and s2 and to the URL connection con, to be used for the reply. Finally, state fork sends a run message to the new router r, thus starting the corresponding thread.

Figure 8 specifies the behavior of a router. A router first calls method findRoute on itself. This will compute appropriate bus routes. Second, the router sends an html answer page via the attached URL connection. Once its job is done, in step 3 the router sends itself a removeYou message that frees the intermediate object structures created by method findRoute and the router thread terminates.

Using story diagrams for the specification of complex method bodies

For the specification of method bodies of passive objects or of methods employed in story charts, SDM employs so-called *story diagrams*. Story diagrams are based on UML activity diagrams. Activity diagrams allow the visual representation of complex control flows. Like in UML statecharts, the actual actions of activity diagrams are given is pseudo or programming code, only. Story diagrams replace this pseudo or programming code by so-called *story patterns*. This results in a high-level visual programming language dedicated to the specification of operations on complex object structures. In the following we will outline the story diagram notation in more detail and introduce additional language features.

As already discussed, for to-many associations contained in a story pattern our code generator creates (nested) search loops in order to identify neighbor objects that participate in the collaboration. Usually, these search loops terminate as soon as appropriate candidates are found for all objects in the collaboration. Then, the effects of the story pattern are applied to this single *match*. However, there may exist multiple sets of candidates, that would fit within the collaboration, i.e. multiple matches. In certain situations, one may want to apply the depicted operation to all these matches. Therefore, story diagrams (as well as story charts) provide so-called *iterated story patterns*, indicated by two stacked activity shapes, cf. Step 1 of method findRoute of class Routing shown in Figure 10. For an iterated story pattern the search loops do not terminate after the first match but they enumerate all possible matches and the effects of the story pattern are applied to each match. Thus, in Figure 10 the story pattern of Step 1 is executed by binding s to the (one and only) source stop of the current routing object depicted by the rfrom link. Then, we iteratively match variable m to departing moves that fulfill the time constraints given as boolean expressions in curly braces. This means the move's departure must lie between the earliest and latest times of the this object, i.e. of the router. For each such move we create a Hop object h and add it to the todo and the hops list of the current router. Finally, the totalArrival attribute of the new hop h is initialized to the latest departure time plus 120 minutes. This will restrict the depth of our routing search. We are allowed to do so, since Paderborn is a small town and bus rides taking more than 2 hours are not reasonable.

Sometimes one wants to execute additional operations on each iteration of an iterated story pattern that do not fit into a single story pattern. Therefore we provide a special transition labeled *[each time]* allowing to specify additional steps to be executed each time an iterated story pattern matches. When all matches have been considered execution follows an *[end]* transition. In our example, after each execution of Step 1, we follow the control flow arc labelled [each time] reaching Step 2. Step 2 calls method markFinalHop which is discussed below. When Step 1 and 2 have been executed for all possible object matches, the execution reaches pseudo Step 3.

Handling to-many associations is a special strength of story patterns. A single story pattern containing multiple to-many links generates a lot of navigational code consisting of several nested search loops. These search loops contain code that tests all other conditions of the story pattern such as arbitrary boolean constraints, attribute constraints, and additional required links. Using a plain story pattern the depicted effects of the story pattern are executed at most once for the first combination of objects that meets all constraints. Using an iterated story pattern the effects are executed on all valid combinations of objects. In addition an [each time] transition may be used to branch to additional substeps to be executed for each match of the story pattern. Altogether story pattern allow to search for quite complex object patterns and free the programmer from coding a lot of navigational code.

In addition to plain associations we also support sorted and ordered associations. Sorted associations organize neighbors according to their *compare* method. Ordered associations provide a user defined sequence of neighbors. In our example, the todo association is ordered, cf. Figure 6. Inserting links in

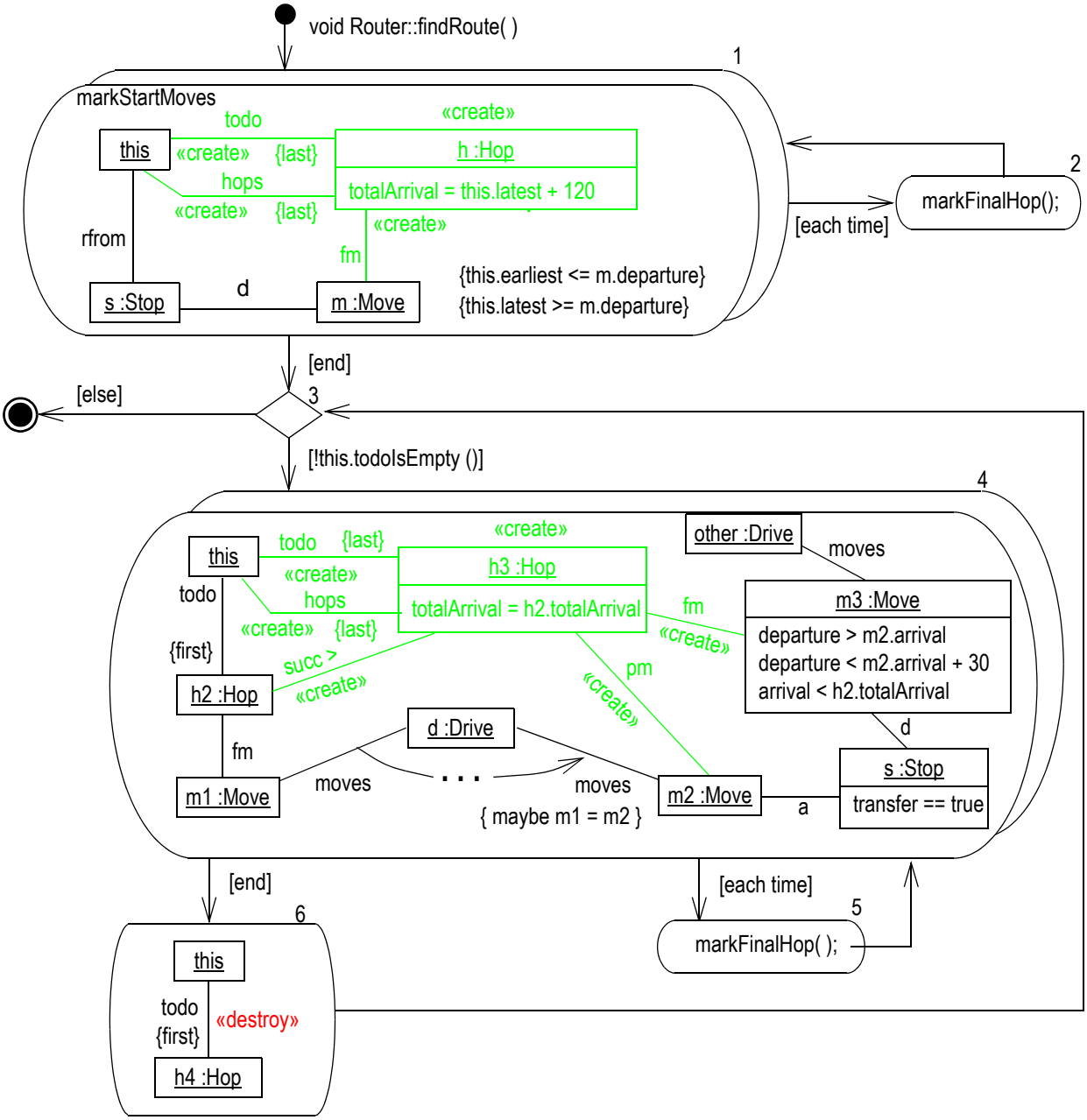


Figure 10 Story diagram Routing::findRoute

plain or sorted associations is shown by a normal «create» link (or a link attached to an added object). Insertion into an ordered list needs additional position information. Prepending or appending to the whole list is depicted using the constraints {first} and {last}, respectively. Appending is the default if no explicit position is given. In case of appending or prepending to another object, we use an arrow from the preceding to the succeeding link.

A similar notation is used for object look-up in ordered (and sorted) associations. For example note, the use of {first} and {last} constraints in Step 4 and 6. For look-up one may not only seek for the direct successor or predecessor within an ordered or sorted association but just for a later or earlier object. this may be specified using a "**. . .**" tag attached to the corresponding successor / predecessor arc, cf. the moves links from drive d to moves m1 and m2.

While iterated story patterns provide convenient means to loop through multiple matches, story diagrams provide pseudo activities and guarded transitions to model iterations and branching, explicitly. Pseudo activities are depicted as diamond shapes. They are just no-ops and used for branching only. Each activity (except iterated story patterns) may have an arbitrary number of outgoing transitions. In

case of multiple outgoing transitions one has to provide mutual exclusive guard conditions that determine the next execution step, uniquely. However, in general it is not possible to check statically whether the provided guards are actually mutual exclusive. Non-exclusive guards would be a source of undesired nondeterminism. We deal with this problem by two means. We introduce the special guard [else] which denotes transitions to be taken if no other guard is valid. In case of only two outgoing transitions this already resolves any nondeterminism. In case of more than two outgoing transitions we avoid nondeterminism by requiring a user defined order in which the guards are tested. This is indicated by ordering numbers subscribed to the guard condition, e.g. $[!\text{this.todo.empty}]_1$. If not necessary, the ordering number may be omitted.

In our example Step 3 to 5 form an explicit loop that terminates as soon as the todo list of the current router is empty. Step 4 implements the scenario described by snapshot 3 of Figure 4. The iterated story pattern looks for all reasonable possibilities to switch from one bus to another. It takes the first hop h2 from the todo list and determines its start move m1. The successor arc tagged with "**...**" shown below drive d specifies the transitive closure of a forward list traversal of d's moves starting at m1 reaching all successor moves m2. For each move m2 the story pattern checks whether arrival stop s is marked as a transfer stop. At such transfer stops we look for departing moves m3 that fulfill three attribute conditions. First, m3 has to depart after the arrival of m2. Second, m3 must depart within 30 minutes (waiting more than 30 minutes is considered to be inacceptable for bus passengers). Third, m3 should not exceed the maximum travel time stored in h2.totalArrival. For each such move m3 the iterated story pattern of Step 4 creates a new hop object h3 and adds it to the router. In addition a succ link from h2 to h3 is added in order to store the sequence of hops that builds a certain bus tour. The totalArrival time is forwarded from h2. Finally, moves m2 and m3 are marked by previous move (pm) and first move (fm) links. This information will be used to describe how to change busses when the query result page is computed.

Note, the default semantics of story patterns do NOT allow to bind two variables to the same object. Binding two variables to the same object would introduce the alias ban problem or violate the identification condition known from graph grammar theory, if one of the objects is deleted. On the other hand, occasionally it is elegant to allow such an alias ban. Thus, the designer may enable binding of multiple variables to the same object by enumerating the variables in a 'maybe' constraint (separated by '=' symbols).[1] In our example, we look for successors m2 of move m1 that reach a transfer stop s. However, move m1 itself may directly reach a transfer stop. Adding the constraint {maybe m1 = m2} allows to bind m1 and m2 to the same move in the object structure and thus handles this case.

Each time Step 4 is executed we also execute Step 5 calling markFinalHop which will be discussed below. When Step 4 terminates all successors of the first hop in the todo list have been considered. Then, Step 6 deletes this first entry of the todo list and execution reaches Step 3 again. Step 3 through 6 are repeated until the todo list is empty, i.e. until all routing alternatives within the given limits are considered. Note, we use the ordered todo list as a queue of hops to be considered. This results in a breadth-first-search routing algorithm. The routing is restricted by the maximal travel time of 2 hours.

To complete the description of our algorithm, we have a look at method markFinalHop shown in Figure 11. Step 1 checks whether the (just added) last hop of the todo list reaches the target stop of our ride depicted by the *rto* link. On success, the target move *m2* is marked with a *tm* link.

Step 2 is reached only if Step 1 was successful and if the new arrival time described by m2 is an improvement of the so far known arrival time given by h.totalArrival. In this case, Step 2 propagates the improved arrival time to all hops in the same hop tree that have a later totalArrival, thus pruning the depth of our search tree. Step 2 introduces a new kind of story pattern element a multi-object variable rep-

1. Unless the identification condition is violated.

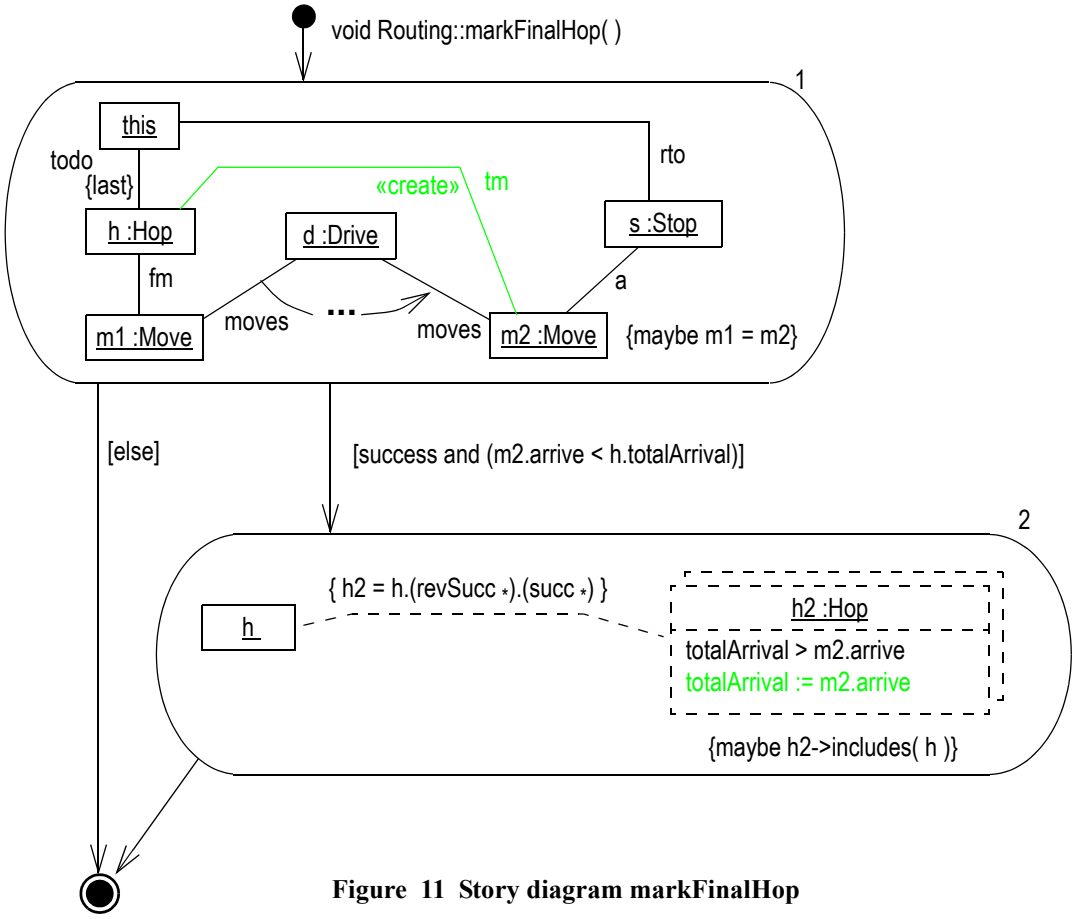


Figure 11 Story diagram markFinalHop

resented by two stacked and dashed boxes. Such variables are bound to all objects that fulfill the constraints associated to the multi-object variable. Multi-object variables conveniently allow to deal with sets of objects. In our case, h2 is bound to all hop objects that are reachable by building the transitive closure[1] (cf. the *) of *revSucc* links starting from hop h and building the transitive closure of traversing succ links from this intermediate set of hops. Note, revSucc is the implicit role name of the succ association traversed against its direction. Within the attribute compartment of h2 the totalArrival of all hop objects bound to h2 is set to the new arrival time m2.arrive.

The constraint {h2 = h.(revSucc *).(succ *)} is just an example for the frequent case that a story pattern needs not only to deal with neighbors reachable by direct links but that one wants to reach neighbors via several links. Therefore, we currently develop a sublanguage for story patterns providing convenient navigational expressions. This language is based on the object constraint language of UML. In addition we exploit our experiences with the path expression sublanguage of Progres, cf. [Sch91, Zü96], that for example provides transitive closure operators.

To conclude the description of story diagrams, we describe the remaining features of story patterns, that proved to be useful but are not used within our bus route example. Boxes crossed out by a bold X represent unbound variables that are quantified by a "not exists" quantifier, i.e. it must not be possible to bind an object to such a variable. Normal unbound variables are implicitly quantified by an exists quantifier. Accordingly, links crossed out by a bold X represent the condition that the corresponding objects must not be connected by such an link. Dashed boxes represent optional unbound variables. Such variables are bound to objects if possible. They are dropped otherwise. Note, it is not allowed to use more than one non-normal variable within one constraint since this leads to undefined or confusing semantics.

1. Note, the transitive closure operation is one of our extensions of the UML object constraint language stemming from PROGRES navigation (path) expressions, cf. [Sch91b].

Please note, we restrict the control flow graphs built by story diagrams to so called 'well-formed' diagrams which directly correspond to nested control structures of standard imperative programming languages. This allows the generation of usual Java code from story diagrams. Note, sometimes a certain activity within a story diagram performs complex (e.g. mathematical) computations only. In such cases it might be more appropriate to use normal Java code instead of a story pattern. Thus, story diagrams (as well as story charts) allow to specify the body of an activity with Java, too.

Story diagrams allow to model the logical functionality of the desired system at a high level of abstraction. Story diagrams provide a formal yet readable and executable specification. They free developers from the error-prone and tedious task of manual coding in a plain programming language. In addition, story diagrams use a notation similar to the story boards employed in activity 2 and 4 for requirements specification and analysis. This narrows the semantical gap between the languages employed in the earlier phases of system development and the languages used for realization. Note for example the similarities of the snap shot in step 3 of Figure 4 and of the story pattern in step 4 of Figure 10.

Activity 5 1/2: Intermediate step automatic code generation

Activities 3 to 5 result in a complete operational specification of the desired system consisting of detailed class diagrams, story charts, and story diagrams. The class diagrams define the static program structure. Story charts specify the behavior of active objects and story diagrams specify methods employed as basic actions in story charts and methods used by such actions. From this complete specification the code generator of the Fujaba environment generates fully automatically a Java implementation.

Code generation for class diagrams is quite well understood and provided by most available UML CASE tools. However, most tools create very poor implementations for associations. UML associations are bi-directional. Thus, UML associations should be implemented by pairs of references in the respective classes. In order to guarantee the consistency of the pairs of references that implement an association, on each change of one reference the reverse reference must be updated, too. This can be achieved automatically by encapsulating the write-access to such references and by extending the corresponding write-methods to call each other. This approach is realized by the Rhapsody tool [Rhap] and by our Fujaba environment. The rational rose environment employs pairs of references, too. However the access methods for the pairs of references do not call each other. This burdens the client programmer with the responsibility of keeping the pairs of references consistent.

Method declarations in class diagrams lead directly to the corresponding Java method declarations. A *story diagram* is translated into an Java implementation of the corresponding method. The translation of the *control flow* part of a story diagram to Java code requires to uncover the nested block structure of the depicted control flow and to find the contained loops and branches. Once these building blocks are identified the translation to Java is straight forward, cf. [Klein99]. *The translation of story patterns* contained in story diagrams splits into two parts. First, we must match all the unbound variables of the pattern, second we perform the described effects. The latter step is relatively straight forward, the code generated from the class diagrams provides appropriate methods for creations and deletions of objects or links and for attribute assignments. The matching of unbound variables corresponds to query optimization problems in relational database systems. The depicted object structure corresponds to a query, describing which objects are to be retrieved but not how to find them. The task is to generate all navigational collaboration messages that compute the neighbor objects participating in a collaboration and to generate the code necessary to check all depicted constraints and conditions. This topic is discussed in chapter 3.3.

Code generation for statecharts is also quite well understood. For the statecharts part of story charts we employ a table driven implementation approach, cf. [KNNZ99, Doug98]. The states and transitions are represented by a dedicated object structure that is interpreted at runtime by a small state table interpreter provided by the Fujaba runtime library.

The complete SDM specification we developed for the BusRoute example comprises 15 pages. The generated code for the logical classes corresponding to that SDM specification finally comprised about 4750 LOC. In addition, we developed about 5 pages html code as part of the user interface using a conventional WYSIWYG html editor.

Activity 6: Validating the Model

Once code has been generated and compiled, only a user interface is missing in order to create a first prototype of the desired application. Usually one may use a modern GUI builder for this purpose. Our example application BusRoute is designed to be used via WWW. Thus, we used a standard WYSIWYG html editor to create a start up page and a small WWW browser contained in the Java runtime package to create a first prototype user interface for the BusRoute system, cf. Figure 12. In addition we had to start the BusRoute server process and provide it with some test data on bus stops and drives.

Selecting hyperlink "Query" on the left column of Figure 12 issues a query to our BusRoute server, The BusRoute server returns the html-page shown in the lower right frame of Figure 12. This frame contains a html query form allowing to select start and target stops and the desired time range from pull-down selection boxes. Pushing the "Submit" button issues the query and an appropriate event is send to the BusRoute server. As discussed above, the BusRoute server will create a Routing object running its own thread. This routing thread computes possible routes and derives an html answer page that is send back to the querying browser. Figure 13 shows our WWW browser depicting an example html answer page.

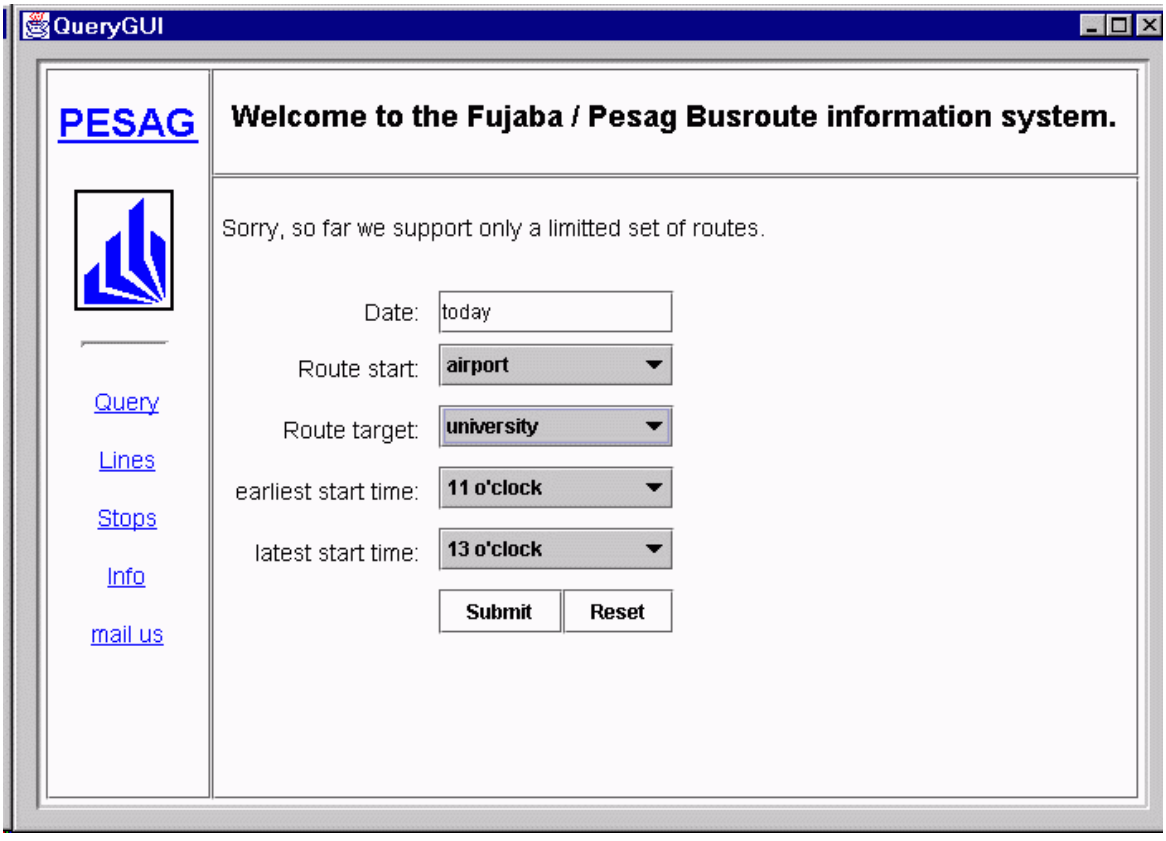


Figure 12 WWW Browser showing the Query page received from the prototype BusRoute server

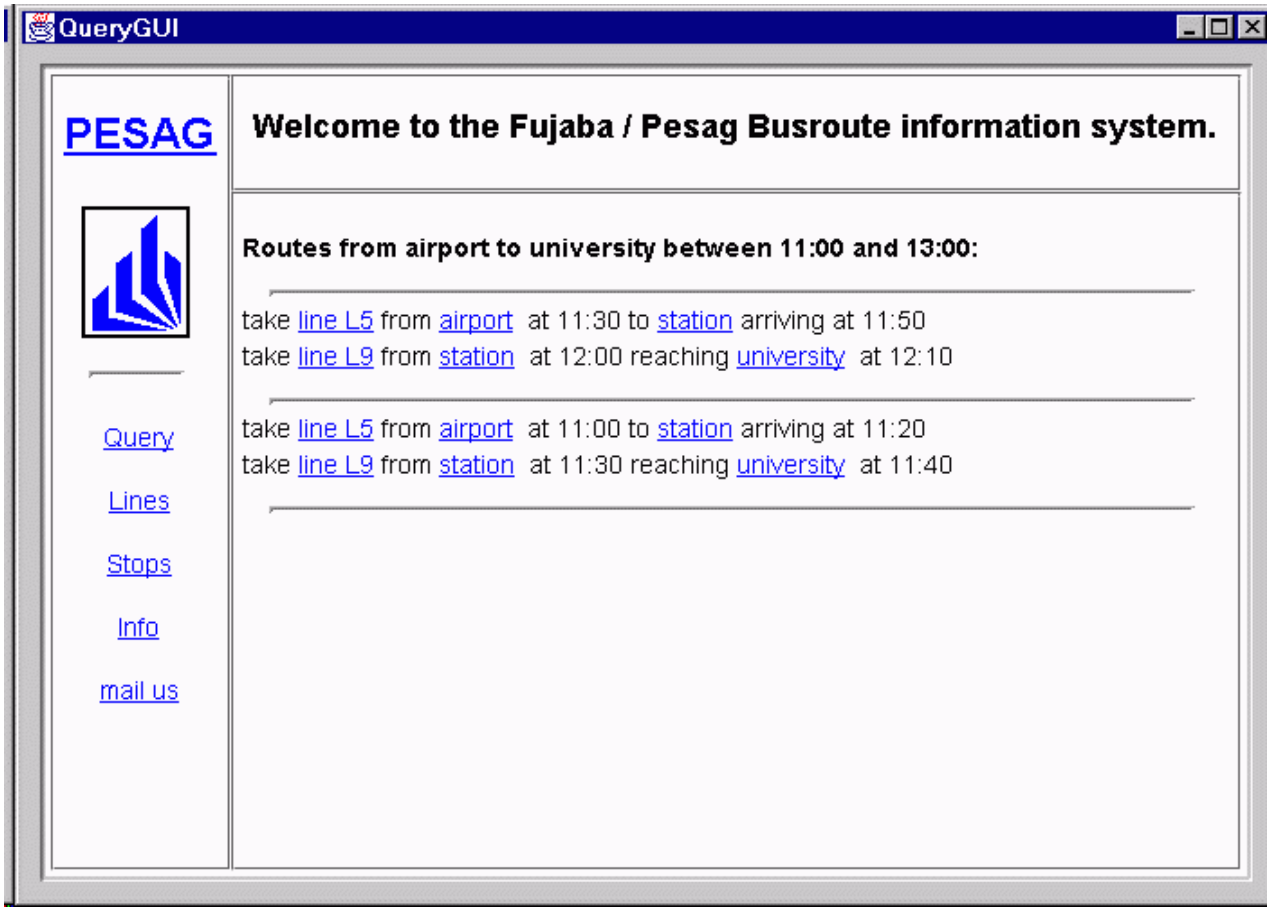


Figure 13 WWW-Browser showing the query result

Story diagrams and story charts provided a quite high level of abstraction for the specification of our example routing algorithm. This high level of abstraction facilitate the modeling of complex object structures and of the corresponding functionality significantly compared to using plain programming languages, only. However, frequently the specification we still contain errors causing undesired or unexpected runtime behavior. Validation and debugging will be necessary.

One may write test routines for this purpose and use a usual debugger to step through the execution of application code in order to determine bugs and to fix them. Unfortunately, usual debuggers provide only limited means for the investigation of complex runtime object structures. Only some sophisticated debuggers like XDDD provide a graphical view of the runtime object structures. However, this graphical views are still in-appropriate for complex graph-like object structures. One main problem is that to-many associations are implemented using some kind of container classes. Usual debuggers show the internal structures of these container classes, e.g. internal nodes employed in a sorted tree. Instead of showing the set of neighbor objects attached to a given object, directly, a usual graphical debugger like xddd would show a set object containing references to a tree of inner nodes and these inner nodes contain references to the neighbor objects of interest.

Therefore, we have developed a dedicated graphical browser called Dobs (Dynamic Object Browsing System), cf. Figure 14. The Dobs browser employs a table of predefined container classes. The Dobs browser uses the Java runtime type information api java.lang.reflect to examine runtime objects. For each object a box is shown containing an internal id and the type of the object and optionally all its basic attributes together with their current values.

If an object contains a reference to another object, the neighbor objects are shown on demand, connected by a simple link optionally showing role names. If an object contains a reference to a container

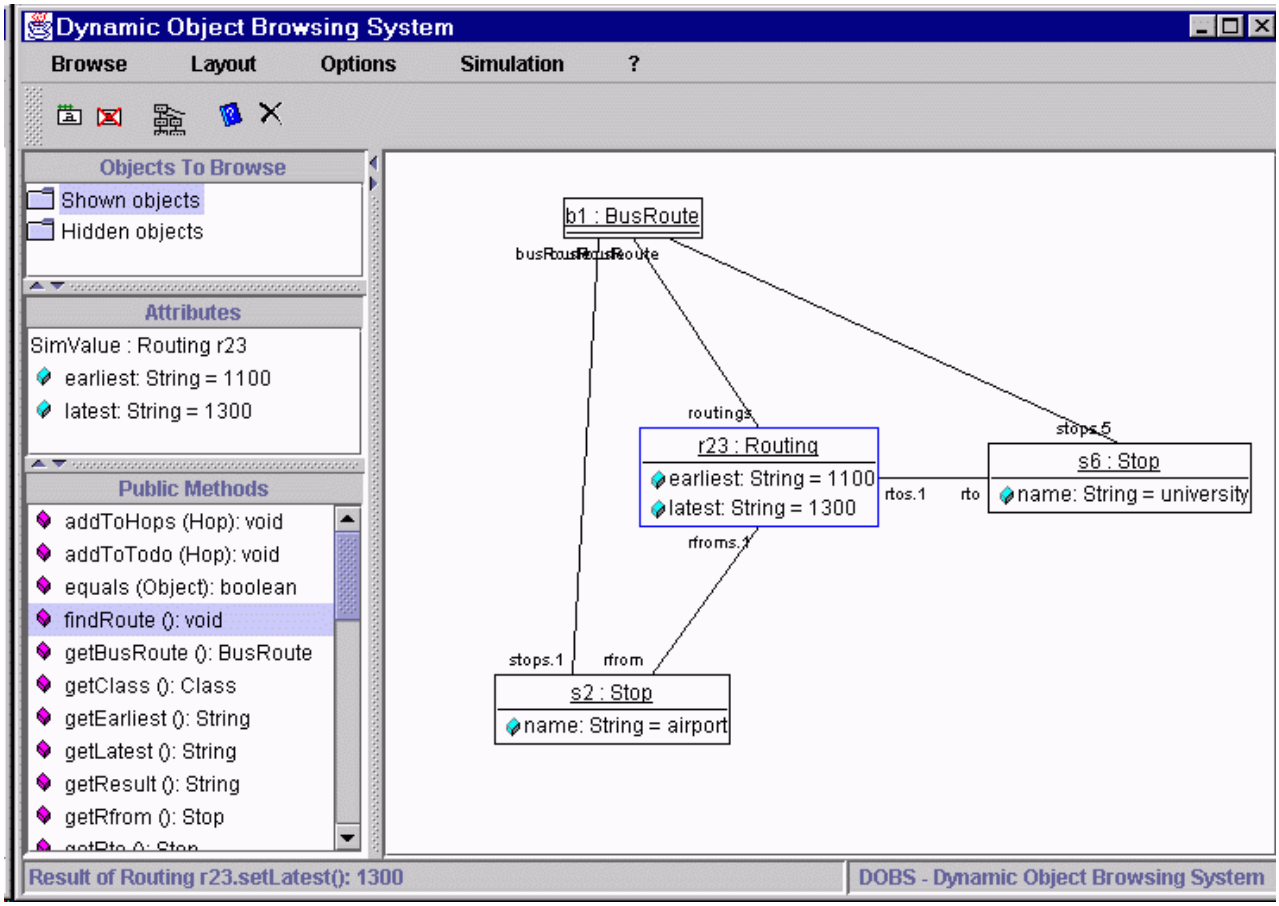


Figure 14 The Dobs browser showing a cut-out of the routing object structure

object, the container is investigated and the contained objects are retrieved. Instead of showing the inner structure of the container, only the contained objects are depicted and directly linked to the start object. In Figure 14, the BusRoute object b1 contains two such containers of references. A set of stops and a set of routings. The Dobs browser just shows the contained stops s2 and s6 and the contained routing r23. For the stop objects the name attribute is depicted and for the routing object attributes earliest and latest.

The Dobs browser not only provides a sophisticated view of complex graph-like object structures. Using the java.lang.reflect programming api, the Dobs browser allows to call methods on selected objects, interactively. In Figure 14, the routing object r23 is currently selected. In the lower compartment of the left column, the Dobs browser depicts all methods of the selected objects. One may invoke any method interactively. For methods requiring parameters a generic form pops up, allowing to provide parameter values. Then, the method is called and one may for example step through it using a conventional debugger. Once the called method returns (or intermediately on demand) the Dobs browser analyses the depicted object structure again and shows the new content.

In our example we issue method findRoute on the current router object. Figure 16 shows a detail of the resulting object structure. The routing has created 4 hop objects. For hop object h24 and h26 the neighbor objects are shown. Move m9 is the first move of hop h24 departing from stop s2, the airport, and arriving at s5, the station. Hop h24 has one successor, hop h26. Move m14 attached to hop h26 reaches stop s6, the route target. Note, the BusRoute object and the neighbors of hops h25 and h27 are hidden to simplify the picture. One may compare this runtime object structure to the initial story board scenarios, e.g. in Figure 4.

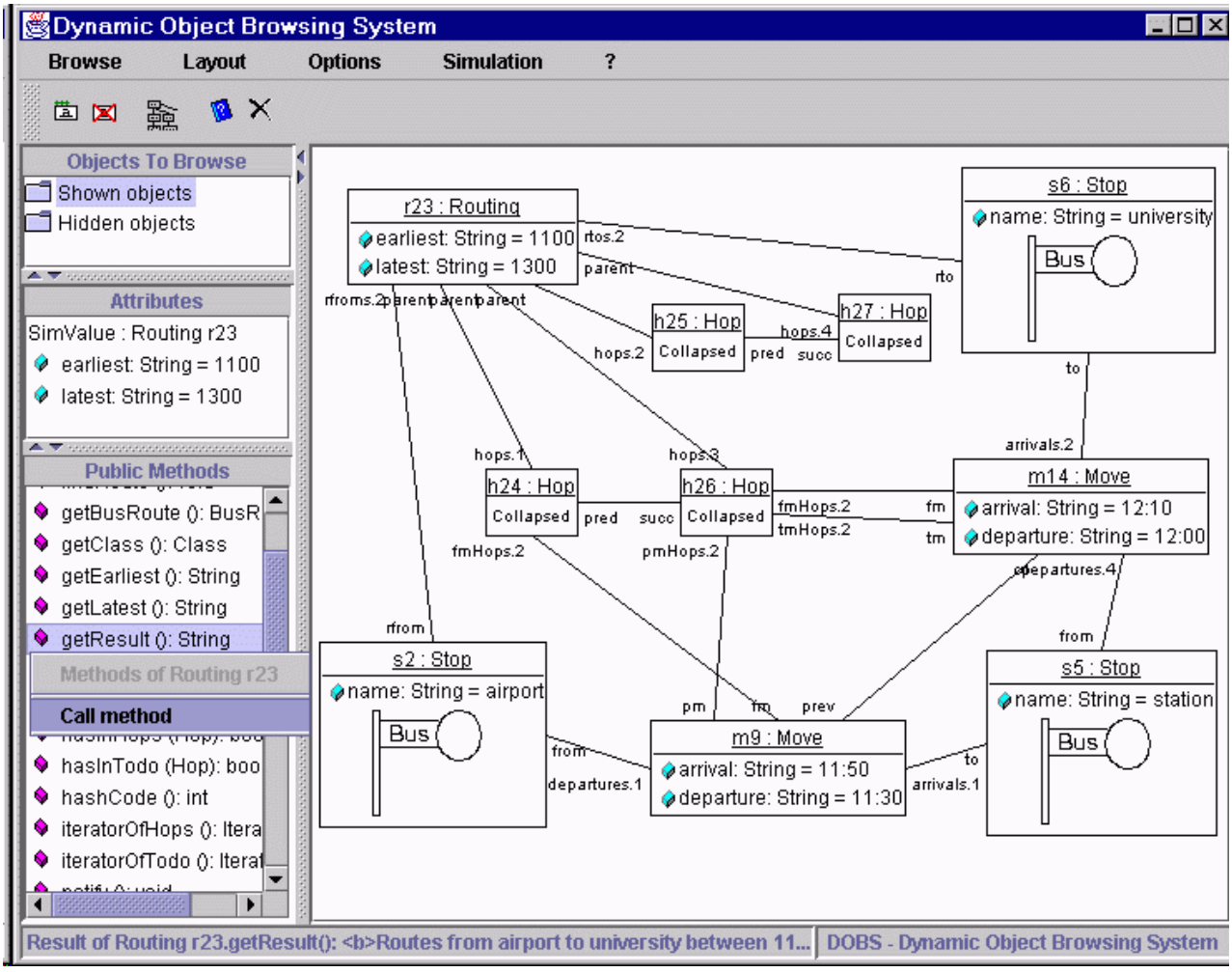


Figure 15 The Dobs browser showing the effects of calling method findRoute

The dobs browser may be adapted in several ways. First of all the table of pre-defined container classes may be extended to cover other container libraries. Certain links and certain kinds of objects may be excluded from depiction. The cut-out of depicted objects may be chosen manually. The appearance of depicted objects may be adapted for individual objects or at type level. Only selected attributes may be shown or the attribute compartment may be omitted at all. Different icons may be attached to different objects depending on their type and internal state, i.e. attribute values. If an icon is provided the object id compartment may be omitted. Objects may provide a method computing their xy-position at the screen thereby providing an application specific layout. The method invocation column may be hidden. In future, one will be able to attach gui-elements like buttons and menus to different objects,. In Figure 16 we assigned a map of Paderborn's bus lines as gif for the BusRoute object. In addition Figure 16 shows some bus objects. These bus objects position themself at those bus stops that reflect their current position (according to their schedule). Such a view could become an alternative user interface for our BusRoute system.

Such a prototype may be shown to a customer as proof of concepts. For the BusRoute system, the representative of the civil services office was an electrical engineer, who had no problems in catching the ideas outlined by story boards and specified by story diagrams. He got a demonstration of the BusRoute prototype and followed the execution of the routing algorithm for some critical example queries, thoroughly. Finally, he was convinced that our solution overcomes the problems of the former commercial busroute system and we got the project to develop the final system.

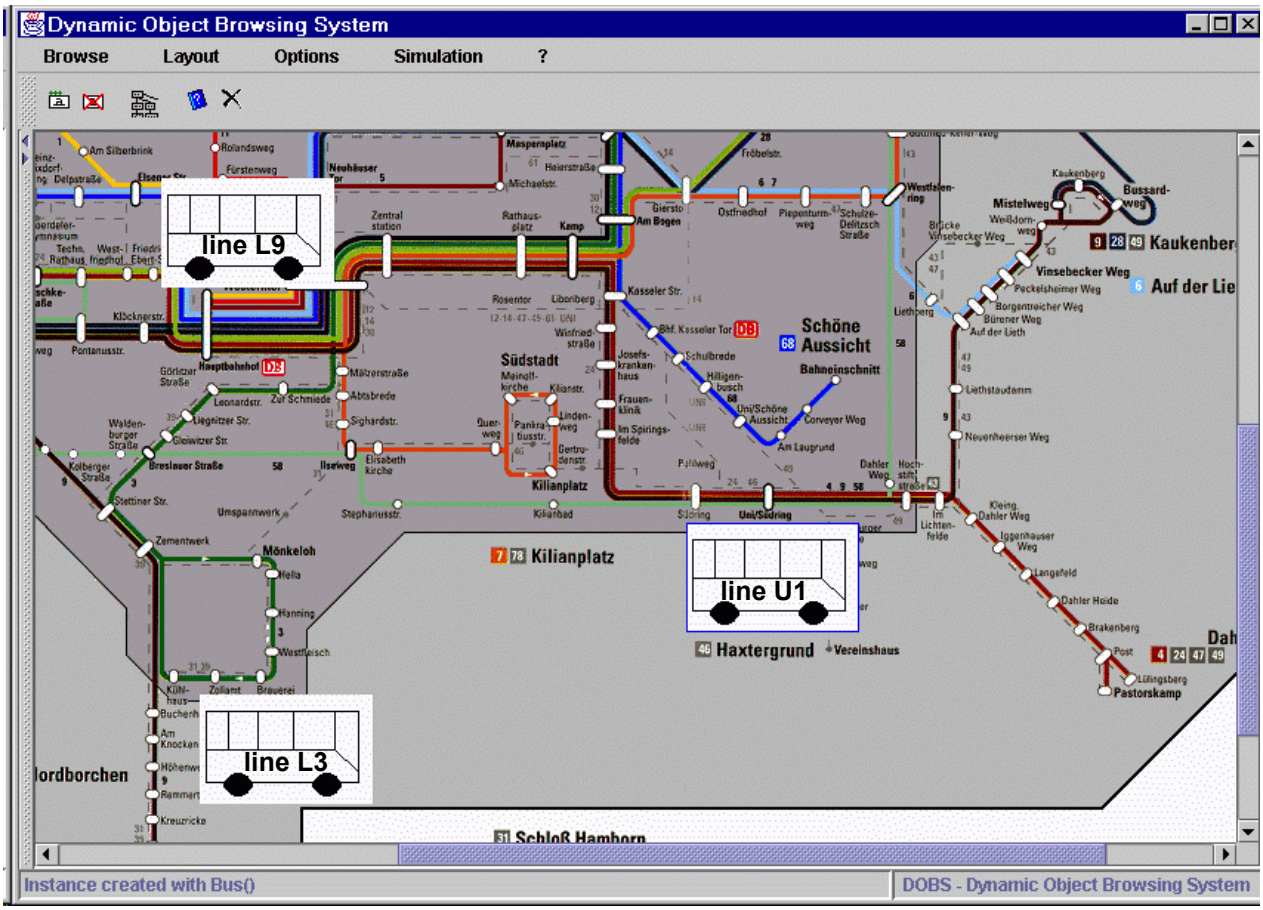


Figure 16 The Dobs browser adapted as a map view showing current bus positions

2.4 Conclusion

Current object-oriented software development methods focus on project organization issues, cf. the Unified Process [JBR99]. They do not provide concrete help on how use-case are refined by scenarios and how class diagrams may be derived from scenarios and how scenarios may be turned into state-charts and method body specifications and how a reliable and robust implementation of such a specification is achieved. Story driven modeling is a possibility to complement existing approaches and, thus, to overcome these deficiencies. We propose story boarding as a central activity to analyse the dynamics of object structures. From story boards, class diagrams are derived semi-automatically. Together, story boards and class diagrams build a domain model. This domain model is then refined to a conceptual model by developing algorithmic ideas and by corresponding extensions of story boards and class diagrams. In the next activity, the ideas outlined in the conceptual model are formally specified using story charts and story diagrams, yielding an executable specification. This formal, executable specification enables code generation, using the Fujaba environment. The generated and compiled code may than be validated and debugged using the interactive Dobs browser. The Dobs browser may even serve as an initial interface of a first functional prototype. For the final system, mainly user interface parts need to be added.

With the generation of usual Java code that seamlessly fits in standard architectures, SDM overcomes a major deficiency of its predecessor PROGRES. PROGRES relies on a proprietary, non-standard database system, that so far prevented its wide industrial use. In addition, we enhanced the graph rewrite rule notation to become more compact and we adopted a lot of standard object-oriented

method elements (especially from UML). This facilitates both, to read SDM specifications and to learn writing them.

Usage of SDM and its predecessor PROGRES started in 1988 at Aachen University with case tool development. Two of the most recent systems developed this way are the DYNAMITE process modeling system [HJKW96] and the cobol reverse engineering environment [Cre00] and the VARLET database system re-engineering environment, cf. [JSZ96]. Today, SDM and PROGRES are used at about 20 University sites and recently we were successful to attract the first industrial users. SDM and PROGRES have been taught with great success in yearly courses to students at Aachen and Paderborn University since 1992. In addition, we applied the method to several industrial research projects. The BusRoute system proposed in this paper, was actually sold to an industrial customer (with profit).

SDM is based on a rigorous usage of the UML. Each diagram models a certain aspect of the overall systems. All these diagrams must form a complete and consistent specification of the whole application. This enables fully automatic code generation for complete applications including all method implementations. These code generation concepts are discussed in the next chapter.

3 From Design to Code

This chapter discusses our code generation concepts for UML diagrams. Our approach uses Java as implementation language. We have chosen Java as the implementation language for several reasons. We wanted a well accepted object oriented programming language. It should be well accepted in order to reach a broad audience. It should be object oriented since story driven modelling exploits object structures intensively. This reduced the candidates to C++ and Java. C++ is even more widely known as Java. In addition, C++ is better standardized and has better tool support. Still C++ programs are somehow faster then Java programs, this holds especially for file access and string handling. However, the platform independency of Java attracted us. In addition, Java is much simpler than C++. The Java syntax is much simpler which facilitates e.g. reverse engineering, i.e. parsing, dramatically. And the garbage collection features reduced the memory management failures of our student programmers by orders of magnitudes.

In principle, architectures and designs should be independent from the programming language that will be used to implement the desired system. One should do his design first and based on this design and other requirements one should choose the most suitable programming language for the realization of the system. However, due to our experiences, a detailed, language independent design does not work. The properties of the target language influence a detailed design in many ways. For example, usually one will use the names of design artifacts within the generated code. Thus, the design names should be valid identifiers in the chosen implementation language, e.g. they should not contain blanks or semicolons. Different programming languages have different restrictions on valid identifiers. Similarly, different programming languages may employ different parameter passing concepts. For example Java provides only in parameter and no in-out parameters. This restricts method signatures in the class diagrams, accordingly. Another example is that Java does not support multiple inheritance for full classes. If the design uses multiple inheritance for full classes then code generation for Java runs into problems (this will be discussed in the corresponding chapter). If we use elder versions of Fortran or Basic or Pascal or Cobol or C as programming language, then inheritance is not supported at all and using inheritance at the design level increases the gap between design and coding. Next, elder versions of Fortran or Basic do not provide dynamic memory management features. This automatically restricts the design to imperative concepts. Object oriented languages like C++ provide memory management concepts however they may not provide garbage collection. Without garbage collection in the programming language, the design has to provide strategies for object allocation, deallocation, copying, and ownership. With garbage collection in the target language, these topics require less design efforts. To summarize, various properties of the chosen implementation language will affect various aspects of a detailed design. There is no language independent design. The target language influences a detailed design in various ways.

One of our design goals for code generation from UML diagrams is to generate high quality, human readable code. Our future goal is that UML becomes a general purpose visual programming language for complex object oriented applications and that the generated code is some kind of intermediate code that is used by subsequent, fully automatic back-end post-processors, only. If this is achieved, the code no longer needs to be human readable, only back-end programs would deal with it. However, during the next decades, we expect that many developers will have to read, debug, modify, or extend the generated code, manually, during further system development. At the coding level there exist sophisticated editors, compilers, debuggers, cross-referencing tools, documentation generators, aspect weavers, search facilities, slicing tools, delta computation and patch tools, version and configuration management tools, test automation and coverage tools, metric tools, performance analysis tools, etc. In addition, other code generating tools like e.g. GUI builders may be used and such code needs to be integrated with the code generated from the UML specification. Until all these function-

alities are provided or integrated at the UML level, too, the developers will still have to use some of these code level tools to fulfill their tasks. Using such a tool at the code level implies reading the code e.g. during code level debugging sessions. Thus the generated code should be human readable and it should be easy to map UML design elements to their implementation and vice versa.

Note, the user may not only read the generated code but there is also a high likelihood that the user or some code based tool will also (have to) change the generated code (manually). For example, during a code level debug session the user probably wants to fix some minor bugs immediately within the code. Some GUI builder generated code may be integrated, manually. An aspect weaver may add some code to some methods. A merge of concurrently created versions of some files may change the generated code or may require manual conflict resolution steps. Some of these changes may be design relevant. For example if the user extends or modifies the signature of some methods or if the user adds new attributes or methods. In our approach, the UML specifications covers also method bodies. Thus, in our approach even changes to the method bodies are design relevant changes. Due to such changes the UML design and the corresponding code may easily decouple from each other and after some weeks or months the design does not longer describe the implementation and the design becomes useless. To avoid this decoupling of code and design, our approach tries to provide round-trip engineering support. This means, our Fujaba environment is able to analyse manually modified or even manually created code and to create or update the corresponding UML design elements. However, this works only as long as there exists an (almost) bidirectional mapping of UML design elements and the corresponding code fragments or code clichés. Thus, our approach does not only try to generate human readable code but also code that is easily reverse engineered. For more information on our round-trip engineering concepts cf. [Klein99, KNNZ00b, NWZ01].

Appendix A provides a formal description of the semantics of UML diagrams within our approach. Contrarily, the current chapter gives a very informal description of our code generation concepts. The code generation could easily be described in a formal way, too. Our formal notion of UML story diagrams could be used to specify the code generation for the different UML diagram elements. In [Zü96] we have done this for the language Progres, a predecessor of this work. Due to our experiences with [Zü96], such a formal specification of the code generation concepts is hard to understand. One concentrates on understanding the correctness of the code generation mechanisms. Large parts of the code generation mechanism deal with symbol table creation and information retrieval and with gluing small fragments of source code together. Dealing with these details, it is not easy to understand the general code generation concept and to understand the mapping of UML elements to source code.

From a scientific point of view, a formal definition of the code generator could allow to prove the correctness of the code generator. If the target language has a formal semantics, too, then one could prove the equivalence of the original UML specification and of the generated implementation. Unfortunately, we do not yet have a formal semantics for our target language Java and thus such a correctness prove is not yet feasible.

Thus, the current chapter describes our code generation concepts in an informal manner. We will discuss the elements of UML that are used in story driven modeling using typical examples and for each such example we discuss the corresponding code fragments. This is accompanied with additional discussions of alternative code generation concepts and of design trade-offs. We hope that this approach provides an understandable yet precise description of the mapping of design elements to source code.

In the following discussion, we will have to distinguish between design elements and implementation elements. Frequently the design element and the implementation element will have a similar name, e.g. class, attribute, or method. If necessary, we will denote design elements as UML elements, e.g. UML class, and implementation elements as Java elements, e.g. Java class. We use the keyword Java to denote implementation elements because we target code generation for Java.

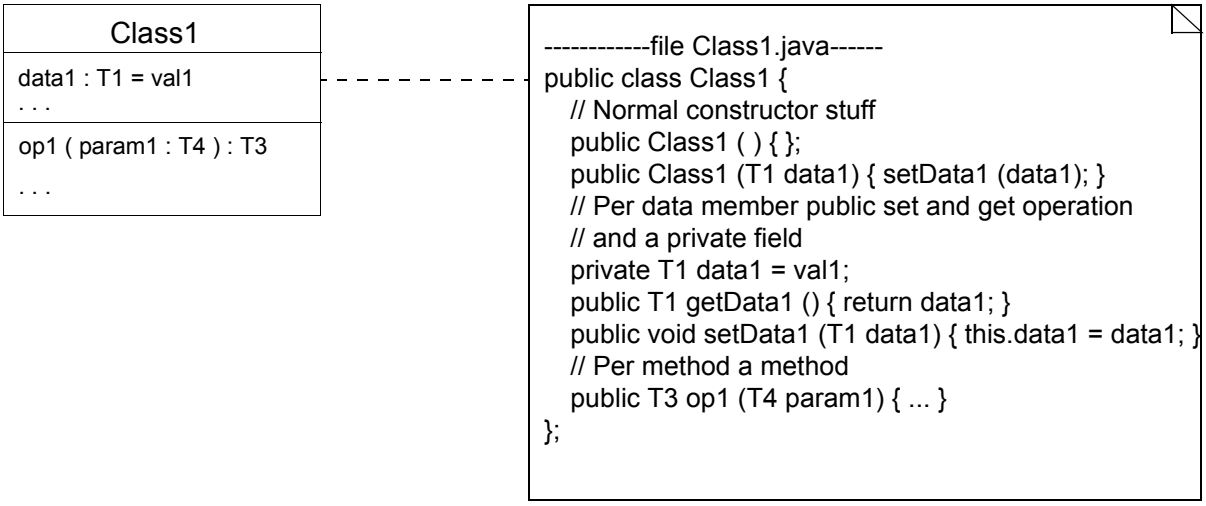
3.1 Class diagrams

Code generation for class diagrams is well understood and provided by most existing CASE tools. However, there are still some technical details that need to be discussed. Especially, the implementation of associations inhibits tricky aspects and design trade-offs that need a careful handling.

Classes, attributes, and method declarations:

Usually an UML class is implemented by a Java class with the name. However, sometimes a UML class that e.g. serves as a communication interface between different processes on different machines may require an implementation involving several Java classes and additional library functions. In such cases we employ special stereotypes to flag special code generation requirements. Thus, a plain UML class corresponds to a Java class with the same name.

Example 3.1; Java code for an UML class



Accordingly, not every Java class corresponds to an UML class. Usually a program employs a large number of minor helper classes that are no design issues. For example, in Java one often employs little classes like Comparator or MouseListener, that are just used to pass a single method or a little code fragment as a parameter to some operation or to store such items in some data structures. Such artificial helper classes are usually omitted from an UML class diagram.

In Example 3.1 the Java class Class1 gets public visibility. If a class or an attribute or a method is depicted in a class diagram, explicitly, it represents an important design artefact. Thus, if not stated otherwise, the default visibility of the corresponding Java element should be public.

Any class should have certain *default* constructors. In our approach we always employ a default constructor without parameters. Such a default constructor allows to create a well initialized object without the need of any additional input data. If an object provides only constructors that require input parameters their may exist situations or program parts where one would like to create such an object but he is not able to provide meaningful values for the constructor parameters. Instead he might want to pass the new object to some other routines that have the task to fill the object attributes. Thus, classes with parameter less constructors offer more flexibility for their usage.

However, sometimes one has all data at hand that is required to initialize an object. Having only a parameter less constructor one would first have to create the object and then one would have to fill the attributes. For such situations all classes should provide a so-called *comfort constructor*. A comfort constructor has parameters for all attributes of the corresponding class. Thus, a comfort constructor allows to create an object and to assign values to all its attributes within a single statement. This

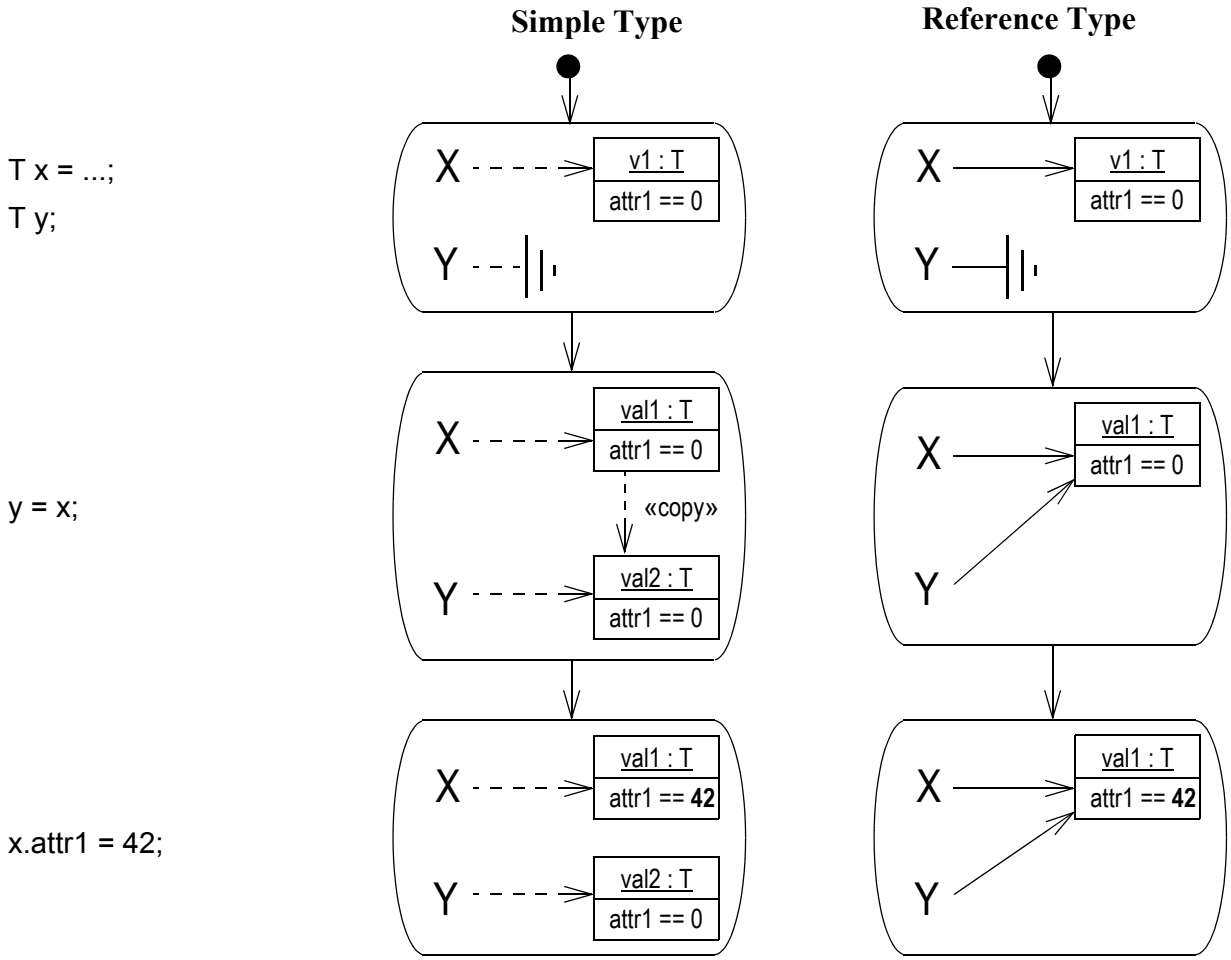
is especially helpful during maintenance. If the class is extended by an additional attribute then one has to visit all places where such objects are created in order to ensure that the new attribute is initialized, properly. In a large program it may be hard to find all such places. If we have always used a comfort constructor, then we extend the declaration of the comfort constructor with an additional parameter. If we now run the compiler, it will issue error message for all places where the old version of the comfort constructor is used, since a parameter is missing in these calls.

In addition to the default parameter less constructor and the comfort-constructor one may introduce some intermediate constructors that provide common initialization parameters, only, or that e.g. allow the initialization via a string parameter.

Note, constructors are standard elements of classes. Each class has these constructors. Thus, in the UML class diagram there is no need to mention standard constructors, explicitly. Instead, standard constructors should be omitted from the class diagram in order to keep the class diagram simple.

Attributes in the class diagram become Java attributes. Note, attributes must use "simple" types like bool and int, only. Basically, simple types are types that are passed as values and not as references. Types that are passed as reference could create the so-called alias-ban problem. Consider the following example:

Example 3.2: Simple versus reference types

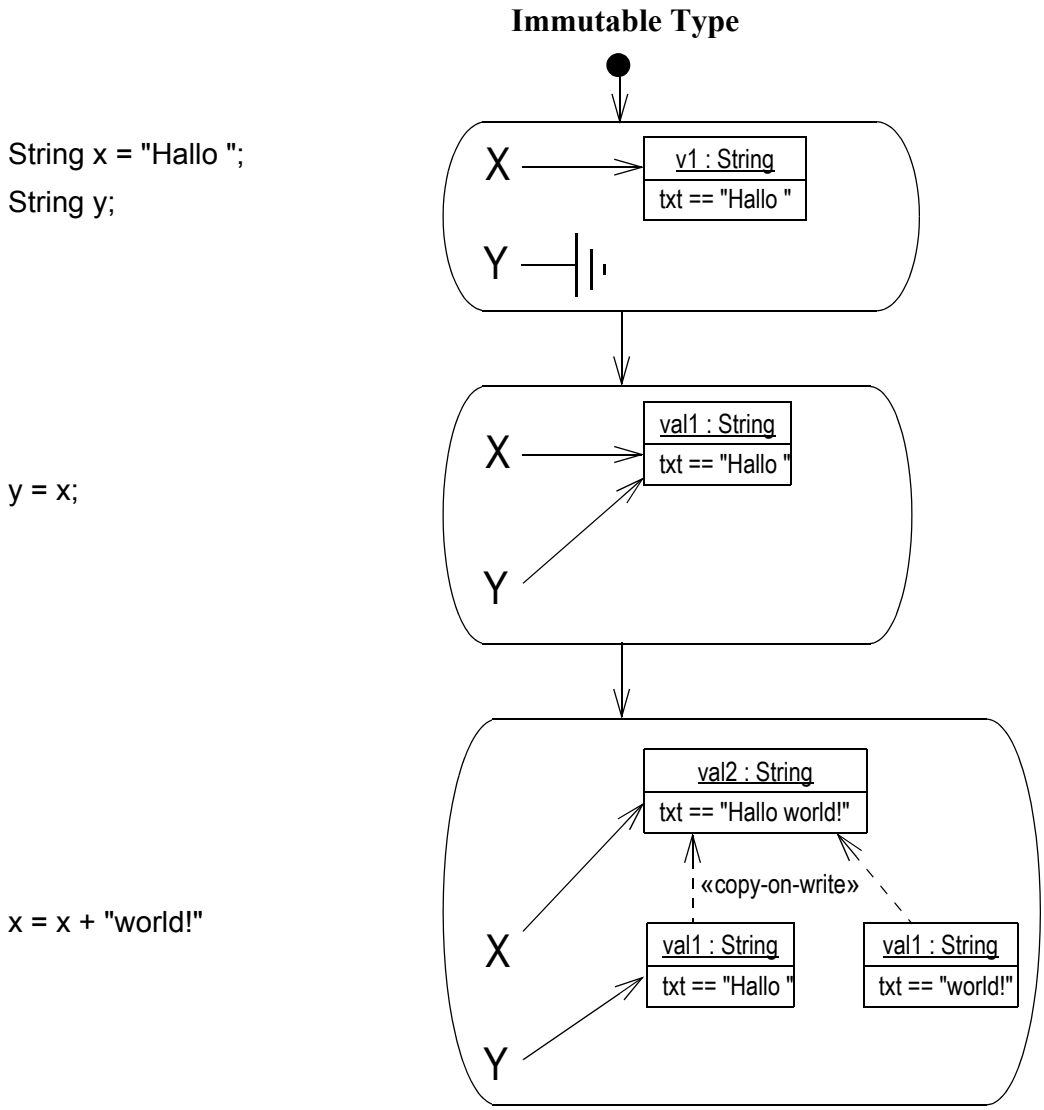


In Example 3.2 variable x is assigned to variable y. If variable x holds a reference to some object then the assignment y = x copies only this reference from x to y. The referred object is not copied. Thus, variables x and y now refer to the same object. If the object is changed via one of the variables then this change affects the other variable, too. If variable x holds only a "simple" value then the assign-

ment y = x copies this value. If the content of variable x is modified the copy contained in variable y is not affected. This difference between simple types and reference type is very important and must be modeled at the design level, explicitly. Mixing such aspects is a typical source of hard to find bugs and a frequent source of maintenance problems. To distinguish simple types and reference types at the design level, strictly, in our approach attributes must have simple types, only. If a type has reference behavior one must introduce a class in the class diagram and use an association in order to model such a data value.

In Java there are only few pre-defined simple types. Frequently, one may want to introduce an application specific simple type, e.g. type Date with attributes day, month, and year. In Java the only way to introduce a new user defined type is to introduce a new class. However, Java classes are always reference types. If we introduce a new class Date and if a method employs two variables x and y of type Date then an assignment y = x copies the reference, not the value. To achieve something like a simple type, in Java one uses so-called *immutable* classes. A built-in example for such an immutable class is the Java class String. A Java String is a complex object and a variable x of type String actually holds a references to String objects and an assignment y = x results in two references to the same String object. However, the alias-ban problem described above cannot occur for Java String objects. In Java, class String offers read methods, only. One may compare a String object with another String object or search for a substring or access a certain character. However, it is not possible to change a String object. There is no method to change a character within a given string or to extend a given string. Instead, if one wants e.g. to concatenate two strings this does not append the second string to the first string but it creates a new string that contains the concatenation of the two previous strings, cf. Example 3.3. This behavior is called *copy-on-write*. Instead of changing an existing value, a copy is created and the changes are applied to the copy, only.

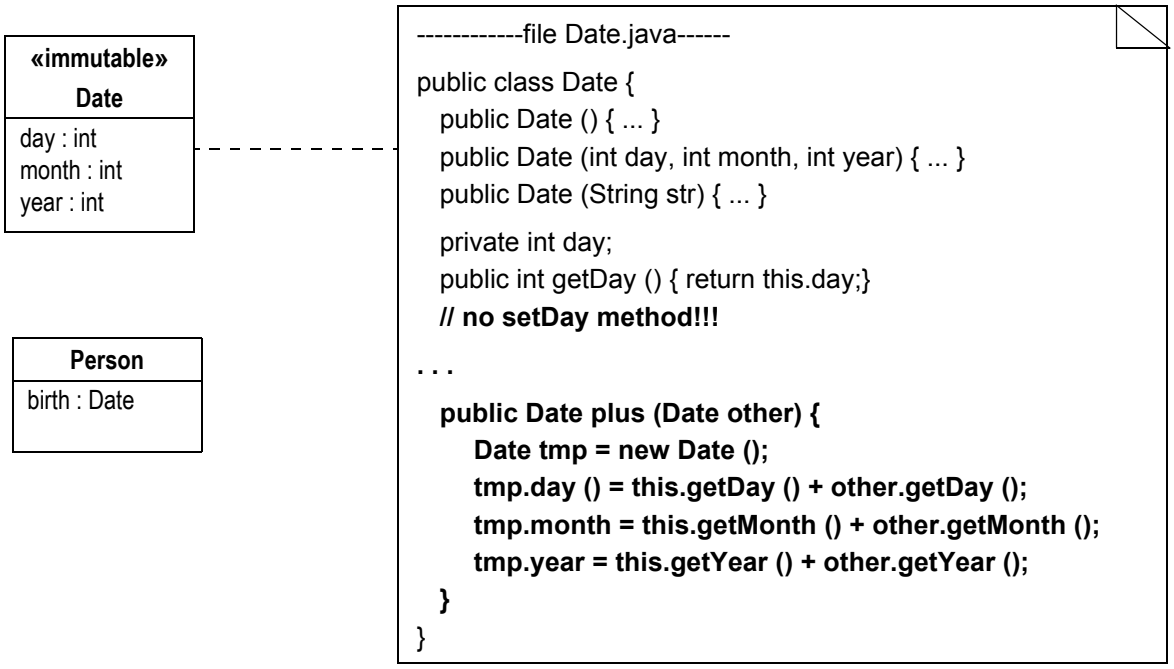
Example 3.3: Immutable types



This copy-on-write behavior avoids the alias-ban problem of reference types since the object referred by variable y remains unchanged.. Although an assignment creates a second reference to the same object, this does not harm because the referred object is never changed. Note, the problem described in Example 3.2 is created by the change of an object that has multiple references. Immutable types avoid this problem by just not changing existing objects. Instead, any modifying operation m creates a copy of the old object, applies the changes to that copy and returns the new object as result. Instead of x.m() one just writes x = x.m (). This achieves that other references, e.g. y, are not affected by m.

In Java such immutable types allow users to create their own "simple" types. Example 3.4 shows the user defined immutable type Date. Class Date has three attributes day, month, and year. In the Java implementation these attributes are private and only a get method is supplied. The modifying operation plus creates a new Date object tmp and applies the modifications to this new object tmp. The resulting object tmp is then returned. In our approach such immutable types may be used like simple types, i.e. immutable types are allowed as attribute types. One would not use an association to an immutable type. Immutable types are usually omitted in class diagrams.

Example 3.4: Immutable type Date



Our code generation for UML attributes follows the Java Beans coding conventions. This means, a UML attribute data1 becomes a private Java attribute with appropriate get and set methods getData1 and setData1, respectively. This encapsulation of attributes by access methods is best software engineering practice. It allows to control the access to the attribute and to establish side effects very easily. For example, the set method may control whether the new value for an attribute fulfills certain consistency constraints, e.g. that the value for a birthDate field in type Person should not be in the future. Or the set method may notify certain observers about the value change. Such a notification is e.g. the basis for the well known model view controller design pattern.

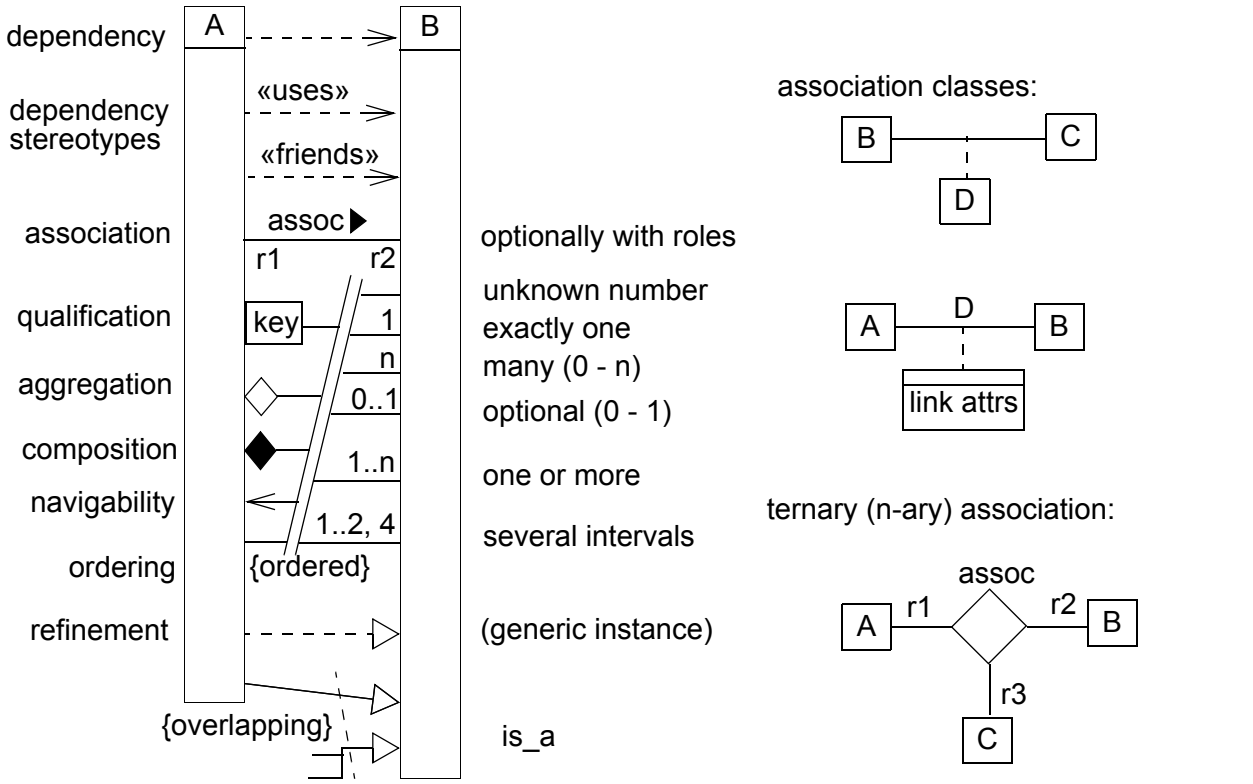
Note, the get and set methods for class attributes are implicitly known. Any UML attribute has these methods. Thus there is no need to mention these methods in the class diagram, explicitly. They should be visible on demand, only. Omitting these access methods simplifies the class diagram, significantly.

UML method declarations in the class diagram become corresponding Java method declarations in the implementation. Note, from UML method declarations in class diagrams only Java method declarations can be generated. The method bodies are generated from the corresponding UML behavior specifications, e.g. from story diagrams.

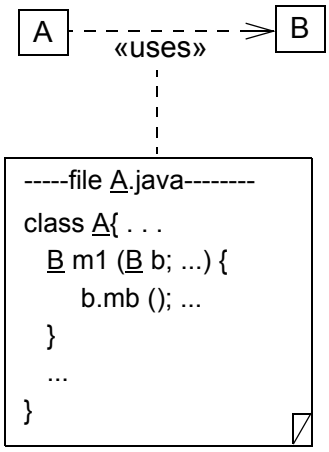
Example 3.5 shows the possible relationships between classes in UML class diagrams. Dashed lines denote so called class dependencies. This means, one class somehow depends on the other class. This information is important for maintenance and reuse reasons. If a class A relies on another class B then changes of class B may effect class A and thus class A should be reviewed, too. Similarly, the reuse of class A in another program will probably involve the reuse of class B, too. UML allows to attach certain stereotypes to class dependencies, e.g. «uses» and «friend». A «friends» dependency corresponds to the friends concept in C++. In Java there is no similar concept, thus we will not use «friends» dependencies.

Class relationships in UML:

Example 3.5: Possible relationships in UML class diagrams



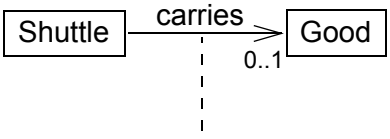
A «uses» dependency models some kind of import. A «uses» dependency allows class A to use class B and its public features. Class A may use class B e.g. as a method parameter or return type. Or some method of A may call some method of some other class that returns an object of type **B** and store this object in a local variable and call some operations on that local variable. Or class B may provide some static features that may be used by class A directly.



In languages like C++ or Modula2 a class or module A may use another type B, only, if A imports B, explicitly. In C++ this is usually done via an #include directive. In Java it is more common to import whole packages instead of single classes. Thus, in Java a «uses» dependency implies an import statement only if the corresponding classes belong to different packages and if the import is not yet created by some other «uses» dependency. Thus, usually we ignore «uses» dependencies.

UML provides various kinds of explicit associations with different cardinalities and with numerous adornments. Usually, associations are bidirectional. This means that an association may be navigated in both directions. For didactic reasons, we start our discussion of the implementation of associations showing explicit navigability in only one direction. In UML, association navigability can be shown with stick arrow heads. An association without arrow heads has arrow heads in both directions, implicitly. If only one arrow head is shown, than the association is navigable in that direction, only. This means, the source object knows how to reach the target object, but the target object does not know how to reach the source object.

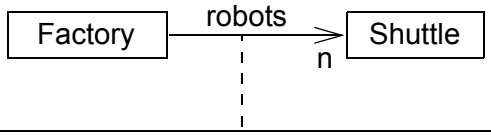
From the implementation point of view unidirectional associations of cardinality to-one correspond to usual pointers. Thus, we implement a unidirectional association e.g. with name carries that leads from class Shuttle to class Good via a Java attribute with name carries of type Good in the Java class Shuttle. Due to the already discussed best software engineering principles, the Java attribute is encapsulated via appropriate get and set methods. In addition, the comfort constructor gets an additional parameter for the initialization of the reference attribute.




```
----------file Shuttle.java----------
class Shuttle { ...
   public Shuttle ();
   public Shuttle ( ..., Good carries, ... ) {
      ...; setCarries (carries); ...
   }
   ...
   private Good carries;
   public Good getCarries () { return carries; }
   public void setCarries (Good carries) {
      this.carries = carries;
   }
   ...
}
```

Unidirectional associations of type to-many enable an object to hold multiple references to other objects. To achieve this, one should use predefined container classes, e.g. from the Java Foundation Classes library. Such a library of container classes provides a variety of generic containers like sets, maps, and lists based on different implementations e.g. balanced trees or hash tables. These container classes are organized in an inheritance hierarchy that offers a uniform interface for the usage of different kinds of containers. This uniform interface facilitates to replace one kind of container by another.

Example 3.6: Unidirectional to-many associations via generic container classes




```
----------file Factory.java----------
import java.util. *;
import de.uni_paderborn.fujaba.sdm. *;
...
class Factory { ...
   public Factory ();
   public Factory ( ..., Shuttle[] robots, ... ) {
      ...; addToRobots (robots); ...
   }
   ...
   private Hashset robots = new Hashset (SDM.cmp);
   public boolean hasInRobots (Shuttle elem) {
      return this.robots.contains (elem);
   }
   public Iterator iteratorOfRobots () {
      return this.robots.iterator ();
   }
   public void addToRobots (Shuttle elem) {
      if (elem != null && !this.hasInRobots (elem)) {
         this.robots.add (elem);
      }
   }
   public void addToRobots (Shuttle[] robots) {
      if (robots != null) {
         for (int i = 0; i < robots.length; i++) {
            addToRobots (robots [i]);
         }
      }
   }
   public void removeFromRobots (Shuttle elem) {
      if (this.hasInRobots (elem)) {
         this.robots.remove (elem);
   } }
   ...
}
```

Example 3.6 shows a default implementation for an unidirectional to-many association. Class Factory shall hold a set of Shuttles. As a basis we use an attribute robots of type Hashset. Of course this attribute is encapsulated by access methods. However, a simple get and set method that would retrieve the whole set or replace the whole set by a new one does not achieve the desired encapsulation. The encapsulation must achieve control over the elements that are stored in such an attribute. If the get

method returns the whole Hashset, the caller of that method could add new elements to that Hashset without any control. A solution to this problem could be to return a read-only adapter to the Hashset that allows to iterate through the set of elements and to query for certain elements but that forbids to modify the original set. However, with such a read-only adapter it becomes complicated to program side-effects for read access methods and such side effects may be necessary e.g. for the implementation of lazy evaluation strategies or for the management of read-locks within a transaction protocol for the coordination of concurrent threads.

Thus, we do not use adapter stubs to access internal sets but our code generation strategy generates explicit read and write access methods within class Factory that basically reflect the access methods of a Hashset. We generate for example method sizeOfRobots retrieving the current number of stored shuttles and method hasInRobots allowing to check whether a given shuttle is already stored and method iteratorOfRobots retrieving an iterator that allows to loop through all stored shuttles and method addToRobots storing new shuttles and method removeFromRobots removing given shuttles from the set of known shuttles. All these methods are implemented by just calling the corresponding access methods of the Hashset attribute robots. All these methods may be extended with specific side-effects, easily.

UML class diagrams allow very detailed specifications of association cardinalities. For example, for to-one associations one may distinguish between the cardinalities 0..1 and 1..1. In cardinality 1..1 the additional lower bound specifies that the corresponding Java reference attribute must never be null. Due to our experiences it is very hard to guarantee such a condition in practice. Let us assume that a Shuttle has exactly one Motor reachable via an engine association of cardinality 1..1. If the lower cardinality is interpreted very strictly, then every constructor of class Shuttle must initialize the corresponding engine attribute, directly. However, the Motor may be a complex product consisting of several subparts, itself. In addition, the shuttle may have other complex parts. For the initialization of all these parts, complex operations may be necessary. Some of these complex operations may visit the shuttle under construction in order to derive some specific construction details. Such a recursive visit is very problematic since the shuttle is still under construction and some methods may be used that rely on the invariants guaranteed for fully constructed shuttles, e.g. that a shuttle always has an engine. Such a recursive visit during the construction phase may be considered as a programming error. However, for complex construction tasks it is frequently very handy first to create an incomplete object at some place and then to pass this object to several other system parts for specific completions. During this phase the object is still incomplete and may violate certain consistency constraints, e.g. lower cardinality constraints.

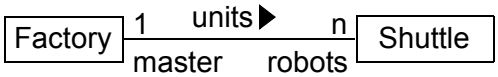
In general, consistency constraints like lower cardinalities may be violated, temporarily. It is a common maintenance problem that program changes cause situations where methods relying on certain consistency constraints are called on objects that accidently violate these consistency constraints at that moment. To address this risk, at the entry of a method one should check whether required consistency constraints hold. Such checks may be used through construction and testing phases and excluded later on, e.g. for performance reasons. Due to our experiences one should rely on such consistency constraints as seldom as possible. Methods with less consistency requirements may be used more flexibly e.g. if a method does not rely on the lower cardinality for shuttle engines, it can be applied to incomplete shuttles, too.

Thus, as far as lower cardinality constraints for associations are concerned, our constructors do not guarantee this property and one shall not rely on this constraint. Actually, we recommend not to use lower cardinality constraints in class diagrams at all. Due to similar arguments, other cardinality constraints like "a car has exactly 4 wheels" are not supported by our code generation concept, either. For

code generation we just distinguish to-one associations that are implemented as reference attributes and to-many associations that are implemented by generic container classes.

In UML, standard associations have bidirectional navigability. In Example 3.7 this means that a Factory object knows about its Shuttle objects as well as the Shuttle objects know about their Factory object. Bidirectional associations are naturally implemented by pairs of references, i.e. each reference from a factory to a shuttle has a corresponding reverse reference from the shuttle to the factory. The problem with this implementation idea is to guarantee the consistency of these reference pairs. Consistency of reference pairs means that if e.g. a factory f believes that a shuttle s belongs to its robots then the shuttle s should refer to factory f as its master. It must not happen, that the reverse reference points to some other factory or that the reverse reference is null. In our approach we achieve this through the encapsulating access methods for Java reference attributes. Our write access methods call each other, mutually. Each time one direction is established the corresponding access method automatically establishes the reverse direction, too, by calling the appropriate access method on its partner object, cf. Example 3.7:

Example 3.7: Bidirectional associations via pairs of mutual references:




```
----------file Factory.java----------------------
 1: import java.util. *;
 2: import de.uni_paderborn.fujaba.sdm. *;
 3: ...
 4: class Factory { ...
 5:    ...
 6:    private Hashset robots = new Hashset ( );
 7:    public void addToRobots (Shuttle elem) {
 8:       if (elem != null && !this.hasInRobots (elem)) {
 9:          this.robots.add (elem);
10:          elem.setMaster (this);
11:       }
12:    }
13:    public void removeFromRobots (Shuttle elem) {
14:       if (this.hasInRobots (elem)) {
15:          this.robots.remove (elem);
16:          elem.setMaster (null);
17:       }
18:    }
19:    ...
```

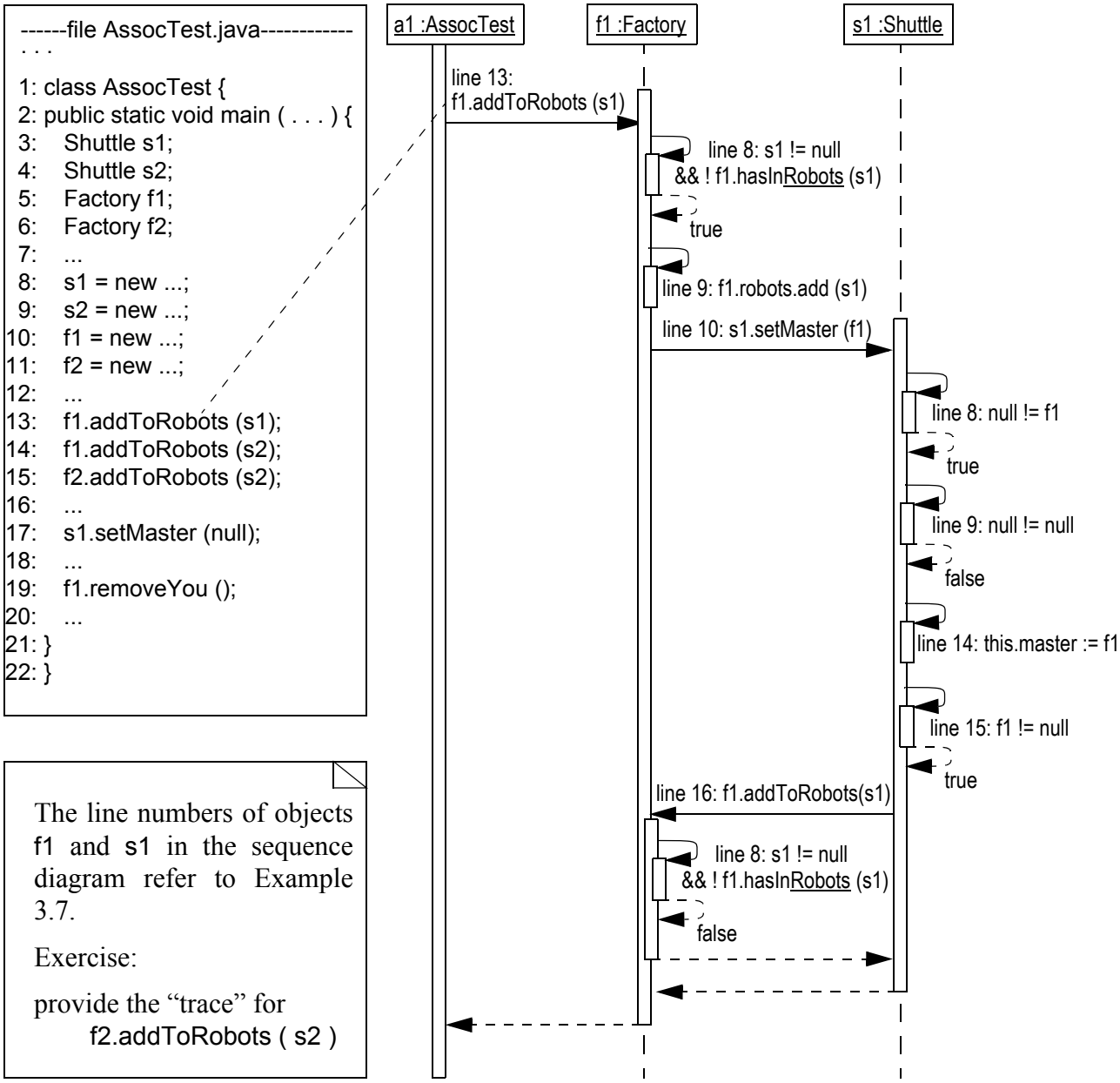
```
----------file Shuttle.java----------------------
 1: import java.util. *;
 2: import de.uni_paderborn.fujaba.sdm. *;
 3: ...
 4: class Shuttle { ...
 5:    ...
 6:    private Factory master = null;
 7:    public void setMaster (Factory master) {
 8:       if (this.master != master) { // newPartner
 9:          if (this.master != null) { // inform old partner
10:             Factory oldMaster = this.master;
11:             this.master = null;
12:             oldMaster.removeFromRobots (this);
13:          } // if
14:          this.master = master;
15:          if (master != null) { // inform new partner
16:             master.addToRobots (this);
17:          } // if
18:       } // if
19:    } ...
```

In Example 3.7 Java class Factory has an attribute robots of type Hashset implementing the to-many end of the units association and Java class Shuttle has an attribute master of type Factory implementing the reverse to-one end of the units association. Adding shuttles to a factory is done via the addToRobots method, cf. line 7 of file Factory.java. Setting a factory as master for a shuttle is done using method setMaster, cf. line 7 of File Shuttle.java. These two methods call each other mutually, cf. line 10 of file Factory.java and line 16 of File Shuttle.java, respectively. To see how these methods work Example 3.8 provides an application example and a trace of the method executions.

In Example 3.8, the main method of some class AssocTest first declares and initializes two Shuttle variables s1 and s2 and two Factory variable f1 and f2. In line 13 of file AssocTest.java we call method addToRobots with parameter s1 on variable f1. The right-hand side of Example 3.8 shows a sequence diagram tracing this method call. In line 8 of file Factory.java, method addToRobots first

checks if its parameter elem actually holds a shuttle and if this shuttle is not yet stored in the robots Hashset. The latter is done using method hasInRobots.

Example 3.8: Changing links based on mutually calling access methods



In our example both conditions hold and thus the if-branch is executed. Line 9 of file Factory.java stores the new shuttle s1 using method add of class Hashset. This establishes the forward reference from factory f1 to shuttle s1. Next, method addToRobots calls method setMaster on its parameter elem and passes the current factory object this as parameter, cf. line 10 of file Factory.java. Method setMaster of class Shuttle first checks whether the new master, passed as parameter, differs from the already known master stored in attribute this.master, cf. line 8 of File Shuttle.java. If parameter and attribute are already equal then nothing needs to be done and method setMaster terminates directly. In our example attribute this.master is still null and the parameter master contains factory f1. Thus, we enter the if-branch.

Line 9 of File Shuttle.java checks whether attribute this.master already contains a factory. If this is the case then the old master is going to be overwritten by a new master. If we would just overwrite the old attribute value then the old master would still belief that the shuttle belongs to it since only one

half of the reference pair would be changed. In order to avoid this inconsistency we inform the old master factory that it is going to loose one of its shuttles by calling method removeFromRobots on the old master and by passing the current shuttle as parameter, cf. line 12 of File Shuttle.java. Method removeFromRobots is given in lines 13 to 18 of file Factory.java. In line 14 we use method hasInRobots to check if the shuttle to be removed is actually contained in the set of robots. In this case line 15 uses method remove of class Hashset in order to remove the corresponding entry from the set of known robots. (Note, the shuttle itself is of course not removed.) Next, line 16 tries to remove the reverse reference by calling method setMaster(null) on parameter elem. This enters line 9 of File Shuttle.java, again. In order to terminate this recursion, we already have assigned null to attribute master in line 11 of File Shuttle.java and we used a local variable oldMaster for the call of method removeFromRobots, cf. line 10 and 12. Since the parameter master and the attribute this.master both have value null, the if-branch of method setMaster is not entered but method setMaster terminates, directly. After the execution of line 16 of file Factory.java method removeFromRobots terminates, too.

In Example 3.8, shuttle s1 does not have an old master. Thus the if branch of lines 10 to 12 of file Shuttle.java are skipped and we reach line 14. Line 14 assigns the parameter master to the attribute this.master, i.e. factory f1 is stored in shuttle s1. At this point the pair of references between f1 and s1 is installed properly. In our example method setMaster of class Shuttle has been called from its partner object / method. However, method setMaster may be called by other clients, too. If method setMaster is called from some other client, directly, then after line 14 only the reference from s1 to f1 is established but the reverse reference is still missing. Since method setMaster does not know its caller, it assumes that the reference from the master to the shuttle still needs to be established. Thus, line 15 of file Shuttle.java checks whether the value of attribute this.master refers to a factory now. This would be wrong if the parameter to the call of setMaster would be null. In our example the parameter is not equal to null. Thus, line 16 calls method addToRobots on factory f1, again, passing shuttle s1 as parameter. In order to avoid that methods addToRobots and method setMaster call each other, infinitely, the first line of each of these methods checks whether the new partner object is already known. In that case nothing needs to be done anymore and we terminate the mutual call chain. In our example this happens in line 8 of file Factory.java.

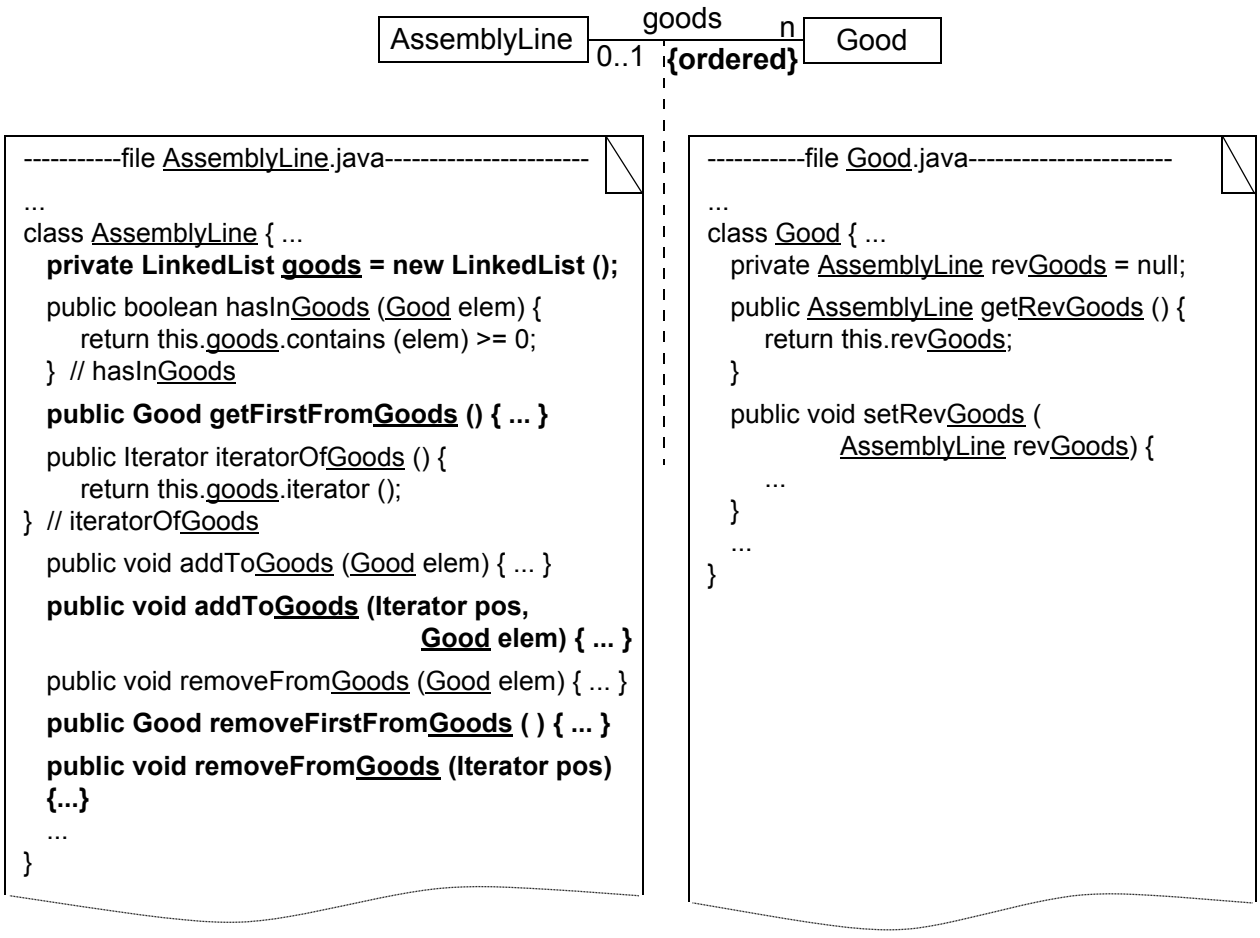
Note, a single call to either method addToRobots or to method setMaster suffices to establish a pair of references between a factory and a shuttle. The two methods call each other, automatically, in order to establish the reverse reference. Both methods may be used fully symmetrically. This reflects the bidirectional nature of associations. The encapsulating access methods guarantee the consistency of the reference pairs. If an object a has a reference to an object b then object b always has the reverse reference, too. If a reference is removed or overwritten, then the reverse reference is removed, too. Due to our experiences, the encapsulation of reference pairs by mutually calling access methods is the only feasible way to guarantee this consistency. If the client is responsible to update both sides of such reference, pairs serious maintenance problems are created. Frequently, a team member will fail do update both directions, correctly. This will corrupt the whole data structure and create a hard to debug situation. Having mutually calling access methods these pairs of references become very reliable. In the Fujaba project these access methods have proven to manage the consistency of the reference pairs without any problems.

Having such reliable reference pairs turns a usual object structure into a proper implementation of an object oriented graph. Such an object oriented graph has the advantage that each object knows all its neighbors. This enables us to cut certain objects from their current neighborhood, properly, and to move them to some other "place" in the graph. Usually, removing an object from an object structure is a difficult programming task, since one has to determine all references to this object in the whole object structure in order to reset them properly. In our approach this is very easy since the object has explicit (reverse) references to all such objects. Thus, we are able to generate a removeYou method

for an object that removes all references between an object and its neighbors. Method removeYou guarantees that no former neighbor has a "dangling" reference to the current object. Such an isolated object may then be connected to some other objects somewhere else in the object structure. One needs not to fear that a former neighbor still holds a reference to the object and could modify it later. Alternatively, the isolated object may be left as an easy target for the garbage collector.

UML class diagrams allow to employ several constraints on associations. A to-many association with the constraint {ordered} models a collection of neighbors that are stored in a user provided order. For the code generation this means, that we should not use a Hashset for the implementation of such an association but some kind of list. A list allows to insert objects at certain positions, e.g. to append them to the end of the list, and a list retrieves the elements in the stored order. Thus, our approach implements ordered associations just using the Java Foundation Classes container LinkedList instead of Hashset, cf. Example 3.9.

Example 3.9: Ordered Associations (Implemented as list)



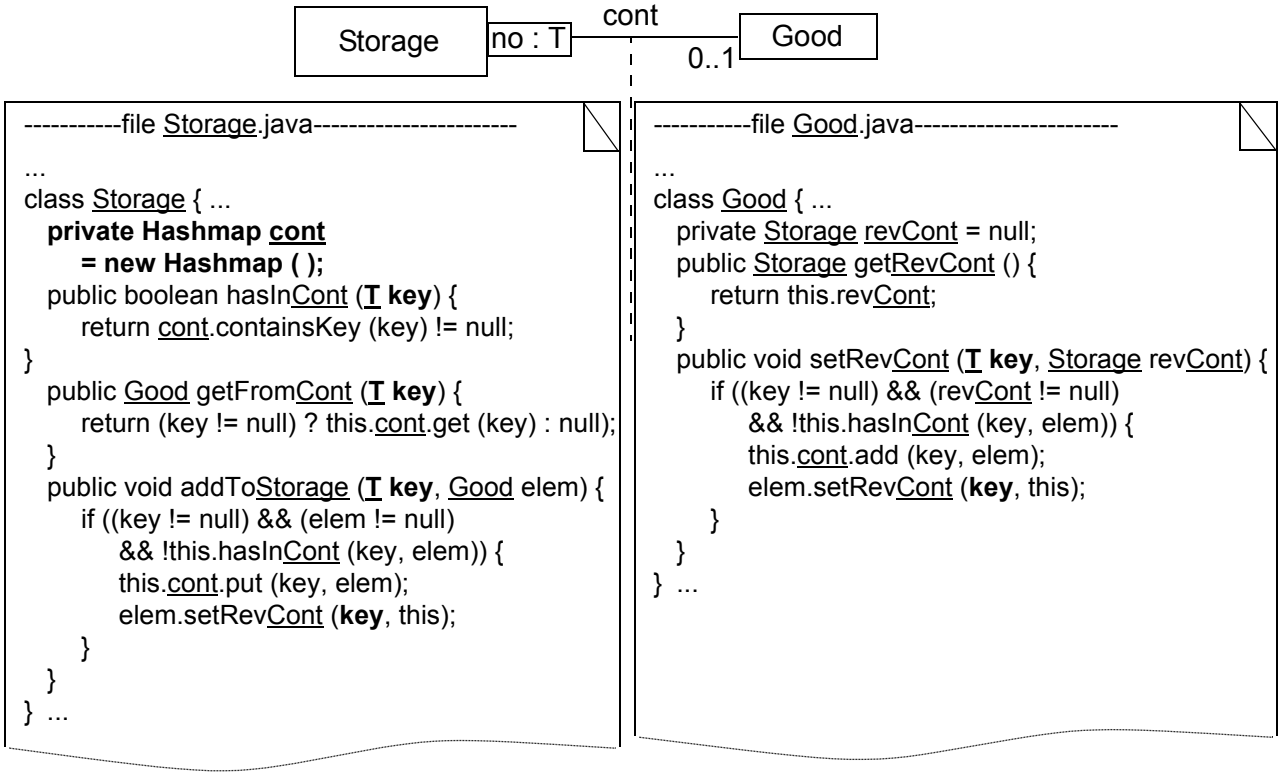
The LinkedList provides the same user interface as a Hashset and thus we encapsulate it by the same access methods. However, a linked list behaves a little bit different. By default, new elements are appended to the end of the list. And the linked list iterator visits the list elements in the given order. We provide some additional access methods that further exploit the special list properties. For example, a method to read the first list element and a method to insert an element at a certain position and a method to remove (and return) the first element from the list, cf. Example 3.9.

Note, the write access methods of linked lists that call their reverse access method must terminate the mutual call chain by using an internal flag whether they are already active. For details of this code start the Fujaba environment enter an ordered association and have a look at the generated code, cf. [Fujaba].

Alternatively to the constraint {ordered} one may employ the constraint {sorted}. A sorted association retrieves its elements in an order corresponding to some sorting criteria. Sorted associations are easily realized using a tree based set implementation like class Treeset. The sorting criteria may be determined by providing an appropriate comparator at the construction of the Treeset. For sorted associations special access methods that retrieve the first or last element are provided in addition to the usual access methods.

As a special adornment, UML associations may have a *qualifier*, cf. Example 3.10. A qualifier corresponds to a key or index attribute. Such a key or index attribute allows to access certain entries of the association via a key or index value. In Example 3.10, a Storage object stores Good objects via association cont under the key no. Thus, one may retrieve a certain good via a storage number no. Note, a qualified association always models a to-many association. The provided cardinality models the uniqueness of the qualifier. For a qualified association, cardinality 0..1 determines that the qualifier is unique, i.e. at most one object is stored under a given key value, i.e. the qualifier is a key attribute. Accordingly, cardinality 0..n specifies that multiple objects may share the same qualifier value, i.e. the qualifier is only an index value.

Example 3.10: Qualified Associations (Implemented via maps)



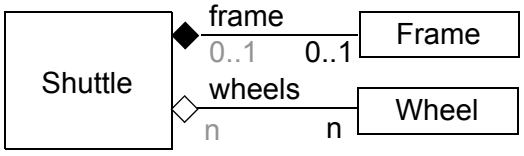
As already mentioned, we use map classes from the Java Foundation library in order to implement qualified associations. Again, these map classes conform to the uniform usage interface of container classes. However, in order to add an element to a map, one must provide a qualifier for that attribute. Since a qualified link may be established via the reverse method, the reverse method requires an additional parameter for the qualifier, too. In addition to the usual read access methods, we provide key or index based access methods.

Note, qualified associations inhibit a number of additional very tricky implementation problems. For example, at the removal of a qualified link one may not have the qualifier at hand. In this case one has to search through the whole map in order to detect the corresponding entry. However, the same object may be stored in one map under different keys. Sometimes one may want to remove only one of these entries, sometimes all of them. Sometimes one may want to retrieve the key(s) under which certain

objects are stored. Additional maps might be employed to support this operation efficiently. Some qualified associations may use a qualifier which is an attribute of the stored object, too, e.g. a serial number of a certain good. In this case the set method of such an attribute has to take care, that if the attribute is changed then the entries in the maps need to be updated, too. See [Fujaba] for more details.

UML associations may carry an aggregation or composition adornment, i.e. a white or a black diamond, respectively, cf. Example 3.11. An aggregation models a *contains* relationship, e.g. the shuttle may contain its wheels. Semantically, an UML aggregation implies that there should be no cycles in the aggregation structure. Cycles would just contradict to the general understanding of a contains relationship. However, it is not so easy to guarantee that a new aggregation link does not create a cycle. In simple cases it would suffice to check that at least one of the connected objects does not yet have an aggregation link attached or that no old link could form a chain with the new link. In general, this restriction is to conservative. It prevents e.g. that two nodes of distinct branches of a tree may be connected which is perfectly OK for aggregations. Unfortunately, there exists no simple scheme that allows to prevent cycles in aggregation structures. Thus, our code generation concept does not create any different code for aggregations then for plain associations.

Example 3.11: Aggregations and Compositions



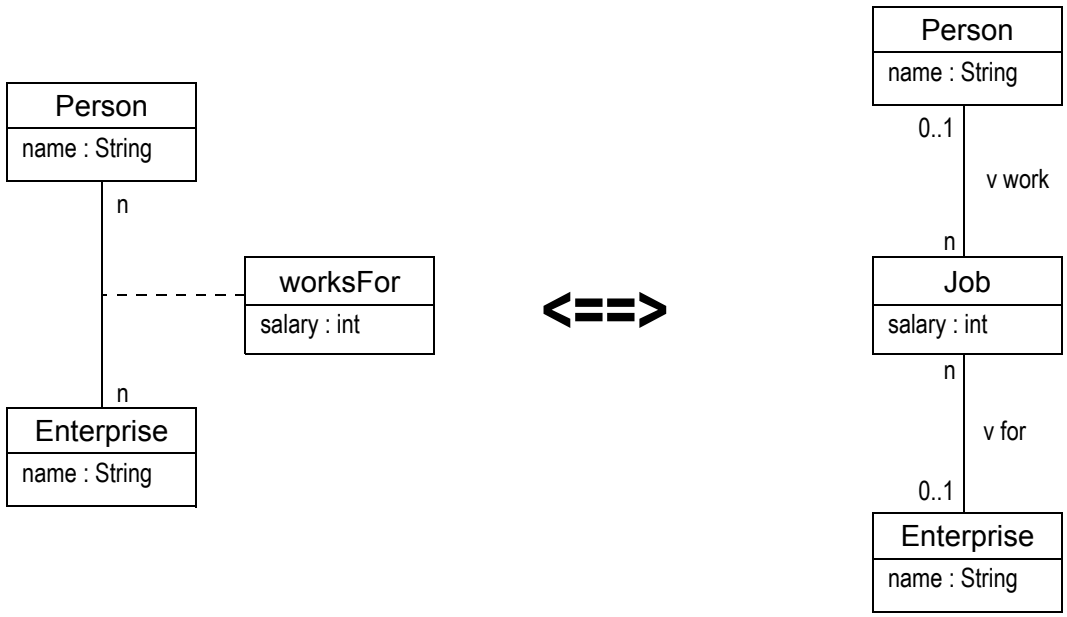

```
----------file Shuttle.java----------
class Shuttle { ...
  ...
  public void removeYou ( ) {
     ...
     if (this.frame != null)) {
        Frame oldFrame = this.frame;
        this.frame = null;
        this.frame.removeYou ();
     }
  ...
}
```

In UML, a composition represents a consists-of relationship. Semantically, this has two implications. First, a single object may be part of at most one parent object. This means, the composition adornment automatically implies a to-one cardinality at the same role. Second, a composition should imply a so-called co-incident life time. This means, the objects are constructed and glued together in an atomic step and from then on they stick together until they are deleted together. In practice this is hard to achieve. The situation is comparable to the lower association cardinalities that imply that the parent object must be constructed together with its child objects. As discussed for the lower association cardinalities such a constraint is easily violated, e.g. during the construction phase of such an object structure. This may happen by accident or by purpose, because one first creates only the parent object and then hands this incomplete object to some other system parts for specific completion steps. Thus our code does not enforce the co-incident life time. Parts may be added later on and they may also be exchanged or removed during the life time of the parent object. However, if the parent object is deleted and if it still has composite parts then our code deletes the parts, too. This is achieved via the already mentioned removeYou method. The removeYou method has the task to isolate an object in order to make it available for the garbage collector. In case of a composition, the removeYou call is forwarded to all still connected parts. Thus, the parts are isolated and garbage collected, too. This realizes a so-called weak-existence dependency.

So far we have discussed all kinds of association adornments from the left-hand side of Example 3.5, cf. page 46. In addition, UML class diagrams provide so-called association classes. Association

classes allow to attach attributes to associations. To attach attributes to associations is a frequent modeling requirement. In Example 3.12 a person may work for different enterprises with different salaries. The salary could not be stored at the person nor at the enterprise. It must be attached to the association. However, our implementation approach for associations is not able to attach attributes to associations. We implement associations via pairs of pointers and these pointers are not able to carry additional information. Thus, to implement attributed associations we have to introduce intermediate objects that hold the attributes. The intermediate object has references to the connected objects and connected objects have references to the intermediate objects. These references to the intermediate objects replace the direct references to the actual neighbor object. Altogether, this corresponds to the situation shown at the right-hand side of Example 3.12.

Example 3.12: A typical attributed association

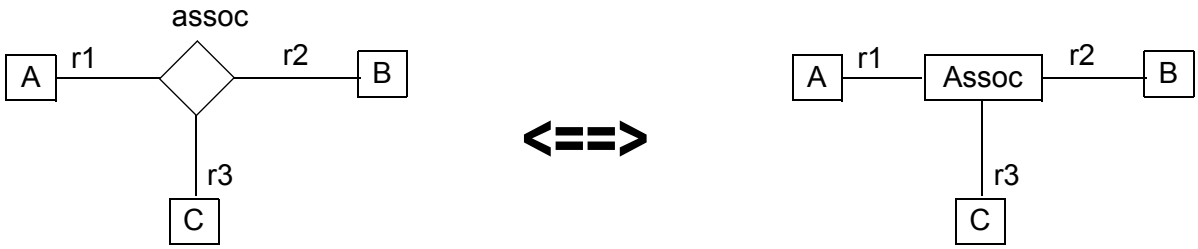


The problem with this implementation approach is that one has to answer a lot of detail questions concerning the behavior of such an implicit intermediate object. For example, how are the attributes accessed and modified. Should the intermediate object be destroyed if one of its neighbors is destroyed? Is it allowed to redirect an attributed link from one object to another? Does the creation of an attributed link between two objects overwrite (i.e. destroy) an already existing attributed link between those objects? Is it allowed to have multiple attributed links between the same pair of objects, e.g. may a single person work for the same enterprise in two jobs with different salaries? And so on. If the intermediate object is created implicitly during the code generation process, it is very difficult to answer all these detailed behavioral questions. In addition, the implicit intermediate class will probably become visible to the modeler in several situations. For example, if one wants to access or modify the attributes of the association, he will probably have to retrieve the intermediate object, first.

Thus, our approach does not support attributed associations but the user has to model the intermediate class and its behavior, explicitly. Introducing the intermediate class at the design level enables the user to deal with all the behavioral details explicitly that would be hard to define in an implicit transformation process. In addition, for an explicitly modeled intermediate object the code generation is very easy as already discussed. This facilitates our code generation concept and saves us a lot of implementation work for the Fujaba environment. In addition, this has the advantage that the gap between class diagram and code is narrowed and that it becomes easier to identify the class diagram elements and their implementation within the generated code. This facilitates the maintenance of that code.

Similar arguments hold for ternary or n-ary associations, cf. Example 3.13. Ternary and n-ary associations are most commonly implemented using implicit intermediate objects. However, an implicit transformation during the code generation involves a lot of questions on implementation details. For example, for ternary associations the interpretation of the provided cardinalities is quite hard and frequently wrong understood. According to the OMG UML standard semantics guide, cf. [UML99], the cardinality of a branch of an n-ary association defines, how often a given object may participate in instances of the n-ary association where all other participants are fixed. Assume that in the ternary association of Example 3.13 the branch attached to class A has cardinality to-one. According to the standard this does not mean that at-most one ternary association may be attached to an object of class A but only that the same pair of B and C objects may be attached to a given A object at-most once. While this semantics contradicted to the intuition of most of our trial users it is also very hard to implement such a constraint in our implementation approach. An A object may participate in a large number of ternary association instances and each time a new ternary association is attached one would have to check whether one of the existing ternary association instances connects exactly the same tuple of objects. Thus, our approach does not provide ternary and n-ary associations but the user has to introduce the necessary intermediate class explicitly. This clarifies all code generation questions and provides much more freedom in handling cardinality constraints.

Example 3.13: Realization of ternary associations



To summarize our code generation concept for associations, we implement bidirectional associations via pairs of references. For roles of cardinality to-one we use usual pointers. For roles of cardinality to-many we use standard container classes from the Java Foundation library. Different kinds of containers from this library allow to realize ordered, sorted, or qualified associations, easily. We encapsulate the access to the attributes that implement an association via access methods that call each other, mutually. This guarantees the referential integrity of the reference pairs implementing an association. A reference has always a proper reverse reference. Within about 300000 lines of code for the Fujaba environment itself and about 100000 lines of code for example applications this implementation approach for associations has proven to be very reliable and extremely practical. A single method call suffices to create a proper bidirectional link between two objects. Since each object knows all its neighbors it is very easy to cut an object out of a complex object structure (which is done via a simple call to the generated removeYou method). In addition, the bi-directional links turn a complex object structure into an object oriented graph. This enables us to apply the theoretic concepts of graph transformation rules to our object structures. This will be discussed in more detail in the next chapters.

However, these advantages are paid by some additional resource requirements. For each reference we have an additional reverse reference (if it is not required anyway). This probably doubles the memory space required for links. In addition, the creation and removal of links costs about double the time than the modification of a single reference. However, the read access is not slowed down.

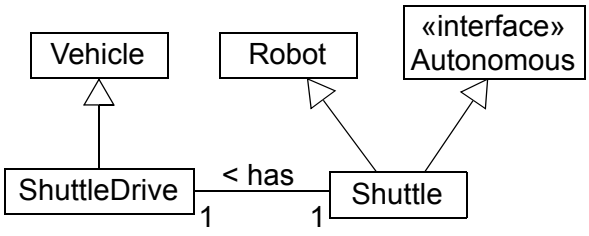
Our code generation concept for associations is close to the one of Rhapsody [Rhap]. Rhapsody generates pairs of references and appropriate access methods, too. The only difference is that Rhapsody employs two versions of each write access method. One version for the external use and one internal version to be called by the reverse partner method, only. Therefore, the Rhapsody generated code

needs not to take care of the termination of the mutual call chain. However, in the Rhapsody generated code a foreign client could call the internal version, accidently, and thereby invalidate the reference pair structure. Rational Rose creates pairs of pointers, too, cf. [RR-RT]. However, for to-many associations Rational Rose creates get and set methods that retrieve and write the whole set of references. This does not allow to encapsulate the set of references, properly. In addition, the Rational Rose generated access methods do not call each other, mutually. This means, the client programmer is responsible to keep the pairs of references consistent by establishing both directions himself. This is highly error prone and not acceptable. If one has to use Rational Rose he should adapt the generated access methods according to our discussion.

The inheritance relationship provided by UML class diagrams corresponds closely to the inheritance relationship of Java. However, UML inheritance offers some more flexibility that is not easily implemented in Java. For example, UML allows arbitrary multiple inheritance while Java allows at-most one usual parent class and additional parent classes must be so-called interface classes. In Java such interface classes are restricted to abstract methods and static attributes. Example 3.14 shows a situation where one might have wanted to use multiple inheritance. Class Shuttle should inherit from class Robot in order to be able to act like a Robot and from class Vehicle in order to inherit Vehicle properties. While one could draw such a multiple inheritance situation in a UML class diagram, easily, the translation of such an inheritance situation to Java would require the introduction of some help constructs, since Java does not allow that class Shuttle inherits from both full classes, Robot and Vehicle. As already discussed for attributed and n-ary associations, the introduction of such implicit help constructs during the code generation creates an artificial gap between design and implementation. This creates unnecessary difficulties for the mapping of design elements with their implementation. Thus, our code generation approach does not support arbitrary kinds of multiple inheritance but we restrict the inheritance at the UML class diagram level to the kind of inheritance that is provided by our target language, i.e. to the inheritance allowed in Java. For such a restricted use of inheritance the code generation is straight forward, cf. Example 3.14.

However, we have to provide a solution for situations where one would like to use multiple inheritance for multiple full classes. The general idea for such situations is shown in Example 3.14. It is always possible to replace an inheritance relationship by a one-to-one association. Instead of inheriting from two full classes, class Shuttle inherits from class Robot, only. In addition, we introduce a new class ShuttleDrive and this new class ShuttleDrive inherits from class Vehicle. In order to give the shuttle access to the vehicle properties, we connect class ShuttleDrive and class Shuttle with a one-to-one association. The idea is that the constructor of class Shuttle creates a ShuttleDrive, automatically, and that the shuttle and its drive are glued together. The resulting twin object has now all properties of class Robot and all properties of class Vehicle. Class Shuttle may extend and redefine any Robot properties. Class ShuttleDrive may extend and redefine any Vehicle properties. If some framework parts of the application require parameters of type Robot one may pass the Shuttle object which is a valid Robot heir. If some framework parts of the application require a Vehicle parameter, but one has only an object of type Shuttle at hand then one just traverses the connecting has association and uses the ShuttleDrive object as parameter. If one wants to access some Vehicle features at a Shuttle object, one just traverses the has association and accesses the ShuttleDrive object. Vice versa, from a ShuttleDrive object one easily reaches the twin Shuttle object.

Example 3.14: Inheritance




```
-----file Shuttle.java--------
public class Shuttle
   extends Robot
   implements Autonomous, ...
{
   Shuttle (...) {
       super (...);
       has = new ShuttleDrive ();
       ...
   }
       ...
}
```

```
-----file Usage.java--------
public class Usage {
public static void main ( ... ) {
   ...
   Shuttle s = new Shuttle ();
   Robot r = s;                    // no problem
   Vehicle v = s.getHas ();   // just pass the twin
   s.someRobotMethod ();   // no problem
   s.getHas ().someVehicleMethod ()
                                   // just access the twin
}
```

Due to our experience the replacement of a multiple inheritance situation by such twin classes works fine and creates only minor inconveniences in the usage of such twin objects. One just has to be aware of the twin classes and one has to traverse the connecting association, occasionally. However, one actually needs to be aware of this situation. An implicit creation of the twin classes during code generation would be confusing.

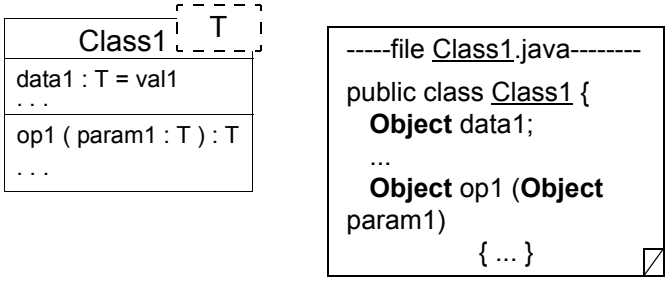
Above solution for multiple inheritance situation still has a serious problem in case of repeated inheritance. Let class Vehicle and class Robot have a common ancestor, e.g. class Machine, with attributes, e.g. an attribute machineId. Class Vehicle and class Robot inherit the Machine attributes. If class Shuttle would inherit from both classes Vehicle and Robot, directly, it would inherit the Machine attributes via both pathes, however, usually it would posses these attributes only once. In our twin class solution, either class Shuttle and class ShuttleDrive inherit the Machine attributes, independently. Thus, our twin objects posses two copies of the Machine attributes. This does not only waste memory space but in addition a serious consistency problem is created. Note, methods of class Robot may access the Machine attributes of a Shuttle object while methods of class Vehicle would automatically access the Machine attributes of the ShuttleDrive twin. Usually, the change of a Machine attribute within one of the twins should change the corresponding attribute in the other twin, accordingly. This may be achieved by overwriting the set methods of Machine attributes within class Shuttle and within class ShuttleDrive. These set methods could be modified to call each other, mutually, in order to guarantee that a change to a Machine attribute affects both copies of that attribute, the one within the Shuttle object and the one within the ShuttleDrive object. Note, if the set methods call each other mutually, one has to take care that the mutual call chain terminates as exemplified for the access methods for associations.

The last UML class diagram concept that we discuss is genericity or template parameters. Template parameters allow to define parameterized classes. Typically, such parameters are other types or classes. The parametrized class may use such parameter types as types for parameters for its own methods or as types for attributes or as types for local variables. In case of a restricted genericity, the

parameter types must be subtypes of predefined types, e.g. of type Comparable. This allows e.g. that methods of type Comparable may be invoked on attributes, parameters or local variables of the generic type. In general, genericity increases the possibility to create adaptable reusable classes in a type save manner, i.e. in a way that still allows strong static type checking. Genericity and inheritance have many common features, however as discussed in [Meye97], neither of these concepts can fully replace the other.

While genericity is a very important concept from the software engineering point of view, it is not provided by Java. Thus, we have to circumvent this feature within our code generation. There are two principal ways to deal with template classes during the implementation in Java. First, one may just generate multiple copies of the template class, one for each type used as template parameter. Within such copies all occurrences of the formal template parameter T are textually replaced by the actual template parameter type. This solution allows strong type checking at the Java code level. However, it creates multiple redundant copies of the whole class. If the class and all its method bodies are fully specified at the design level, this just creates larger applications. If some of the method bodies need to be programmed by hand or need to be modified later on, this may create a serious maintenance problem. One has to keep all the different copies consistent. Due to common software engineering experience, to keep such copies consistent is very error-prone and should be avoided. Therefore, our code generation concept does not follow this multiple copies approach.

Example 3.15: Genericity / templates



To avoid the consistency problem of multiple copies, our code generation concept for template classes generates just a single copy of that class. Within that single class the template parameter T is replaced by class Object, the common root class of Java. In case of the restriction of the template parameter to a certain class C we replace all occurrences of the template parameter T with the minimally required type C. This creates only a single copy of the generic class a code level. However, this single copy has less strict type checking properties. For input parameters and write access to attributes this may create the problem that a given instance of the generic class does not only accept actual values of type T but it accepts actual values of any (sub)type (of class C). Consider for example a generic container class Set that is used within the design to hold e.g. shuttles, i.e. the formal template parameter T is instantiated with the actual value Shuttle. We would generate a Set implementation in Java with an add method with a formal parameter of type Object or e.g. of type Comparable. This method could legally be called with a Person object as parameter, too. In order to guarantee that the set is filled with shuttles, only, we would need additional runtime type checking operations. Another problem is related to the return type of methods and to read access to attributes. In our implementation we replace the template parameter T with class Object. This may create methods that return Object values, only, instead of T values. If one calls such a method in order to retrieve a T value he has to cast the result value from type object to type T, explicitly. In our generic container class example this is especially problematic. We would try to convert objects retrieved from the container into Shuttle objects. However, although the container is intended for shuttles only, it could hold arbitrary objects, e.g. persons. In Java, the attempt to up-cast a Person object into a Shuttle object would result in a runtime excep-

tion. If that exception is not caught and handled, properly, this may cause the termination of the whole application.

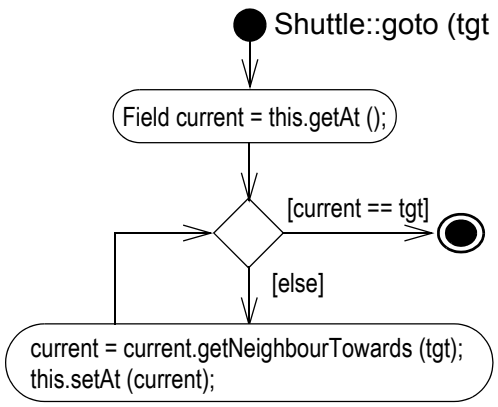
In principle, these casting problems are a major drawback of our code generation concept for template classes. However, due to experiences this approach causes no harms, in practice. Within our implementation we never experienced a situation, where a container contained objects of an unexpected type. That one has to write the up-cast operations on every read access to a generic value is a little bit tedious, but one can live with it. Serious maintenance problems are not caused.

3.2 Activity Diagrams

Activity diagrams are already very close to the implementation level. In our approach, activity diagrams are used for the specification of method bodies, only. We denote the modelled method at the start symbol with class and method name and with formal parameters and return type. This enables us to generate the corresponding method header within the corresponding class. The transitions of an activity diagram model the control flow within the method body. The activities represent basic blocks of code. For automatic code generation, the activities must contain valid Java code that is just copied into the generated method body. Similarly, all guards must represent valid boolean Java expressions that may just be copied into the generated code e.g. as conditions for if-statements

The main task for the code generation from activity diagrams is to analyse the control flow represented by an activity diagram and to translate the depicted control flow in similar control structures in Java. Ideally, one employs only so-called well-formed activity diagrams. A well-formed activity diagram consists of nested blocks of sub-diagrams where each nesting level corresponds either to a sequence of sub-blocks or of guarded branches or of a loop. The nesting structures of such a well-formed activity diagram may be analyzed and translated into the corresponding sequences of statements or into if-statements or into while statements, cf. Example 3.16.

Example 3.16: Translation of well-formed activity diagrams



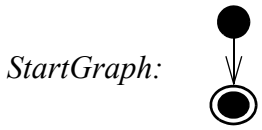

```
-----file Shuttle.java--------
public class Shuttle { ...
   public void goto (Field tgt) {
      Field current = this.getAt ();
      while ( ! current == tgt) {
         current = current.getNeighbourTowards (tgt);
         this.setAt (current);
      }
      return;
   }
}
```

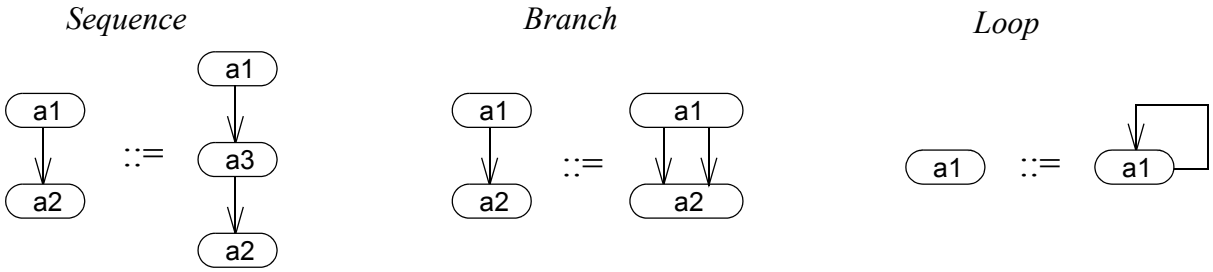
Example 3.16 shows method goto of class Shuttle with one parameter tgt of type Field. The branch activity (with the diamond shape) and the lower activity form a while loop. The first activity and the while loop form a sequence. Thus, the generated method body starts with the content of the first activity followed by a while loop containing the content of the lower activity. The condition of the while loop is derived from the guard attached to the transition leaving the branch activity towards the stop activity.

While Example 3.16 is very simple and the detection of the depicted control structure is quite obvious, in general the analysis of an activity diagram and the analysis of its nesting structure and well-formed-

ness and the translation into Java control structures is not that trivial. In our approach, we employ graph grammar techniques for this task. Definition 3.17 shows a graph grammar that generates the set of all well-formed activity graphs. This graph grammar may not only be used to generate well formed activity graphs but also for parsing well-formed activity graphs. In [RS95] Rekers and Schürr propose efficient techniques for using graph grammars for the parsing of visual languages. This approach allows to analyse a given graph and to reconstruct the derivation tree of rule applications that creates that graph from the start graph. With the help of such a derivation tree the translation of well-formed activity diagrams into the corresponding Java control structures becomes very simple. For the details of this code generation step cf. [Klein98].

Definition 3.17: Graph grammar for well-formed activity graphs

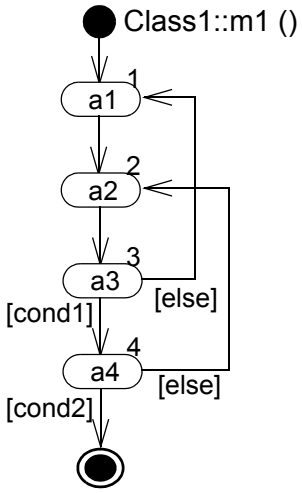




However, not all activity diagrams are well-formed. Example 3.18 shows a non-well-formed activity diagram. The depicted control flow does not correspond to nested control structures and thus a direct translation into Java control structures is not possible. Thus we have developed a second code generation approach that is able to deal with any kind of activity diagram and that is even much simpler to implement. The basic idea is to number the activities and to use a kind of program counter and code that computes the number of the next activity to be executed and to use a switch-case statement to jump to that activity. Each branch of the switch-case statement first contains the code of the corresponding activity. This code is followed by code computing the number of the next activity that shall be executed. The number of the next activity is computed according to the depicted transitions. If only one (unguarded) transition is present, the number of the next activity is just a constant, cf. activity / case 1. If multiple outgoing transitions with guards exist, then an if-then-else chain is generated that checks one guard after the other, cf. activities / cases 3 and 4. Note, in our formal model, transitions with multiple guards have ordering numbers defining the order in which the guards shall be considered. This is now used to define the order in which the corresponding if-conditions are to be generated. The whole switch-case statement is embedded into a while loop that executes an activity and computes the next activity until a stop activity is reached. In our example this is the case if the program counter reaches value 5.

The translation of activity diagrams to such switch-case constructs is straight-forward and always possible. However, the resulting code looks a little bit artificial and does not represent the control structures of the original activity diagram, anymore. If the code has to be understood or modified manually later on, this creates a maintenance problem. Therefore, we consider the translation of well-formed activity diagrams into usual nested Java control structures as valuable and worthwhile.

Example 3.18: Non-well-formed control flow




```
-----file Class1.java--------
public class Class1 { ...
public void m1 () {
      step = 1;
      while (step != 5) {
          switch (step) {
          case 1: a1; step = 2; break;
          case 2: a2; step = 3; break;
          case 3: a3; if (cond1) step = 4; else step = 1; break;
          case 4: a4; if (cond2) step = 5; else step = 2; break;
          }
      }
}
}
```

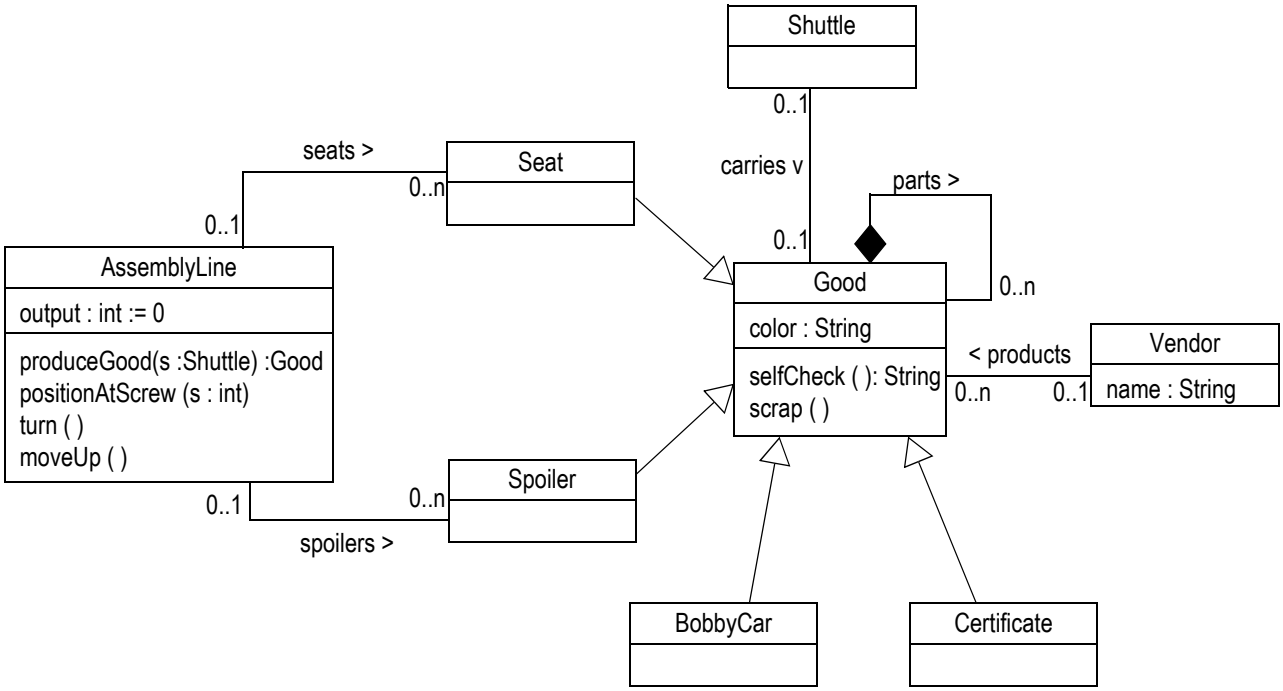
3.3 Collaboration Diagrams

As already discussed we use collaboration diagrams embedded in activity diagrams and in statecharts, cf. chapter 2. We simplified the use of collaboration diagrams systematically by assigning a standard semantics to the graphical elements of a collaboration diagram. For example, a link between two objects is automatically used for the determination of the neighbors of already known objects. Basically, our use and interpretation of collaboration diagrams relies on graph grammar theory, cf. Appendix A:. However, our use of collaboration diagrams corresponds to OQL queries to an objectoriented database or to a constraint satisfaction problem, too. Thus, before we discuss our code generation concepts for collaboration diagrams, we outline these interpretations of collaboration diagrams in more detail. From these different points of view, we derive many design rationals for the code generation process.

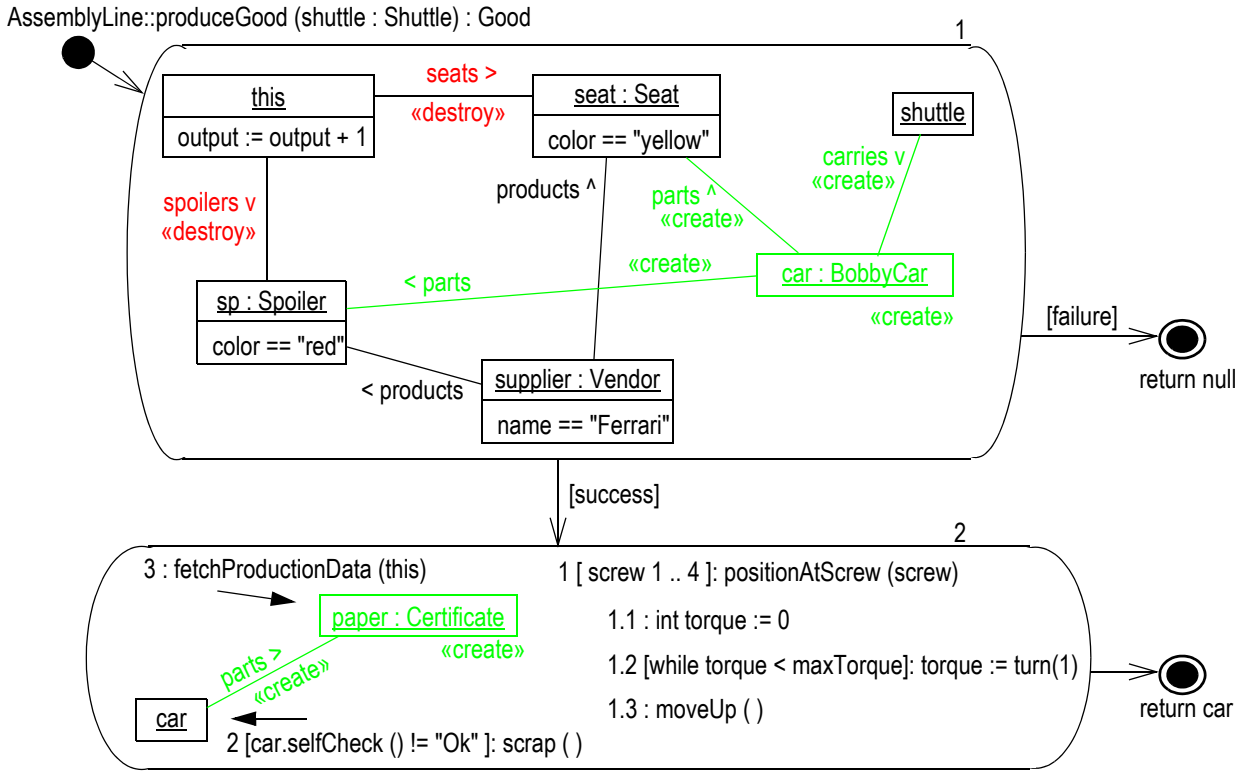
Example 3.19 shows a simplified class diagram for a material flow system example. This class diagram serves as basis for the following collaboration diagram examples. Example 3.20 shows a story diagram with two embedded collaboration diagrams. The first collaboration diagram illustrate the use of the graphical elements of a collaboration diagram in our approach. The second collaboration diagram exemplifies the use of collaboration messages in our approach.

Example 3.21 shows how the first collaboration diagram of Example 3.20 may be interpreted as an OQL query to an object oriented database. Basically, each object from the collaboration diagram is interpreted as a variable that occurs in the select part of the query. Next, for all variables we have to define the domain of the database that is queried. Objectoriented databases manage the extensions of all classes, explicitly. This means, they provide so-called extension sets that contain all valid instances of the corresponding class that are stored in the database. In the from part of the query, we use these extensions as search domains for our queries. Thus, we search through the extensions of classes Seat, Spoiler, and Vendor. In addition, we employ the knowledge of the currently active object this which could e.g. be stored in a special global variable H_Stack of the database. The same global variable H_Stack may provide access to the parameters of the currently executed method, too. The third query section, the where part, specifies the conditions or constraints that must be fulfilled by the queried objects. This part of the query is first used to represent the attribute constraints of our collaboration diagram, e.g. that the seat must be yellow. Second, we employ a set of conditions that represent the links of our collaboration diagram, e.g. the condition supplier.hasInProducts (seat).

Example 3.19: Class Diagram for the transportation system example



Example 3.20: A story diagram with complex story patterns



Example 3.21: Collaboration diagram 1 of Example 3.20 interpreted as OQL query

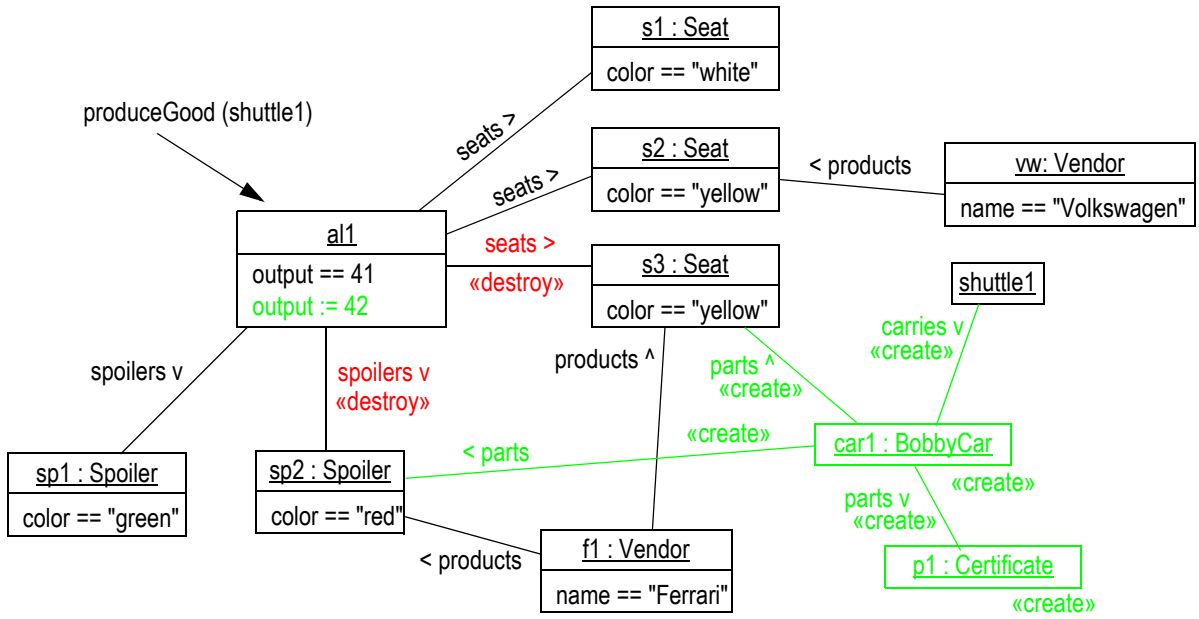
```
select
  this : AssemblyLine, seat : Seat, sp : Spoiler, shuttle Shuttle, supplier : Vendor
from
  H_Stack.current, Seat.extension, Spoiler.extension, H_Stack.parameter ["shuttle" ],
  Vendor.extension
where
  seat.color == "yellow"
  sp.color == "red"
  supplier.name == "Ferrari"
  this.hasInSeats (seat)
  this.hasInSpoilers (sp)
  supplier.hasInProducts (seat)
  supplier.hasInProducts (sp)
execute
  . . .
```

Example 3.22 shows an example object structure for the application of method produceGood. If we assume that this object structure is stored in an objectoriented database and that al1 is our current object and that shuttle1 is passed as parameter to a call of method produceGood, then we may issue the OQL query depicted in Example 3.21 to this database. Such a query would retrieve objects al1, s3, sp2, shuttle1, and f1 as values for variables this, seat, sp, shuttle, and supplier, respectively. This is the only combination of objects in the depicted object structure that fulfills all required conditions.

Interpreting a collaboration diagram as an OQL query is also a code generation alternative. We could employ an objectoriented database to store our current object structure and we could translate collaboration diagrams as described into OQL queries. The query optimizer of the objectoriented database would then execute the difficult task of finding objects that fulfill all required constraints. While this makes our code generation task straight forward, this approach would imply the overhead of an objectoriented database for the execution of story diagrams. In addition to OQL queries, objectoriented databases provide persistence and transaction management concepts. However, these advantages are paid with a significant loss of performance. Due to our experiences with the Progres system, the usage of a database system reduces the execution speed by a factor of about 1000. If one does not need persistence and transaction management, this slow down is not acceptable. In addition, for small applications the requirement of employing a full fledged objectoriented database does not carry its weight in terms of code size and installation efforts.

Alternatively, one may interpret a collaboration diagram as a constraint satisfaction problem, cf. Example 3.23. Viewing our example as a constraint satisfaction problem is very similar to the OQL interpretation. However, this view allows to employ the rich theory of constraint satisfaction problems for the execution of collaboration diagrams. Constraint satisfaction problems are well studied and there exist a number of different general purpose constraint solvers for different classes of constraint satisfaction problems. Thus, another alternative for the execution of collaboration diagram would be to generate input for such a general purpose constraint solver and to employ the constraint solver to retrieve a variable binding that fulfills all depicted conditions. However, this would involve the problem, that the constraint solver needs access to the whole object structure that needs to be searched. Usually, these constraint solvers employ their own special purpose data structures to represent facts and constraints. Thus, we would have to transfer the whole object structure, which may involve some millions of objects, to the internal data structures of the constraint solver and we would have to keep the real object structure and the one known by the constraint solver consistent. This does not carry its weight.

Example 3.22: An example object structure for the execution of method produceGood



Example 3.23: Activity 1 of Example 3.20 interpreted as constraint satisfaction problem

variables	**domains**
this	{al1}
seat	{s1, s2, s3}
sp	{sp1, sp2}
shuttle	{shuttle1}
supplier	{vw, f1}

constraints

seat.color == "yellow"
sp.color == "red"
supplier.name == "Ferrari"
this.hasInSeats (seat)
this.hasInSpoilers (sp)
supplier.hasInProducts (seat)
supplier.hasInProducts (sp)

In our approach, we adapt the optimization and query execution techniques and the constraint satisfaction strategies for our own code generation concept. Our code generation concept uses these techniques to generate plain Java code that works on the object structures and Java classes generated from UML class diagrams, directly. Again, the basic concept of the code generated for a collaboration diagram is to interpret the depicted objects as so-called *collaboration* variables. We use the links between objects for the retrieval of neighbor objects from known objects. Attribute conditions as well as negative diagram elements and general constraints are turned into boolean conditions on the values of collaboration variables. For to-many associations/links we employ nested search loops that explore the search space and retrieve candidate values for the collaboration variables that are then checked by evaluating the boolean conditions representing the constraints of the collaboration diagram. If a valid combination of objects is found for all collaboration variables that fulfills all constraints, then the effects of the collaboration diagram are executed, i.e to-be-destroyed elements are removed or isolated, to-be-created elements are created, attribute modifications are performed, and finally explicit collaboration messages are executed.

To illustrate our collaboration diagram implementation concepts, Example 3.24 shows the Java code generated for Example 3.20. We first discuss a sample execution of this code on the object structure shown in Example 3.22. This sample execution illustrates how the generated code explores the object structure in order to find valid candidates for all collaboration variables. This is the basis for the discussion of the code generation strategy itself.

If method produceGood is called on object al1 with shuttle1 as parameter, the first step is to initialize the local variable sdmSuccess. Variable sdmSuccess flags whether valid candidates for all collaboration variables have been found and whether the collaboration diagram has been executed, successfully. The first task in executing a collaboration diagram is to identify all participants. For the execution of the first collaboration diagram of method produceGood, we already know the values of variables this and shuttle (the latter has been passed as parameter.) Our strategy is to identify the neighbors of known variables by traversing the connecting links by calling the corresponding access methods generated from the class diagram. Our example code starts with the seats link connecting the known variable this and variable seat, with unknown content. According to the class diagram in Example 3.19, the seats association from AssemblyLine towards Seat objects is a to-many association. In our example assembly line al1 has 3 Seat objects attached, namely s1, s2, and s3. We deal with to-many links using a loop iterating through the set of reachable objects. For each candidate object, we test whether it fulfills all depicted conditions. Thus, line 18 uses method iteratorOfSeats to create an iterator that allows to loop through the set of all seats attached to assembly line al1. Lines 19 to 50 contain a while loop iterating through the set of seats until variable sdmSuccess becomes true or until the iterator runs out of candidates. In our example execution, the while condition holds and we enter the loop. Line 20 just retrieves the current iterator element using method seatIter.next (). In our example execution, this assigns object s1 to variable seat.

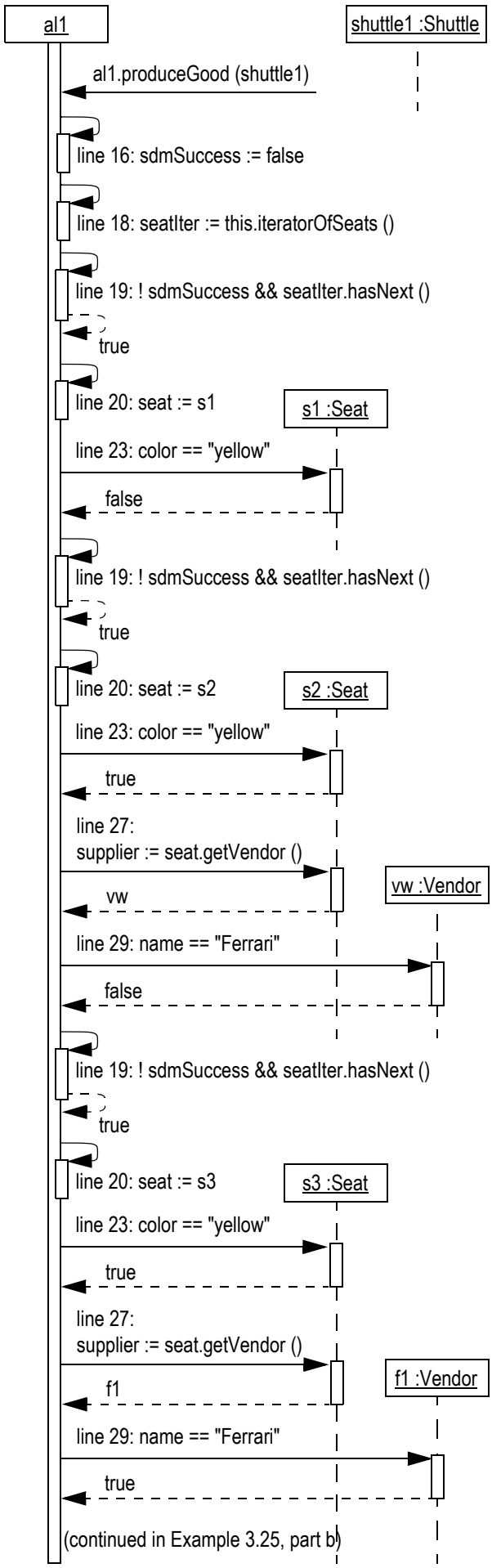
Our collaboration diagram specifies that the attribute color of the wanted seat must have value "yellow"[3], cf. line 23. This condition is checked using method JavaSDM.ensure (). This method is a small library routine provided by our runtime library. Method JavaSDM.ensure() has a boolean parameter. If the parameter is true, the method does nothing. If the parameter is false, method JavaSDM.ensure() throws a runtime exception. If such an exception is thrown, the normal code execution is terminated and the execution jumps to the next enclosing try-catch clause. In our case this would be line 49. Here, we use the exception mechanism of Java in order to jump to the end of our while loop and to start the next iteration. One could achieve the same behavior using a simple if-statement. However, the code for a complex collaboration diagram may involve the validation of many such conditions. Using an if-statement for each such condition would create an unacceptable nesting depth of if-statements. Using the exception handling mechanism avoids this problem and allows to generate straight forward code that just tests one condition after the other. If one of the conditions is violated, we jump to the enclosing try-catch clause. If all conditions are passed, we can continue with the execution of the collaboration diagram.

In our example execution, the attribute condition color=="yellow" is not fulfilled by object s1. Thus, in line 23 method JavaSDM.ensure () throughs a runtime exception which is caught in line 49. This reaches the end of the while loop and thus we check the while condition again. Variable sdmSuccess is still false and iterator seatIter still has candidates. Thus, line 20 retrieves the next candidate, i.e. object s2, and line 23 checks the color, again. This time the condition holds and we reach line 27.

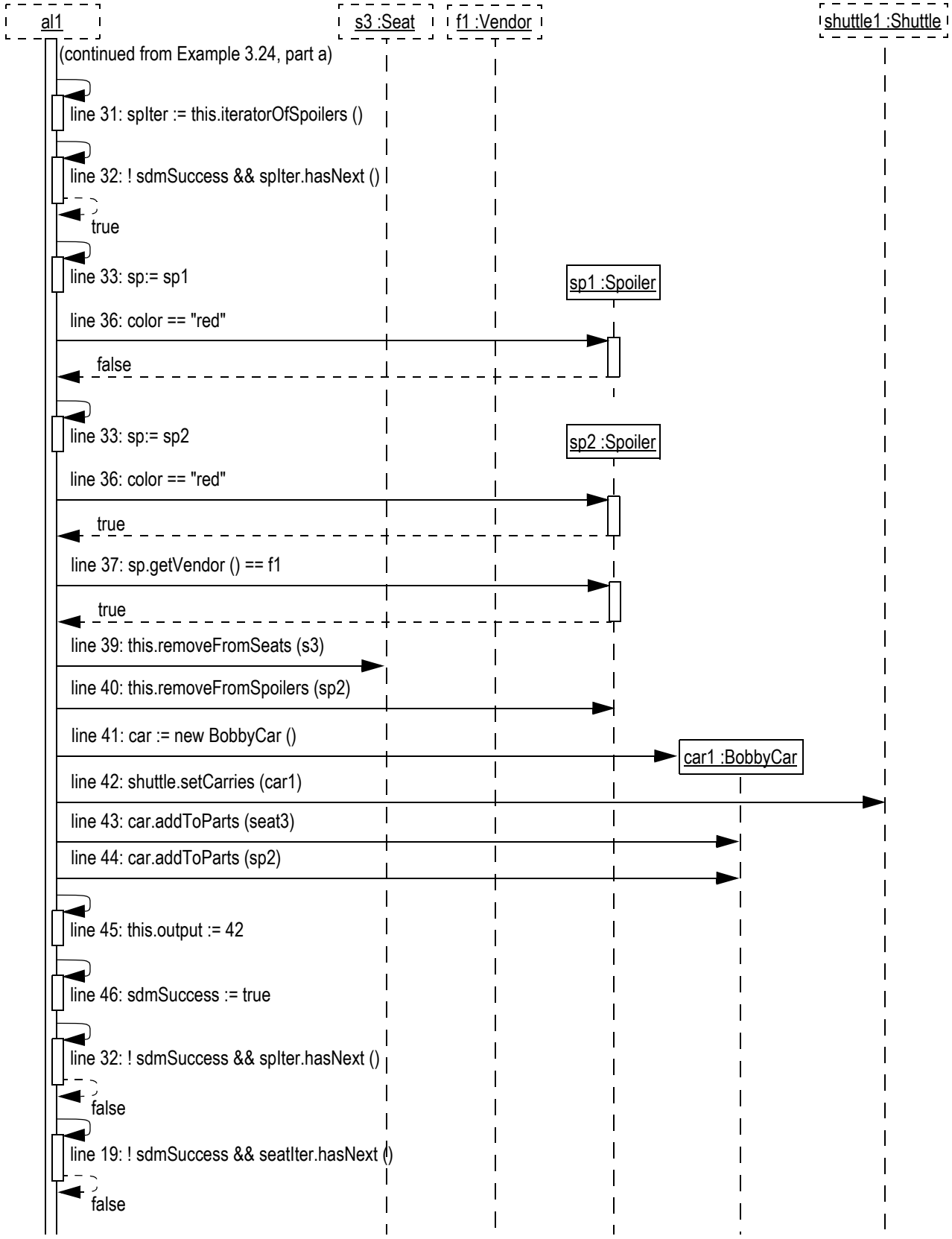
3. Note, attribute color may be null. Thus, we employ a slightly more complex expression for string comparison to avoid null pointer exceptions.

Example 3.24 (part a): Java code and trace for Example 3.20

```
----------file AssemblyLine.java----------
10: ...
11: public Good produceGood (Shuttle shuttle) {
12:    Seat seat; Spoiler sp; Vendor supplier;
13:    BobbyCar car; Certificate paper;
14:    boolean sdmSuccess;
15:    try { // identify participants for activity 1
16:    sdmSuccess = false;
17:    // bind seat : Seat
18:    Iterator seatIter = this.iteratorOfSeats ();
19:    while (! sdmSuccess && seatIter.hasNext ()) {
20:       seat = (Seat) seatIter.next ();
21:       try {
22:          // precondition check
23:          JavaSDM.ensure (seat.getColor() != null )
24:          JavaSDM.ensure (
25:                      seat.getColor().equals ("yellow"));
26:          // bind supplier : Vendor
27:          supplier = seat.getVendor ();
28:          JavaSDM.ensure ((supplier != null)
29:               && (supplier.getName ().equals ("Ferrari");
30:          // bind sp : Spoiler
31:          Iterator spIter = this.iteratorOfSpoilers ();
32:          while (! sdmSuccess && spIter.hasNext ())
33:             sp = (Spoiler) spIter.next ();
34:             try {
35:                JavaSDM.ensure (
36:                   (sp.getColor() == equals ("red"))
37:                   && (sp.getVendor () == supplier);
38:                // do modifications
39:                this.removeFromSeats (seat);
40:                this.removeFromSpoilers (sp);
41:                car = new BobbyCar ();
42:                shuttle.setCarries (car);
43:                car.addToParts (seat);
44:                car.addToParts (sp);
45:                this.setOutput (output +1);
46:                sdmSuccess = true;
47:             } catch (Exception e) {}
48:          } // while spIter
49:       } catch (Exception e) {}
50:    } // while seatIter
51: } catch (Exception e) {}
```



Example 3.25 (part b): Java code and trace for Example 3.20



So far we have found candidates for variables this, shuttle, and seat that fulfill the conditions that the seat shall be yellow and that this and seat shall be connected by a seats link. We have now two options to continue our search. We may try to find a candidate for variable sp by traversing the spoilers link or we may try to find a candidate for variable supplier by traversing the products link. According to the class diagram, the spoilers association has cardinality to-many from class AssemblyLine to class Spoiler while the products association has cardinality to-one from class Seat to class Vendor.

According to optimization principles known from database query processing and constraint satisfaction theory, one should always prefer to traverse a to-one link instead of a to-many link. Traversing a to-many links creates a set of candidates to be visited while traversing a to-one link creates only a single candidate to be visited. Keeping candidate sets as small as possible keeps the number of checks to be performed small and is thus more efficient. Thus we continue the exploration of our example object structure by traversing the products link from variable seat towards variable supplier.

Line 27 traverses the products link from seat to supplier against its usual direction. Recall that all links are bi-directional and that the default role name for the reverse direction is the name of the source class. Thus method getVendor retrieves the Vendor object that is attached to our current seat object s2. This assigns object vw as candidate value for variable supplier.

In this situation we could either try to traverse the spoilers link from assembly line this in order to retrieve a candidate for variable sp or we could validate the attribute condition for the Vendor object, i.e. if the vendor name equals to "Ferrari". According to the force-failure principle known from constraint satisfaction theory, one should validate constraints as soon as possible. If a constraint is not fulfilled, it invalidates a candidate choice. If this is done early, it may save the efforts of subsequent operations that are based on a wrong candidate choice. Thus, our example code checks the attribute condition first.

Line 29 checks whether the current candidate for variable supplier has a name attribute with value "Ferrari". This does not hold for object vw. Thus, we through an exception and start the next iteration of the while loop in line 19. The while condition is still fulfilled and line 20 retrieves object s3 as candidate for variable seat. Object s3 fulfills the color condition and line 27 retrieves object f1 as new candidate for variable supplier. Object f1 has name "Ferrari". Thus, we continue with the look-up of candidates for variable sp. Since the spoilers association is a to-many association, line 31 retrieves a new iterator using method iteratorOfSpoilers and line 31 starts a new while loop for this iterator and line 33 assigns new candidates to variable sp. In our example execution, which is continued in Example 3.25 (part b), these steps retrieve object sp1 as first candidate for variable sp.

Next, we check the color of the current candidate for variable sp, cf. line 36. This step fails and a runtime exception is raised. This time the runtime exception is caught by the try-catch statement covering lines 34 to 47. The assumption is that the violation of the currently checked condition is caused by a wrong candidate chosen in the inner while loop, i.e chosen for variable sp. Thus we handle the constraint violation by considering a new candidate for variable sp instead of jumping to the outer while loop and considering a new candidate for variable seat. We will discuss this strategy in more detail, below.

The second iteration of the inner while loop assigns object sp2 to variable sp. Object sp2 has the correct color. Thus, we reach line 37 and check whether the vendor attached to sp2 via a products link corresponds to variable supplier, i.e. whether f1 and sp1 are connected by an products link. In our example this condition holds, too.

At this point in our example execution, we have found candidates for all collaboration variables that fulfill all depicted attribute conditions and that are connected by the depicted links. Thus, we are ready to execute the effects shown in the collaboration diagram. First we execute the deletions. Lines 39 and 40 remove the seats and the spoilers link connecting object al1 to objects s3 and sp2, respectively. Second, the creations are executed. Line 41 creates a new BobbyCar object car1 and lines 42 to 45 create the new links and modify the attributes as depicted in the collaboration diagram.

Finally, our example execution reaches line 46 where we change variable sdmSuccess to true in order to flag the successful execution of the whole collaboration diagram. This change of variable sdmSuc-

cess causes the while loops of line 32 and line 19 to terminate. This terminates the execution of the first collaboration diagram.

The code for the second collaboration diagram is shown in Example 3.26 (part c). Its derivation from Example 3.20 and its execution is straight forward. We leave this as an exercise for the interested reader.

Example 3.26 (part c): Java code and trace for Example 3.20

```
52:    if (sdmSuccess) { // follow [success] transition
53:       try { // identify participants for activity 2
54:          // identify participants. Nothing to do.
55:          // do modifications
56:          paper = new Certificate ();
57:          car.addToParts (paper);
58:          // messages
59:          for (int screw =1 ; screw <= 4; screw ++) { // 1
60:             positionAtScrew (screw);
61:             int torque = 0;  // 1.1
62:             while (torque < maxTorque) {// 1.2
63:                turnOneDegree ();
64:             } // end [while torque < maxTorque]
65:          } // for [screw = 1..4]
66:          if (car.selfCheck () != "OK") {// 2
67:             car.scrap ();
68:          }; // fi [car.selfCheck () != "OK"]
69:          paper.fetchProductionData (this);// 3
70:          sdmSuccess = true;
71:       } catch (Exception e) { }
72:       return car;
73:    } else { // follow [failure] transition
74:       return [null];
75:    }
76: } // produceGood
77: ...
78: }
```

To summarize, we explore the search space for collaboration variables using nested search loops. According to the force-failure principle of constraint satisfaction theory, we validate constraints as soon as possible. In order to keep candidate sets small, we traverse to-one links before to-many links, if possible. In terms of constraint satisfaction theory, this strategy corresponds to a simple backtracking strategy. For constraint satisfaction problems this is the most basic standard approach. There exist numerous improvements of this simple standard approach. Here we just outline the back jumping approach. For other ideas cf. [FNT98, FNTZ98, Rud97, LV00].

A well known problem of the simple backtracking solution to constraint satisfaction is created if the nested search loops fail due to unrelated reasons. Assume that spoiler sp2 in Example 3.22 has green color instead of red color and that the example object structure contains several additional yellow seats produced by Ferrari. Since all spoilers have green color now. line 36 of our code fails, always. If the outer loop has reached seat s3, we enter the inner loop and test all spoilers. None of them passes the color condition. Thus, the while loop in line 32 terminates because iterator splter has no more elements. The execution reaches the outer while loop in line 19. If we have more yellow seats from Ferrari, the outer while loop chooses the next candidate and we reach the inner loop, once more. The inner loop now iterates through all spoilers, again, although the new choice for variable seat has no effect on the color condition for spoilers and the inner loop will fail again. Our code will run through the set of all existing spoilers for each yellow Ferrari seat, although a careful checking of the failure reason

could reveal that trying new seats cannot solve the color problem for spoilers. In constraint satisfaction theory one would address this problem with the so-called back-jumping strategy. For each failure condition it is statically checked, which candidate choice could affect it. If a failure occurs, one jumps back only to those candidate choices that could potentially solve the problem. In our example, we could implement such a back jumping behavior by attaching error codes to thrown exceptions within method JavaSDM.ensure() and by analyzing these error codes in the while conditions of our loops. In our example, the outer while loop could recognize that the inner while loop has failed due to a color problem and it could abort the search, too, since choosing another seat can not solve this color problem.

As one easily sees, the more sophisticated constraint satisfaction approaches require more complex problem analysis techniques. So far, the Fujaba environment implements only the "simple" backtracking approach. More sophisticated techniques are current work.

So far we have discussed the example code for one collaboration diagram and its execution on an example object structure. This discussion outlined the main ideas of our code generation concept for collaboration diagrams. In the following, we will discuss the code generation concepts in more detail.

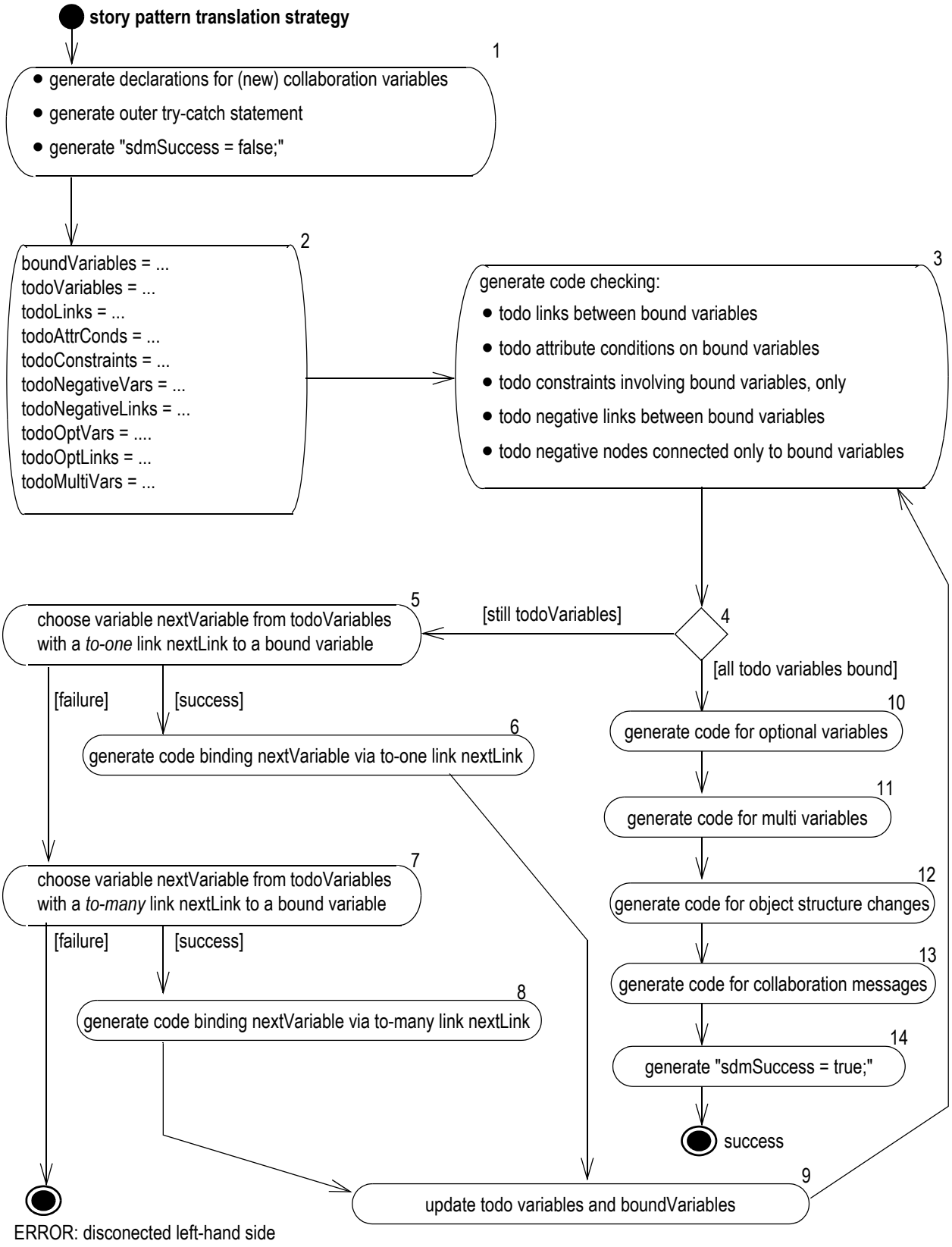
Definition 3.27 shows the general algorithm employed in the Fujaba environment for the generation of Java code from collaboration diagrams. The first step of our algorithm is to generate Java declarations for the employed collaboration variables. Note, according to our semantics, all collaboration diagrams/story patterns within the same activity diagram or story diagram share a common name space, cf. Appendix A. If the same collaboration variable is used in different collaboration diagrams, only one Java declaration is required. If a collaboration variable refers to a method parameter, no additional Java declaration is required. For standard and optional collaboration variables, a Java variable with the same name and type is generated. For multi-object variables we use a standard container type, i.e. HashSet. In Example 3.24 (part a) this step generated lines 12 to 13 from the activity diagram in Example 3.20. Note, collaboration variable this needs no Java declaration, collaboration variable shuttle refers to a method parameter, and collaboration variable car is used in both activities of Example 3.20, but it shares a common Java variable declaration.

In addition, activity 1 of our code generation algorithm generates a try-catch statement surrounding the whole collaboration diagram code and it generates the initialization of variable sdmSuccess = false. This allows us to use method JavaSDM.ensure() for the validation of various kinds of constraints. If method JavaSDM.ensure() throughs an exception it is (at latest) caught by this try-catch block and the search terminates with variable sdmSuccess flagging the failure.

Activity 2 of our code generation algorithm initializes some variables that we use for book-keeping purposes. As discussed, our execution strategy for collaboration diagrams starts with already known or bound objects and tries to identify not yet bound objects by traversing links. Thus, our code generation algorithm first identifies the already bound objects. Recall, already bound objects are in a story pattern depicted as boxes showing only the variable name and omitting the variable type. If a link is traversed from an already bound object to an not-yet bound object, this traversal assigns one or more candidate objects to the corresponding collaboration variable. Thereby, that collaboration variable becomes bound, too. Thus, in subsequent steps it may serve as the source for another link traversal. We employ the algorithm variables boundVariables and todoVariables in order to store the collaboration variables that are already bound and that still need to be bound, respectively. In addition, we use the algorithm variables todoLinks, todoAttrConds, todoConstraints, todoNegativeVars, and todoNegativeLinks to store links that are not yet used or checked, attribute conditions that are not yet checked, general constraints that are not yet checked, negative collaboration variables that need to be excluded, and negative links that must not exist, respectively. In addition, algorithm variables

todoOptVars and todoOpt link store optional collaboration variables and links. Algorithm variable todoMultiVars store the multi-object collaboration variables.

Definition 3.27: Code generation strategy for collaboration diagrams



Once we have done the preparation work, we enter the central loop of our code generation algorithm. We have already introduced the force-failure principle of constraint satisfaction algorithms. Accordingly, the first step in our code generation loop is to look for all kinds of conditions that could be

checked now. Note, a condition can be checked as soon as all collaboration variables, it refers to, are bound. In simple cases like the color conditions in Example 3.20 a condition may refer only to a single variable. Such a condition may be checked as soon as the corresponding variable has been bound. However, more complex conditions like {x.attr > y.attr} may refer to several variables. Similarly, a negative node may be connected to multiple other collaboration nodes. Such a negative node is tested as soon as all attached standard nodes are bound. Note, a link may be used to traverse from a bound to a not-yet bound collaboration variable. In addition, it may happen that two variables become bound that are connected by another link which has not been used for traversal. We call such links check-links. Such check-links represent a condition that is checked as soon as two attached collaboration variables have been bound.

As already discussed in the example execution, we employ the special method JavaSDM.ensure() to validate conditions. This makes code generation straight forward. We just turn the condition to be checked into the corresponding Java expression and embed the resulting expression in a call to method JavaSDM.ensure (), cf. to the corresponding clauses in Example 3.24. This works fine for almost all kinds of conditions. Only negative nodes with multiple standard neighbors require some additional efforts. To be honest, the implementation of the latter construct is still current work.

The next step in the main loop of our code generation algorithm checks whether there are still not-yet bound collaboration variables stored in algorithm variable todoVariables. If this is the case, activity 5 first tries to find a to-one link leading from an already bound collaboration variable to a not-yet bound collaboration variable. As discussed, we prefer to-one links compared to to-many links, since to-one links keep the candidate sets small. If a to-one link is available, activity 6 generates two statements. The first statement employs the appropriate get method that traverses the to-one link and assigns the result to the variable of the reached object. In Example 3.24 (part a) the traversal of the products to-one link from seat to supplier created line 27 of the Java code. Variable seat was already bound and method getVendor() traverses the products link against its default direction. The result is assigned to variable supplier. Note, our approach does not support lower association cardinalities. A to-one link has actually cardinality 0..1. Instead of a valid neighbor object, the traversal of a to-one link may also return null. Thus, we generate a second statement that validates whether the traversal was successful. This is done using a JavaSDM.ensure() statement with a variable!=null condition, cf. line 28 of Example 3.24 (part a).

If it is not possible to extend the current set of bound variables via a to-one link, activity 7 of our code generation algorithm tries to find an appropriate to-many link. If we find such a to-many link, activity 8 generates a while loop iterating through all candidate objects reachable by the chosen to-many link. First, we use the corresponding iteratorOfXyz() method to retrieve an iterator that allows to loop through all objects reachable from the current start object, cf. line 18 in Example 3.24 (part a). Second, we generate a while loop with a combined termination condition. The while loop terminates as soon as Java variable sdmSuccess flags the successful execution of the whole story pattern or if the employed iterator runs our of candidates, cf. line 19 in Example 3.24 (part a). Third, the first line in the body of the while loop retrieves the current candidate object form the iterator, cf. line 20. Note, the result of method xyIter.next() is of type object. We need an explicit guard in order to be able to assign the retrieved object to the target variable. Forth, we generate a try-catch statement covering the remaining body of the while loop. Any exception thrown by a JavaSDM.ensure() clause within the while body will be caught by this try-catch statement. Thereby, a condition validation failure causes a new iteration of the while loop. If possible, another candidate is retrieved and the conditions are evaluated, again. Note, our while loop construction allows nesting of other while loops and of to-one link traversals and of all kinds of condition validations.

Activity 9 of our code generation algorithm just updates the book-keeping variables. The just reached collaboration variable is moved from the todoVariables to the boundVariables. Similarly, all kinds of todo conditions that have been considered are removed from the corresponding algorithm variables. Note, the binding of another collaboration variable may now allow the evaluation of new conditions. Therefore, we jump to activity 3, again, and try to generate code for as many conditions as possible.

This code generation loop terminates due to two reasons, if all collaboration variables are bound or if it is neither possible to find a to-one nor a to-many link that allows to bind a new variable. The latter case indicates an ill-formed collaboration diagram. We restricted story patterns to connected components where each component contains at least one bound collaboration variable. This condition allows us to identify all participants of a collaboration by just traversing links from known objects. We do not need to manage explicit class extensions, i.e. the sets of all instances of certain classes. Managing explicit class extensions is an expensive task since these sets easily become very large. In addition, a lot of care is necessary in order to deal correctly with the garbage collection mechanism. It must always be clear whether a given object should still belong to the extension or whether it should be extended. Our approach avoids these problems by omitting explicit extensions. This is paid by the restriction of story patterns to connected components with bound objects. Due to our experience this restriction is not problematic in practice. One easily finds some common root object that may be used as starting point for the identification of other collaboration participants.

If the collaboration diagram consists of connected components with at least one bound object, the main loop of our code generation algorithm will terminate when all collaboration variables are bound. Note, if all collaboration variables are bound, all conditions are enabled and thus activity 3 has generated validation clauses for all (remaining) conditions. We now reach activity 10 of our code generation algorithm.

Activity 10 of our code generation algorithm handles optional variables. Note, according to the semantics of collaboration diagrams an optional variable is just an extension of the core story pattern part. If no candidate for an optional variable fulfills all required conditions, then the optional variable is just omitted, the story pattern itself is still successful and there is also no need to search for alternative candidates for the core collaboration variables. Our approach does not try to find some kind of "maximal" match. It just finds a match for the core parts of a story pattern and sticks with it, no matter whether the optional parts can be extended or not. To find candidates for optional variables, we use steps similar to activity 5 to 8 of our code generation algorithm followed by activity 3. Note, since an optional variable may get value null, all later usages of that variable has to validate whether it actually contains an object.

Similarly, activity 11 of our code generation algorithm handles multi-object variables. Again, multi-object variables do not affect the success of the whole story pattern but they are just possible extensions of its core parts. For multi object variables special code is generated that collects all valid candidate. The identification of a single candidate is done similar to usual collaboration variables. However, if a search loop finds a valid candidate for a multi object variable that candidate is added to the corresponding collection set and the search just continues. Note, modifications on multi-object variables affect all valid candidates. Thus, such modifications require another loop through the corresponding collection set.

Now all participants of the collaboration are identified. In the next steps activities 12 and 13 generate the code for the execution of the story pattern effects. Code generation for the object structure modifications is straight forward. One visits just one deletion, creation, and attribute change after the other and generates the corresponding code. For deletions of objects we employ the removeYou() method generated for all classes from the class diagram. This method isolates the corresponding object such that it becomes garbage collected. For the deletion of links we employ the corresponding setXy or

removeFromXy method. Creation of objects is done using the plain parameterless constructor that we provide for each class. For attribute assignments appropriate access methods are employed. Note, if the computation of a new attribute value involves access to removed objects, this access is still possible. Although we have called method removeYou on to-be-deleted objects, the collaboration variables still refer to these objects and thus they can still be accessed. However, one should be careful, since the removal of all neighbors may have side effects on attribute values.

Code generation for collaboration messages is straight forward either. The numbering scheme and possible condition and iteration markers clearly identify the control flow of the collaboration messages. The message target is identified via the corresponding message arc and in simple cases the message itself may just be copied to the target Java code.

Note, the last two steps require special care for optional and multi-object nodes. Access to optional nodes always has to check whether the corresponding variable contains a valid object. Operations on multi-object variables need to iterate through all detected matches. For details cf. [FNT98, FNTZ98].

Finally, activity 14 of our code generation algorithm generates an assignment for the Java variable sdmSuccess that flags the successful execution of the whole story pattern, cf. line 46 in Example 3.24 (part a). This causes the termination of all surrounding search loops and signals the successful execution for possibly following branch statements. Note, for iterated story patterns we just omit the assignment to variable sdmSuccess. In this case, after one execution of the story pattern, the search for alternative matches continues and each time a match is found the effects are executed, again. The execution stops when all search loops run out of candidates.

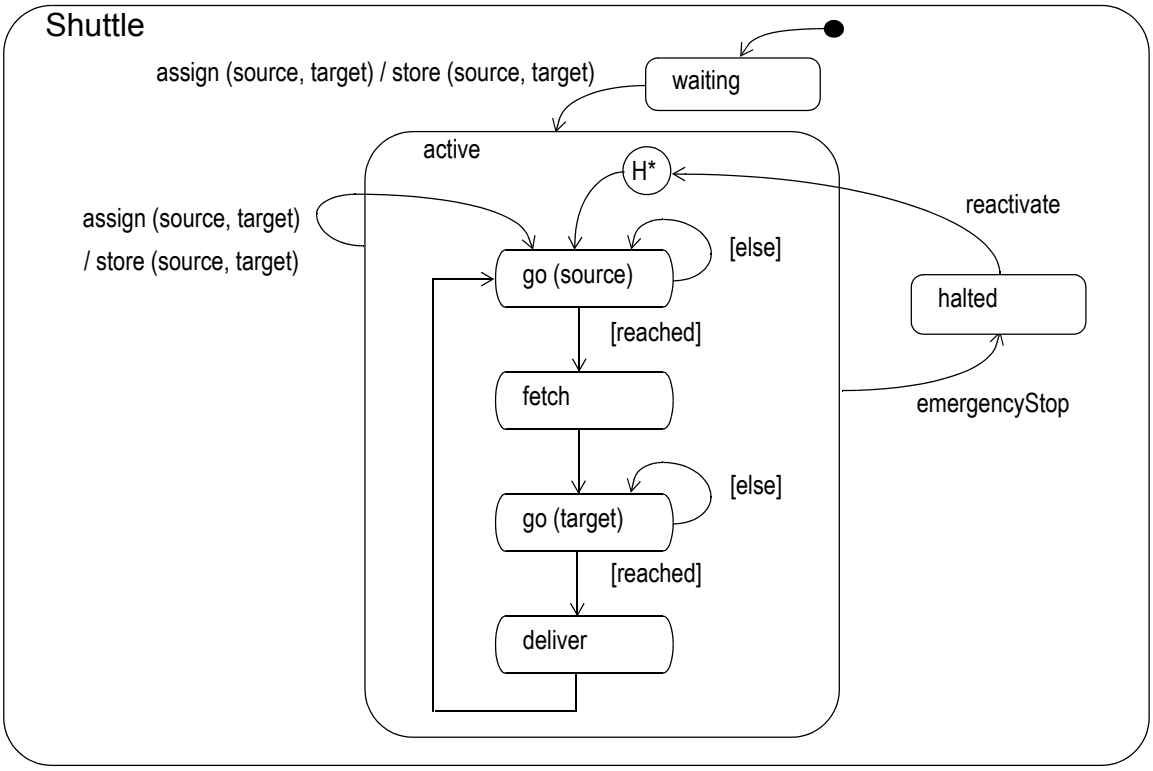
To summarize, in terms of constraint satisfaction theory our code generation strategy for collaboration diagrams is not very sophisticated. We just generate a "simple" backtracking algorithm searching through the search space in some kind of depth-first manner. However, at least we are able to generate an automatic execution of collaboration diagrams. In addition, the generated code is light-weight, since is does not require extensive library overhead as e.g. an objectoriented database. The resulting code is quite human readable which has advantages for maintenance and especially for debugging on source code level. Due to our approach the runtime efficiency of the generated code is OK. In practice most operations work quite locally and provide sufficient information in order to identify the collaboration participants easily. If a search loop is generated, usually one actually has to search through the corresponding set to access a certain element. If this creates an efficiency problem, this is easily resolved by using e.g. a sorted association or a qualified association or some other auxiliary data structures. Using collaboration diagrams to denote the cut-out of the object structure one is going to work which frees the developer from a lot of tedious and error prone coding work compared to a direct implementation in Java. Even more important, a collaboration diagram is much easier to read and to understand.

3.4 Statecharts

Statecharts exist for quite a while now and thus a number of CASE tools provide code generation for them. One of the first CASE tool supporting the execution of statecharts was Statemate [H+90]. Currently, Rhapsody [Rhap] and Rational Rose RT [RR-RT] are well known CASE tools supporting statecharts. Originally, statecharts stem from finite automata theory. Finite automata are used with great success in compiler construction. Thus, a lot of implementation ideas for statecharts can be borrowed from that field. We will discuss three general approaches to the implementation of statecharts. First, the translation into switch-case statements which is employed e.g. by Rational Rose RT. Second, an implementation based on the state design pattern [GHJV95] which is employed by the Rhapsody envi-

ronment. And third, a state table driven approach based on [Dou98, KNNZ00] which is employed in our approach.

Example 3.28: A simple statechart for shuttles



Example 3.28 shows a simple statechart for class Shuttle which will serve as a running example for the discussion of statechart implementation concepts.

Implementation Strategy 1: switch-case statements [BJR99, RR-RT]

The basic idea of the switch-case statement approach is to employ a simple int attribute currentState to store the current state. Each event that could be send to an object is implemented by its own method. The event implementation methods contain a switch-case statement that branches on the content of attribute currentState. Within the different case branches, the specific behavior of the active object is implemented. If an event fires a transition, the corresponding switch-case branch contains code for the actions to be executed and code that assigns the new state number to attribute currentState.

Example 3.29 shows a possible implementation for the event method assign of our statechart example. Method assign may be called in different states. If attribute currentState refers to state waiting or to one of the substates of state active, method assign executes the store action attached to the corresponding assign transition and changes attribute currentState to go_source in order to reflect the state change and executes the do action of the new state, i.e. it calls method go(source). If attribute currentState refers to state halted, method assign does nothing. This reflects that our example statechart ignores assign events while it is in state halted.

Note, Example 3.29 shows a very naive implementation of method assign. More sophisticated implementations could exploit the similarities of different states. For example all substates of state active behave similar. One could also use multiple currentState attributes to store not only the current leaf state but also its parent state. This would allow to employ nested switch-case statements that reflect the nesting structure of or-states in the statechart. A similar approach allows the handling of and-states. For history states one may employ another state attribute that stores the last state that has been left. Sometimes, the target of a transition is computed from multiple guards or from a history state. In

this situation, the execution of the do-action attached to the reached target state may be handled by another subsequent switch-case statement.

Example 3.29: Switch-case version of method assign.

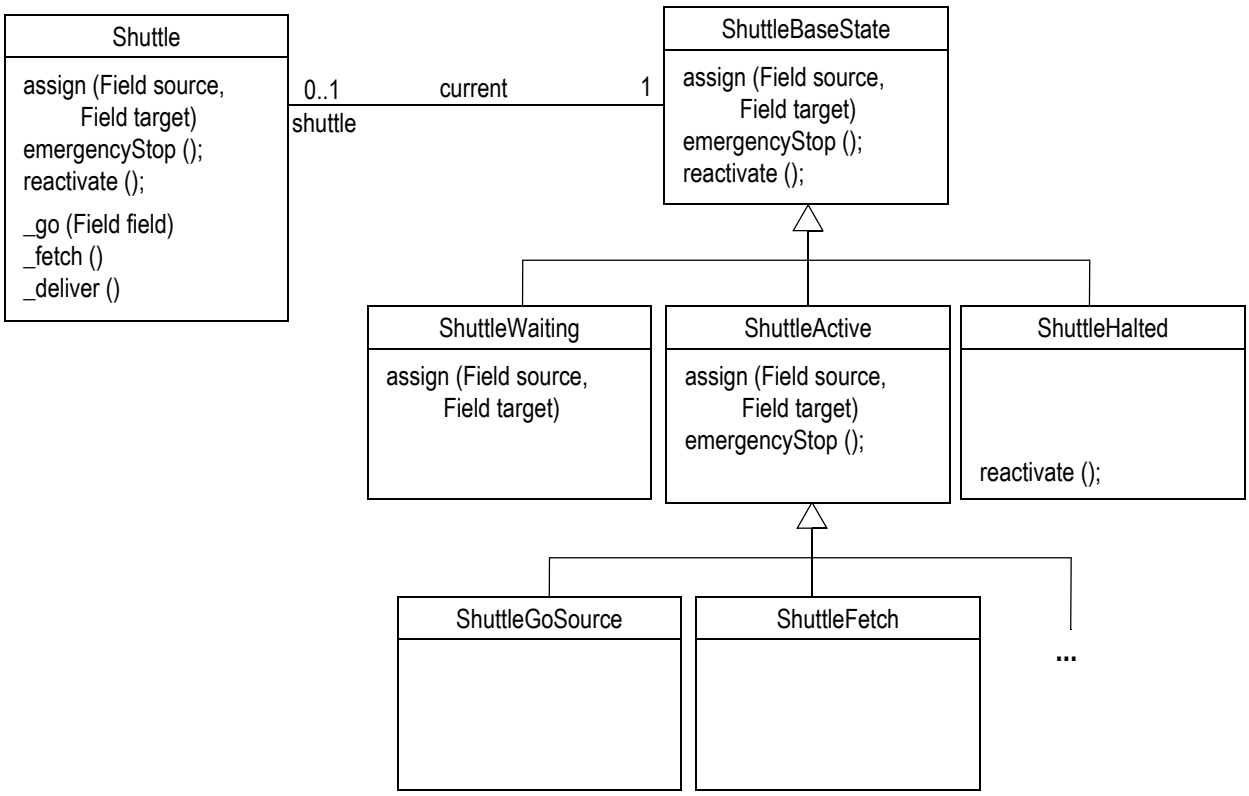
```
----------file Shuttle.java----------
...
public void assign (Field source, Field target) { ...
  switch (currentState) {
  case waiting:      store (source, target);
                     currentState = go_source;
                     go (source);
                     break;
  case go_source:    store (source, target);
                     currentState = go_source;
                     go_source ();
                     break;
  case fetch:        store (source, target);
                     go_source ();
                     break;
  ...
  case halted:       break; // ignore
  ...
```

In principle, the implementation of statecharts via switch-case statement is straight-forward and causes only minor execution overhead. According to the complex semantic details of statecharts, the resulting switch-case constructs may become very complicated, too, and from the tool builders point of view, sophisticated code generation becomes a challenging task. However, for simple statecharts this approach is easy to follow, e.g. in manual implementations.

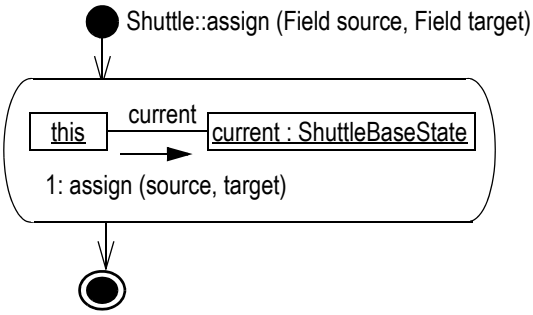
Implementation Strategy 2: State Pattern, [Rhaps99]

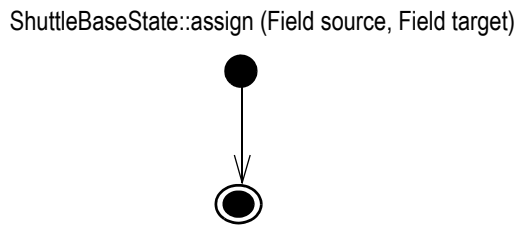
In objectoriented approaches large switch-case structures are a no-no. They indicate situations, that should be solved using inheritance structures. Accordingly, an alternative approach for the implementation of statecharts is based on an objectoriented approach following the idea of the state design pattern proposed by [GHJV95]. Basically, we create one class for each state in the statechart. If a state A is a substate of another state B than the corresponding class A becomes a subclass of class B. The whole inheritance structure is rooted by an additional base class. Events are turned into methods. The methods are declared in the additional root class where they are implemented with an empty method body. If a transition ev leaves a state C, then the corresponding class C redefines method ev to behave as described by the transition. At runtime, each shuttle object is accompanied by an additional state object representing its current state. The current state is attached to the shuttle object via a link of type current. Calls to event methods are forwarded to the current state object. The current state object reacts on this event according to its specific implementation of the event method. In addition, it may replace the current state object by another state object in order to reflect a state change.

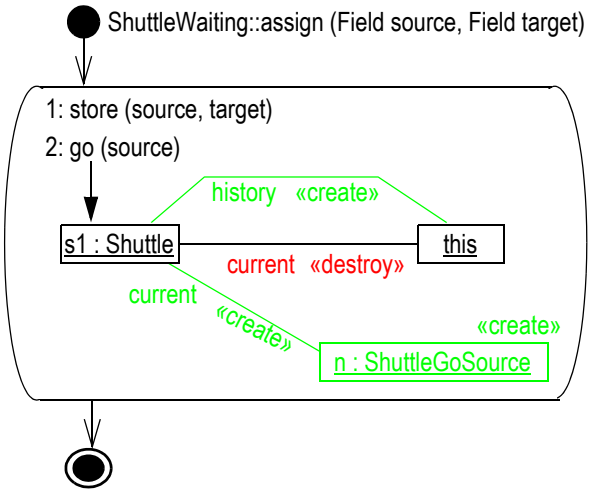
Example 3.30: Class hierarchy modeling the statechart of Example 3.28

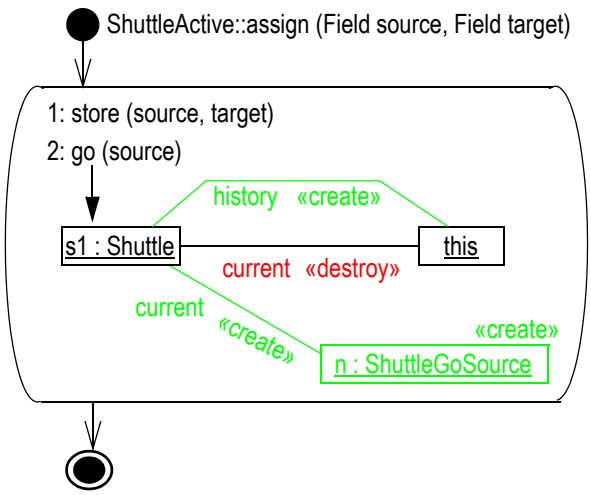


Example 3.31: Method assign for the different state classes.









Example 3.30 shows the class and inheritance structure derived from our example statechart. Class ShuttleBaseState serves as root for the inheritance structure. Our example statechart has three top-level states waiting, active, and halted which are turned into classes ShuttleWaiting, ShuttleActive, and ShuttleHalted, respectively. These classes become direct subclasses of class ShuttleBaseState. States go(source), fetch, go(target), and deliver are substates of state active. Thus, the corresponding classes become subclasses of class active. Class ShuttleBaseState provides methods for all events employed in our example statechart. In class ShuttleBaseState all these event methods are implemented with an empty method body. This models the default behavior of a statechart for events. If no explicit reaction is specified, the event is just ignored. If an explicit transition leaves a state, the corresponding event method is redefined, accordingly.

Example 3.31 shows the different redefinitions of event method assign. First of all, in class Shuttle itself, any call to event method assign is just forwarded to the state object attached to the shuttle object via a link of type current. Note, the current association has static type ShuttleBaseState. Thus, any of our state classes may be attached to a shuttle as current state. If the current state object is of type ShuttleWaiting, the dynamic binding of method calls in objectoriented languages causes the execution of the specific implementation of method assign in class ShuttleWaiting. This implementation of method assign in class ShuttleWaiting is shown as a story diagram in the lower left quarter of Example 3.31. As depicted, collaboration message 1 executes the transition action store(source, target) and collaboration message 2 executes the do-action of the new current state, i.e go(source). In addition, the current state this is replaced by the new state n of type ShuttleGoSource. Class ShuttleGoSource represents the actual target state of the assign transition leaving state waiting. In addition, the former state this is marked by a history link. This history link allows a simple implementation of transitions to history states.

Classes ShuttleGoSource, ShuttleFetch, ShuttleGoTarget, and ShuttleDeliver do not redefine method assign. Instead, the assign transition leaving state active is implemented by an appropriate method in class ShuttleActive, directly. This implementation of method assign is then inherited by all subclasses of class ShuttleActive, i.e. by all classes that represent substates of state active. Thus, if the current state of a shuttle is e.g. of class ShuttleFetch, the assign implementation of class ShuttleActive will be executed. Basically, a transition is implemented at the nesting level it is leaving in the statechart.

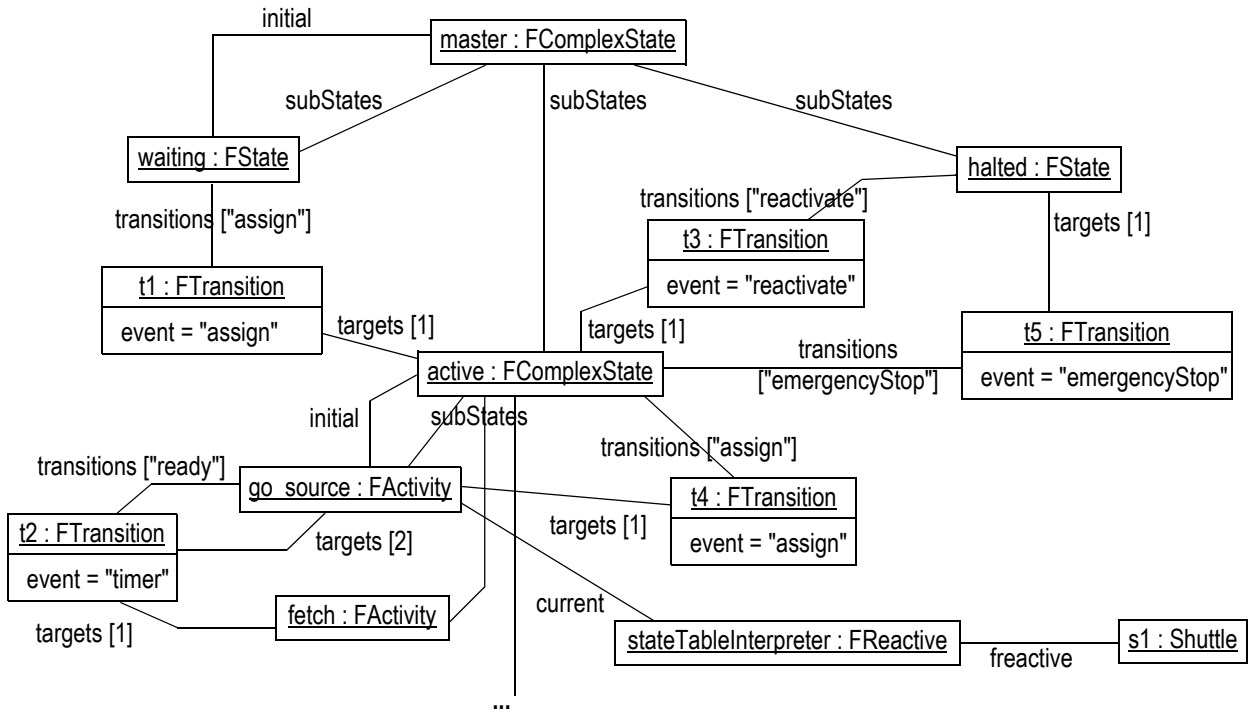
Multiple current links could be used to deal with and-states. Then, an event is forwarded to multiple current states and it is handled in each "parallel" substate, separately. Note, depending on the actual implementation this may imply a sequential execution of the and-substates where the actual sequence may have semantic relevance. In our approach we would address this problem with the explicit user defined order of substates.

Generally, the state pattern based implementation of statecharts is very elegant. It uses the classical objectoriented approach to replace complex switch-case statements. The statechart structure is reflected in the class hierarchy. Each transition is implemented by its own single method. Entry and exit actions could be implemented by special methods redefined at the corresponding inheritance level, too. Using the super.exitAction() construct of Java, one could easily call exit-actions at different nesting levels. Entry and do-actions may be handled, similarly. The forwarding of events from the original object to its current state object causes only minor runtime overhead. Instead of creating new states, all states could be stored in a look-up table where they could easily be retrieved, on demand. A little problem might be that this approach creates a lot of very small classes with only some method redefinitions and a complex inheritance structure. This may be a little bit complex for code maintenance. From the tool builders point of view, still a lot of code needs to be generated.

Implementation Strategy 3: State Table [Dou98, KNNZ99]

A well known approach to the implementation of finite state automata stems from the field of compiler construction. The idea is to use a state table that stores all state specific information including the code to be executed. An index to the state table is used to look up the code to be executed. The code is executed and a successor index is computed representing the new state. Our approach follows this idea. We have been inspired by a similar approach described in [Dou98]. Actually, our state tables use exactly the object structure proposed in Appendix A.8. This state table object structure is then interpreted using the same story diagrams that have been used in Appendix A.8 to define the semantics of our statecharts. In a kind of boot strap approach, we have used the Fujaba environment to translate the story diagrams defining the semantics of statecharts into a Java implementation of Fujaba runtime library functions. These runtime library functions are able to interpret state table object structure that correspond to Definition A.59 or to Definition A.63, respectively. Our code generator just generates an initialization routine that creates the state table object structure representing the statechart of a class. This initialization routine is then called from the constructor of active objects. It creates the state table object structure and adds a so-called FReactive object to the active object. The FReactive object is a small state table interpreter. The FReactive object runs its own thread. It employs its own event queue. The original object is equipped with event methods that just create event objects and enqueue them to the event queue of the corresponding FReactive object. The thread of the FReactive object listens to the event queue. If an event is pending, the FReactive object consumes it and visits the state table object structure in order to execute the event as described in Appendix A.8. Example. Example 3.32 shows a object structure representing a cut-out of the state table for our example statechart.

Example 3.32: Object structure representing our example statechart in Fujaba



From a tool builders point of view, the state table approach needs only little code to be generated. Most of the nasty semantic details of statecharts are encapsulated in the state chart interpreting library functions. The code generator merely has to generate the event methods that forward the events to the FReactive object and the state table initialization routines. In addition, the actions contained within the statechart are turned into usual small help functions. Since the whole complexity of the statechart

semantics is handled by the runtime library functions, improvements and bugfixing of the event handling routines requires changes of the library functions, only. It is not necessary to change the code generator. This saves us one level of indirection and facilitates the implementation work. In addition, the state table approach allows modifications of the state table structure at runtime. This may be interesting for long running programs like servers that must not be stopped for software updates but software updates have to be done in the running environment.

However, these advantages are paid by a certain runtime execution overhead. The state table structure is merely interpreted and interpretation is usually an order of a magnitude slower then the direct execution of compiled code. In addition, we use Java runtime type information and the Java Reflect API to access the small help methods that implement transition guards and entry, exit and do-actions. This look-up of runtime type information and especially the invocation of help methods via the Java Reflect API is relatively slow. However, we did not yet measure the execution speed differences and thus we do not know whether they are significant.

4 From Analysis to Design

<<This chapter will revisit the transition from sample scenarios to formal behavior specifications. This is based on the ideas of Tarja Systä and the SCED and MAS environment. We will try to extend the ideas of Tarja Systä towards story boards and story diagrams.>>

4.1 From Scenarios to Class Diagrams

4.2 From Scenarios to Test Specifications

4.3 From Sequence Diagrams to Statecharts

4.4 From Story Boards to Story Diagrams and Story Charts

5 CASE Tools

Story Driven Modeling (SDM) employs major parts of the UML. In contrast to the UML, in SDM the different diagrams do not stand on their own as unrelated description parts. SDM integrates all its sub languages and sub descriptions to a complete and consistent overall model, cf. Figure 1. Each use-case is refined either by another use-case diagram or by one or more story boards. All elements employed in story boards like object(kind)s, attributes, and messages are finally declared in the class diagrams. The behavior of each active class is modeled by exactly one story chart covering exactly the events declared in the signal department of that class. Each object, attribute, and action employed in the story chart is declared in the class diagram. Each method declared in the class diagram is for-

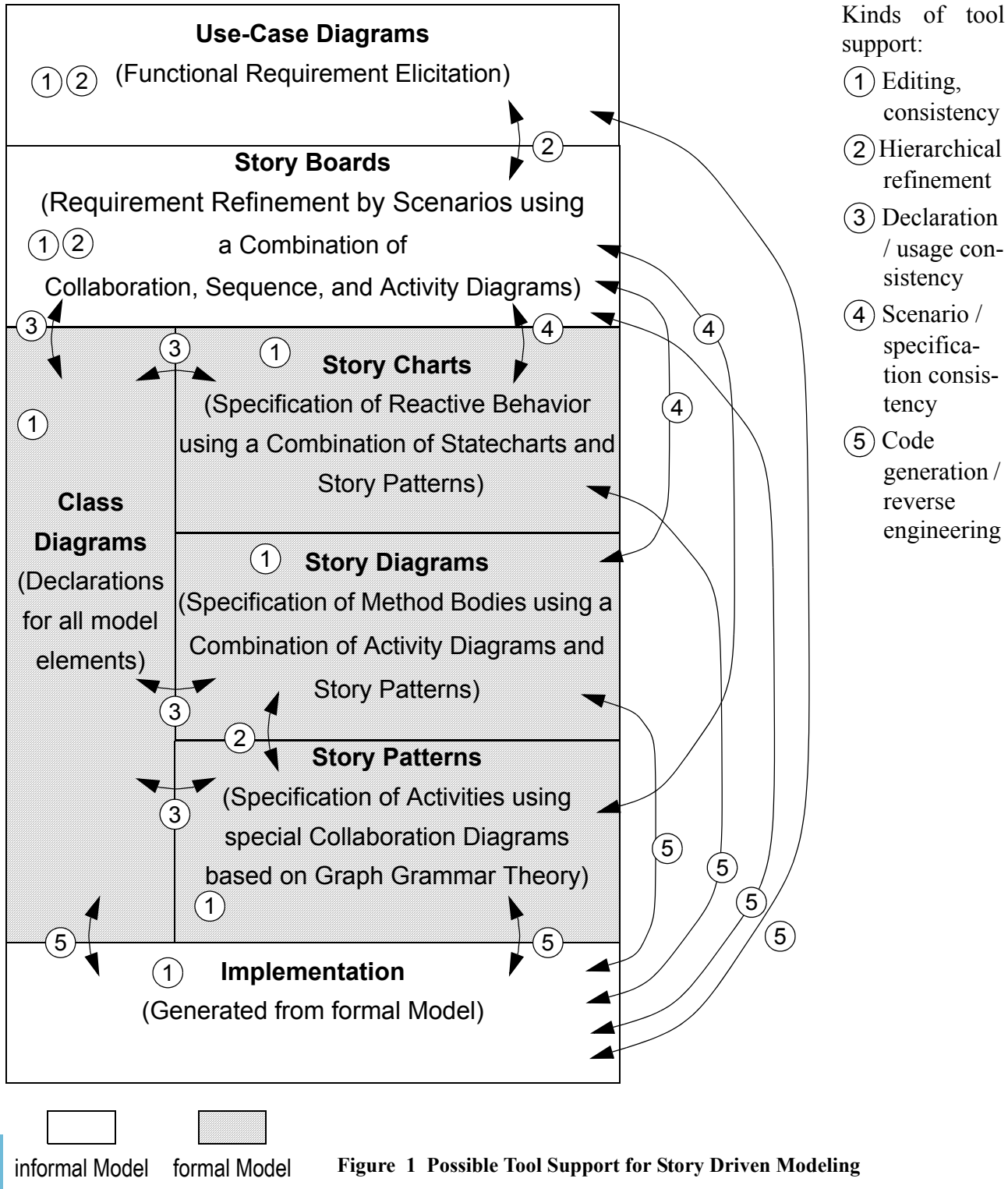


Figure 1 Possible Tool Support for Story Driven Modeling

mally modeled by exactly one story diagram. Again, all story diagrams employ only objects, attributes, and methods provided by the class diagrams. Scenarios described by story boards describe valid execution traces for the story charts and story diagrams that model the exact system behavior. The application code implements the structure described by class diagrams and the behavior modeled by story charts and story diagrams, correctly and robustly. This organization of specification parts allows many context sensitive compile time checks enforcing the consistency and completeness of the overall model. In addition, many refinement and realization steps may be supported by tools.

In this chapter, we will discuss possible tool functionalities supporting our approach. For each desired functionality, we will evaluate a number of existing UML CASE tools. In addition, for some features we name exceptional tools that are not intended as general UML tools but that provide special useful functionality that should be incorporated in UML tools. Our UML tool evaluation focuses on Rational Rose RT [RR-RT], Rhapsody [Rhap], and TogetherJ [Toge] as general industrial UML tools. In addition we consider Argo, Sced and Fujaba as prototype UML tools stemming from research projects. For a more general overview of available UML tools cf. [Jeckle].

Disclaimer: We have tried each tool that we are going to evaluate and we have tried tutorials and available documentation and we have discussed our evaluation with some long term users of the corresponding tools. However, CASE tools are moving targets for such an evaluation. It might happen that some tools have new and additional features in new releases that become available even before this work is published. In addition, some tools may have some features and we were just not able to find or to invoke it. For example, code generation from UML diagrams frequently requires certain consistency conditions for that diagrams and the functionality is just not offered, if the diagram is not created in a correct way. Some times, the option just needs to be enabled at tool installation.

5.1 Evaluation of available UML tools

In detail, sophisticated CASE tools should provide the following support for story driven modeling:

(1) Editing and per diagram consistency.

Of course, a CASE tool should minimally allow to edit the different kinds of diagrams employed in UML and in our approach. Editing support for a single UML diagram should also include basic consistency checks guaranteeing conformance with the UML meta model. Basically, this means that a use-case diagram may employ only use-case diagram elements and that e.g. there may be no usage relationship between user roles. Similarly, basic context sensitive constraints should be enforced. For example, a tool should check for name clashes in all name spaces.

Most existing UML CASE tools provide satisfactory editing support for almost all kinds of UML diagrams. This holds especially for the UML tools considered in this evaluation. However, if one looks for editing support, only, he or she may consider the usage of an adaptable drawing tool, like Visio or even Microsoft Powerpoint. Compared to UML CASE tools such general purpose drawing tools frequently have more convenient user interfaces and more sophisticated support for printing diagrams or for the embedding of UML diagrams into text documents or for presentations of diagrams e.g. during a talk. In addition, most UML CASE tools restrict the user to stick to the UML meta model. While this is desirable if the diagrams are input for subsequent analysis and generation steps, if only an illustration is required, one may want to employ additional graphical elements like general images or anotations. Only if the tool provides substantial consistency checking support and if the derivation of subsequent system parts, e.g. code, is desired and possible, the usage of a CASE tool pays the effort.

The Argo environment represents an exceptional example for tool support for inner diagram consistency checking. The Argo environment follows a new interactive concept. During diagram editing,

the consistency and completeness of the diagram is tested on a logical level. For example, a just created class is directly marked as incomplete, since a class without attributes, methods, and associations makes no sense. If only static class features are added, Argo automatically suggest to turn the class into a Singleton in order to improve the overall design. If a new statechart is created, Argo automatically flags that an initial state is missing. States without outgoing transitions are spotted. Branch states with only one outgoing transitions are marked. All incomplete and inconsistent modeling elements are prioritized and combined to a todo list for the whole specification. This todo list guides the designer through his work and reminds him on not yet finished modeling tasks. Overall, the consistency and completeness checking facility of Argo points into a promising direction. Unfortunately, the evaluated version 0.8.1 from October 2000 does not yet provide sophisticated cross diagram consistency checks. For example, for a statechart transition it is not checked whether the corresponding event and action methods are declared in the corresponding class. However, the concept could easily be extended to such checks and then it will provide extremely valuable modeling support.

② Hierarchical diagram organization

A hierarchical organization of the whole specification as proposed by SDM means that each subdiagram of the overall specification has its specific logic position and task. For example, a use-case diagram is either the root diagram or it refines a certain use-case of its parent use-case diagram. Each story board either refines a certain use-case of a parent use-case diagram or a certain activity of a parent story board. Each statechart is either the (one and only) specification of an active class or the one and only specification of some complex state contained in its parent statechart. Each story diagram defines exactly one method body.

If at all, existing UML tools allow to organize the different diagrams in packages. While this allows a hierarchical organization of diagrams, this organization is left to the user. It is not driven by the content or logical structure of the diagrams as defined in SDM. Thus, it is possible to organize e.g. use-case diagrams such that they represent a use-case refinement hierarchy, but this is neither enforced nor supported by the tools. Accordingly, it is difficult to maintain the consistency of the diagram hierarchy. Most tools do not identify use-cases that have not yet been refined. The same holds for scenario diagrams. To be honest, this feature is still under development for the Fujaba environment, too.

Hierarchical diagram organization is not yet supported by today´s UML tools, since this kind of diagram organization is not yet commonly accepted in the UML community. No tool allows to check whether all use-cases have been refined by either some hierarchical decomposition or by some scenario diagrams. We hope that this will change in the near future. First, it is relatively simple to realize this kind of support. Second, the benefits will be tremendous. Third, UML users are desperately seeking for cooking receipts guiding them in using the UML. Our hierarchical diagram organization is a big step forward in this direction. Based on this idea, loosely coupled sets of diagrams are turned into closely connected building blocks forming a common overall picture. Each diagram has its dedicated place. Each diagram models its specific aspect of a specific system element.

③ Declaration / usage consistency

In a consistent system specification, the class diagrams should provide declarations for all kinds of elements employed in scenarios, statecharts, and method body specifications. For each kind of object there should exist a class declaration. For each employed attribute the corresponding class has to declare a corresponding data member. For each link employed in a scenario or in a story pattern, an appropriate association must exist. Method and signal declarations are provided for all kinds of method invocations and messages in the behavior descriptions. Based on this declaration / usage consistency, a CASE tool may perform numerous context sensitive consistency checks. This guarantees the uniform usage of all model elements. Sophisticated analysis operations may even detect object

collaborations that are no valid extensions of the given class diagram, e.g. due to the violation of association cardinality constraints.

Note, the Declaration / usage consistency is not just restricted to error detection. It is also possible to employ this consistency for automatic derivation steps. For example, a tool could analyse multiple collaboration diagrams and propose an appropriate class diagram that covers all employed scenario elements. The user may improve this derivation process by providing meaningful names for class diagram elements.

Frequently, the scenario diagrams and the class diagrams are developed in parallel. If some parts of the class diagram exist already, the UML tool could support the creation of scenario descriptions by proposing possible types, attributes, and methods during the extension of a scenario. For example, if a new message is going to be send to some object `o1`, the tool could derive the set of all known messages understood by the target object. The user could either choose one of these methods or he could introduce a new kind of message to be used in this collaboration. This new message kind could automatically be added to the corresponding class.

Finally, the declaration / usage consistency may also be exploited for completeness checks. As well as any behavioral element must have a declaration in the class diagram, each class diagram should be employed in some behavioral diagram, otherwise it is useless.

The correspondences between class diagrams and behavior diagrams is quite well understood. Most modern CASE tools cover this functionality quite well. Usually, UML tools provide two options for behavior diagrams. First, the behavior diagram may be intended as an early outline of ideas. In this case, the tools do not enforce declaration usage consistency, but the user may introduce scenario elements without already worrying about a consistent declaration. Second, during later phases scenario diagrams may be intended as valid behavior specifications that may even serve as input for code generation steps. In this case, the declaration / usage consistency is enforced by the tool. Rational Rose RT [RR-RT], Rhapsody [Rhap], and TogetherJ [Toge] offer both opportunities. Argo does not yet support cross diagram consistency checks. In Fujaba all diagrams contribute to code generation. Thus, Fujaba enforces the declaration / usage consistency, always. In some cases Fujaba does not even allow to create scenario elements if no declaration is chosen from the class diagram. A more relaxed support for scenario diagrams in earlier phases is current work.

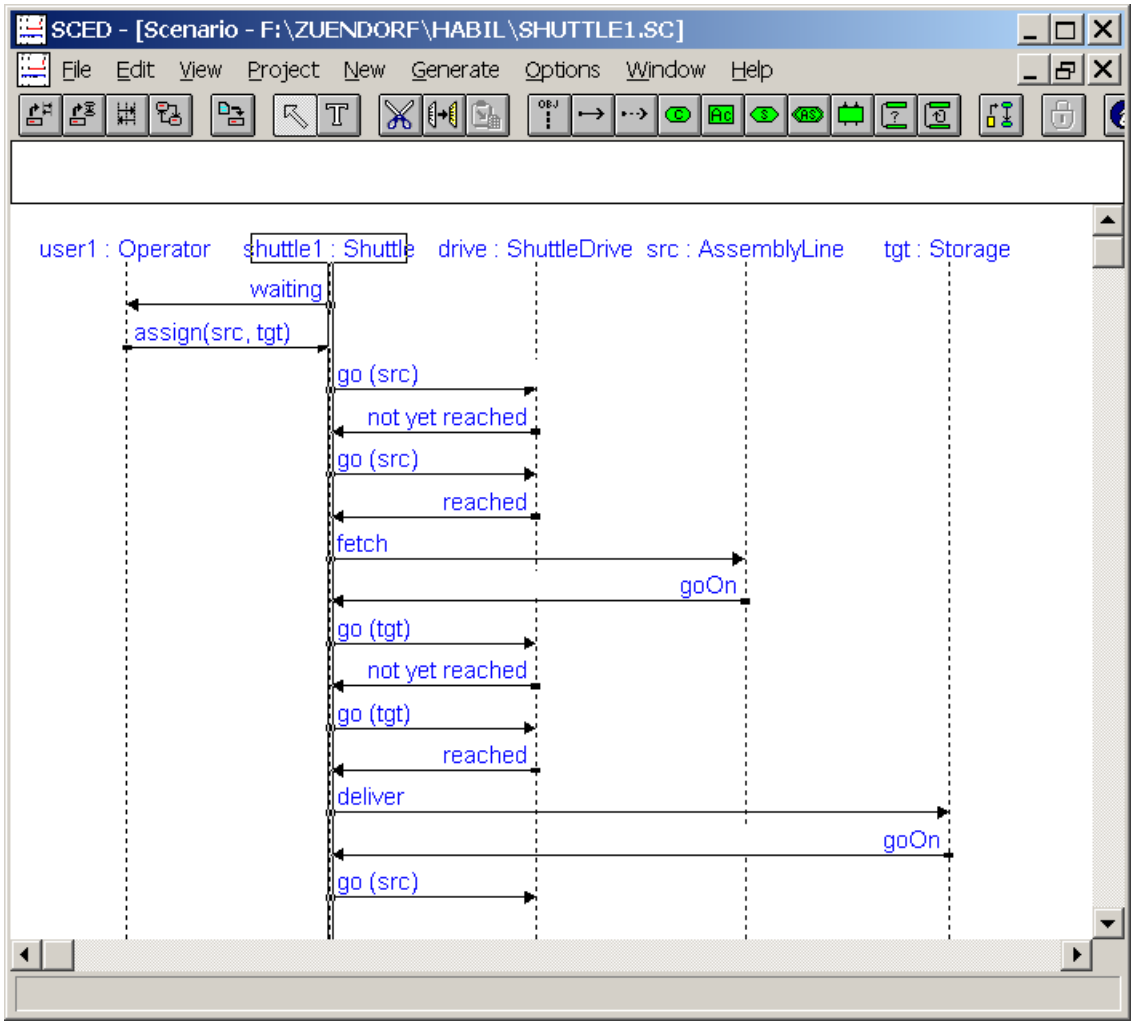
④ Scenario / specification consistency

Scenario diagrams or story boards describe possible executions for certain use-cases. Multiple scenarios for a single use-case may outline alternative system behavior depending on certain conditions. If the set of scenarios provided for a single use-case is exhaustive, i.e. if all possible alternative execution passes are enumerated, then all scenarios together actually represent a full behavior specification. Thus, it should be possible to derive the statechart / story chart or the method body specification for the corresponding use-case implementation, automatically. In practice, the problem is a little bit more complicated. Usually, a single scenario employs not just one object and method but multiple objects of different types and multiple different methods are involved. Conversely, a single method may be employed in multiple scenarios. In addition, the same method invocation may exhibit different behavior in different scenarios. In order to derive a valid specification of a method body from all usages of this method in different scenarios, a tool must combine the different behaviors to a consistent behavior specification for the method body.

Due to our knowledge, Sced [Sced, MS00, MS01, Sys97, KMST94] is the only tool that provides such a functionality. Sced provides sequence diagrams and statecharts. Sced is able to derive statecharts for active objects from their example behavior in different sequence diagrams. To do this, Sced employs techniques known from compiler construction theory. Sced interprets the sequence of events received

and send by a single object within a sequence diagram as an example word of a certain formal language. Then, a grammar is derived describing a language that contains all these example words. Exploiting compiler construction techniques, this grammar corresponds to a finite automata recognizing the considered language. This finite automata is then turned into a statechart for the corresponding active class.

Example 5.1: The Sced environment showing sequence diagram 1

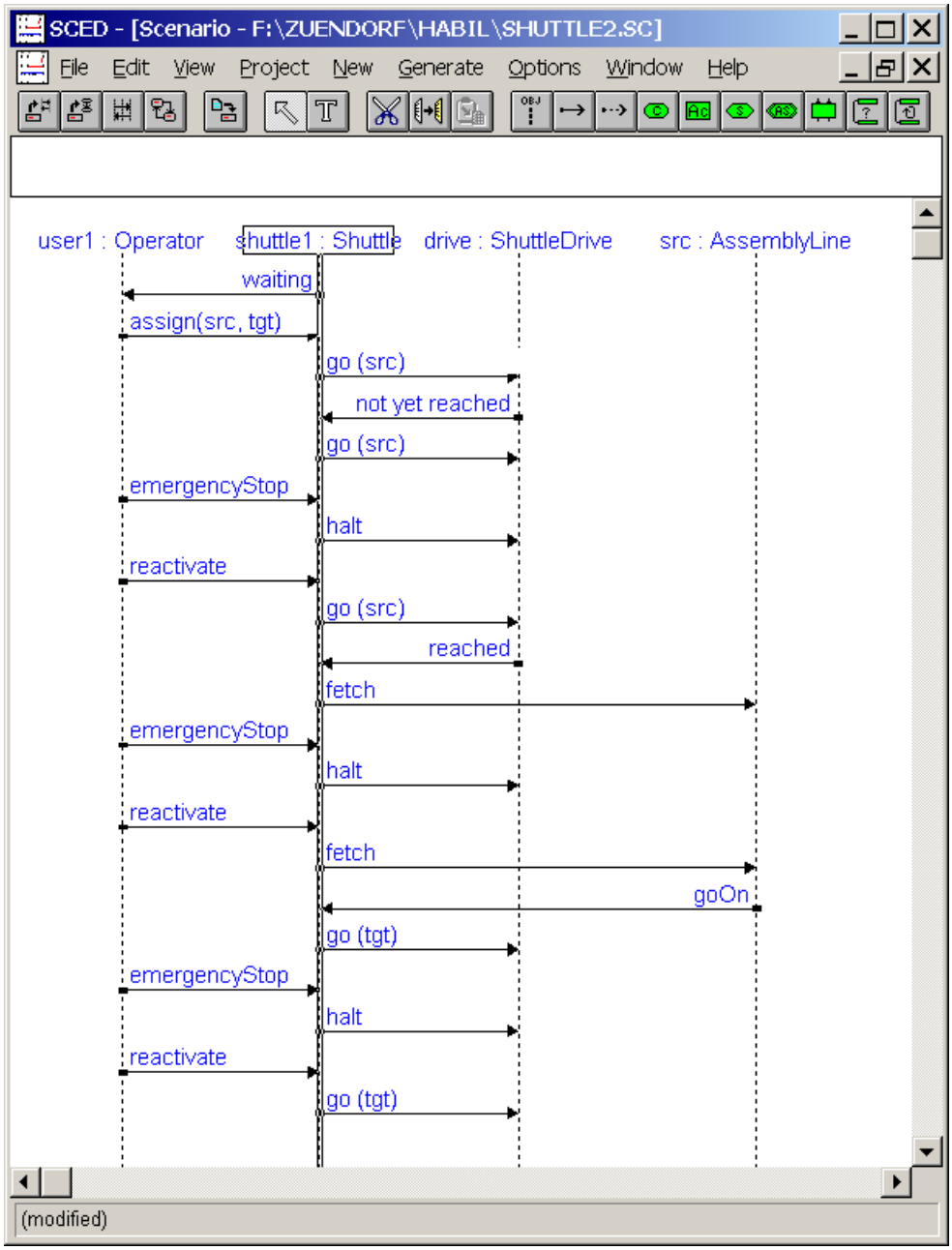


Example 5.1 and Example 5.2 show two scenarios corresponding to our factory example. Both scenarios involve a Shuttle object. The first scenario shows the usual behavior of a shuttle. Example 5.2 shows an exceptional behavior where the usual process is interrupted by an emergencyStop event. Example 5.3 shows the statechart synthesized from these two scenarios for class Shuttle. Confer this automatically derived statechart to the manually created statechart shown in Example A.57 on page 158.

Basically, the Sced approach works as follows: each message send from a shuttle object to some other object is considered as a basic action of class Shuttle. At a first shot, Sced introduces one state for each such basic action. Our example sequence diagrams start with a waiting message send from shuttle1 to object user1. After sending a waiting message, Sced assumes that shuttle1 is in state "waiting". In both scenarios, shuttle1 receives an assign(src,tgt) message and responds with a go(src) action. This behavior is turned into an assign transition from state "waiting" to state "go(src)". Next, a not yet reached message is received and shuttle1 again responds with a go(src) action. This is turned into a self transition from state "go(src)" to itself. In Example 5.1, the receiption of a reached event in state "go(src)" is answered with a fetch action. The assembly line responds with a goOn event turning

shuttle1 into state "go(tgt)". In Example 5.2, after sending a go(src) message the second time, the shuttle receives an emergencyStop event. It reacts by sending a halt message to its drive object. In the statechart this creates the emergencyStop transition from state "go(src)" to the top "halt" state. After receiving a reactivate event, shuttle1 sends a go(src) message to its drive, again. In the statechart a corresponding reactivate transition leads from the top "halt" state to state "go(src)".

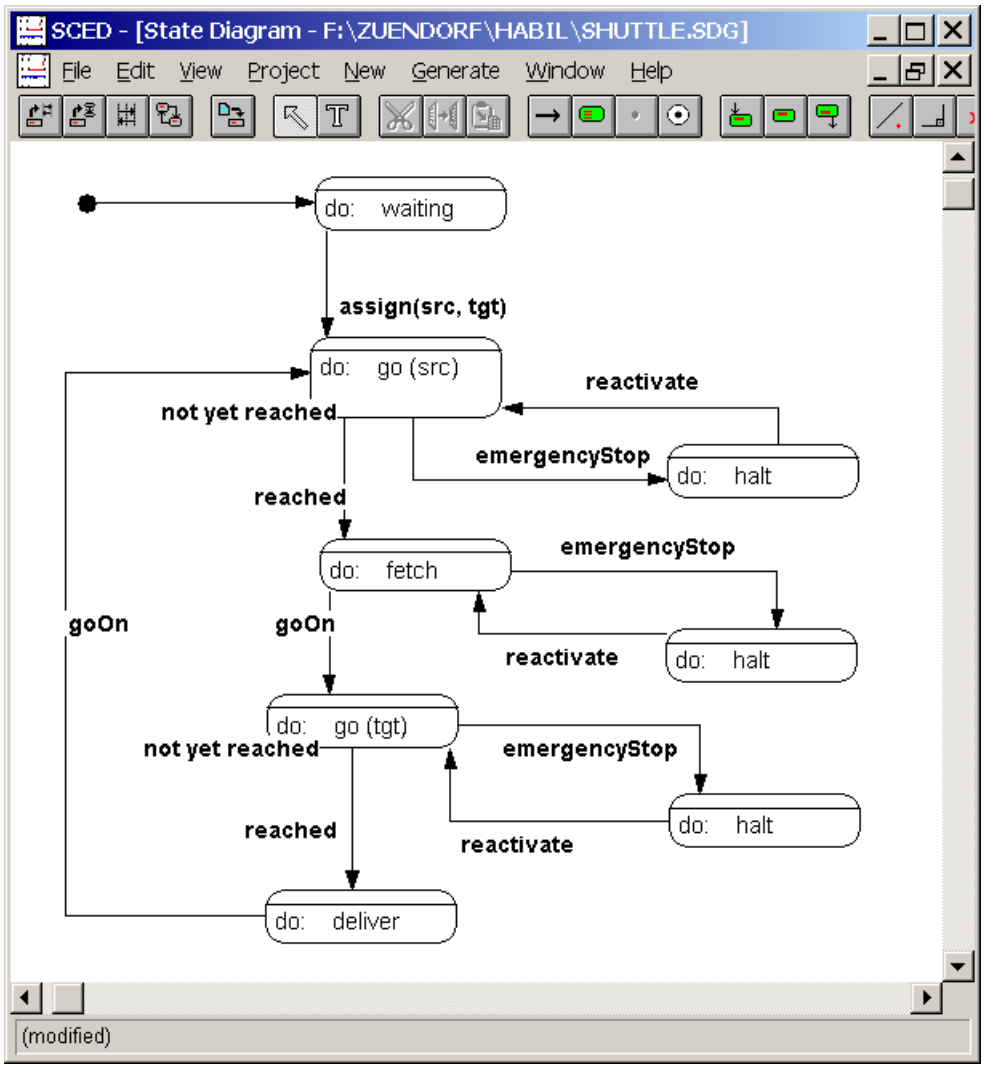
Example 5.2: The Sced environment showing sequence diagram 2



The basic idea of Sced is, that similar reaction corresponds to similar states. If at two different points in time one object executes the same action, it has reached the same state, again. This idea also allows to combine states and actions of different scenarios, easily. If the object behaves similar in two different scenarios, it has reached the same state in both scenarios. While this works fine in simple cases, in more complex situations, one object may exhibit different behavior on the receiption of the same event. In Example 5.2, shuttle1 answers the receiption of an emergencyStop event with a halt action, always. Thus, we would assume that in the statechart corresponding emergencyStop events should reach a single "halt" state. However, after receiving a reactivate event, shuttle1 reacts differently, each time. The first time, shuttle1 reacts with a go(src) action, the second time with a fetch action, and the

third time with a go(tgt) action. Sced resolves this conflict by assuming, that the three halt actions do not correspond to a single "halt" state. If an object is assumed to be in a certain state a second time, but it behaves differently than the first time, then there must be two different states performing the same action. Thus, Sced splits the corresponding state into two states with the same action. In addition, the transition that has led to the split state is redirected to the new copy. In this way, Sced has created three different "halt" states from the scenario of Example 5.2.

Example 5.3: The Sced environment showing the generated state chart



Note, in our example, splitting state halt has sufficed in order to resolve the behavioral ambiguity. In general, it may be necessary to split earlier states, recursively, until a different prefix in the event chain is reached that allows to differentiate between the behaviors. Note, Sced assumes that the reason for different behavior lies in the history of received events. In practice, the reason may stem from attribute values or other conditions.

Initially, Sced creates a flat finite automata. This flat automata does not yet employ the additional features of statecharts like entry-, exit-, and transition actions, complex or-states, and-states, and history states. In order to utilize these additional features, Sced provides a post-optimization mechanism that simplifies the initial flat automata. In Example 5.3, this post-optimization mechanism could combine the go(src), fetch, and go(tgt) state within a complex state and it could introduce a history state that allows to combine the three halt states into a single one.

Sced supports not only the derivation of statecharts from sequence diagrams. If some statecharts have already been created, Sced also supports the creation of sequence diagrams that represent valid traces

of the existing statecharts. During editing the sequence diagram, Sced evaluates the current states of the participating objects and on the creation of a new message, Sced proposes all events that the receiving object could now react on.

In addition to the Sced functionality, a sophisticated UML tool could provide even more scenario / specification consistency support. In SDM, one task of scenario diagrams is the description of test cases. A complete set of test cases should somehow cover the corresponding statechart behavior, "completely". A simple notion of statechart coverage could for example require that each state of the statechart receives each event kind at least once. A tool may ensure this statechart coverage by prompting not yet covered event / state pairs and by generating additional scenarios, that close the coverage gap, automatically. The incorporation of such a functionality into Fujaba is current work.

⑤ Code generation / reverse engineering

The final result of a system specification is the system implementation. A good UML tool should exploit class diagrams and all kinds of behavioral diagrams for code generation. Code generation for class diagrams is well understood and provided by nearly all tools and for various target programming languages. For object oriented models and programming languages, basically classes become classes and method declarations become method declarations and attributes become attributes, hopefully encapsulated with appropriate access methods.

Code generation for associations is not well supported. Most CASE tools just declare reference attributes within the connected classes. In case of to-many association roles, various kinds of container classes are employed and may be chosen by the user. As described in Example 3.7 on page 49 in chapter 3.1, the update methods for such association stubs should call each other in order to guarantee the consistency of the pairs of references implementing association instances. Unfortunately, most CASE tools do not generate such code. Instead, the client is responsible to keep the pairs of reference consistent that implement the bidirectional associations. Due to our experiences, keeping such pairs of references consistent, manually, is just not possible. Only the Rhapsody environment and our Fujaba tool support an appropriate code generation for associations guaranteeing reference pair consistency, automatically. TogetherJ does not support bidirectional associations at all. Rational Rose RT allows to encapsulate the association stubs, but the encapsulation methods do NOT call each other. This is a very serious drawback of TogetherJ and Rational Rose RT. Users have to invest a lot of unnecessary extra effort to fix this problem, manually. This bears a high risk, that complex object structures become unmanageable.

As already discussed, example scenarios or story boards may be considered as test cases for the final application. Accordingly, a tool may generate test methods from scenarios. Basically, scenarios define sequences of steps that represent (the) possible execution pathes of the corresponding class or method. One idea for code generation from scenarios are automatic black box tests as required e.g. in the eXtreme Programming approach, cf. [Beck99]. The expected sequences of execution steps could be turned in corresponding (string) constants. The method bodies could be extended by trace instructions that collect actual execution steps. Each scenario is turned into a test routine that creates an appropriate start situation, calls the corresponding implementation, and compares the resulting execution trace with the expected trace given by the scenario diagram. Such a functionality is not provided by one of the tools considered in this evaluation. Its development is current work within the Fujaba project.

Code generation for statecharts is quite well understood. Basically, statecharts correspond to finite automata and the corresponding approaches known e.g. from compiler construction techniques are applied. Rational Rose RT generates code that implements statecharts as if-then-elseif chains or switch-case statements. Rhapsody employs the so-called state pattern idea described by the "gang of

four" [GHJV95]. Fujaba uses a "jump table" based implementation, cf. [Doug98, KNNZ00]. Generally, each of these approaches does its jobs. However, one common problem with code generation from statecharts are the numerous tricky details in dealing with concurrency and possible nondeterminism. Although, statecharts have (multiple) formally defined semantics, the code generated by different tools may implement some other semantics. Such often subtle differences in the meaning of a statechart may have serious consequences e.g. if the generate code is used in embedded systems like cars or planes. A better standardization is necessary. In addition, the tools should prompt the user on problematic usages of state chart features.

Code generation concepts for activity diagrams require a precise description of the activities, themselves. Usually, activities are described by some natural language text or some pseudo code notation. For code generation, this informal description may just be replaced by program code of the target language. If all activities contain valid code of the target language, the activity diagram is actually turned into a usual control flow diagram. Control flow diagrams are known since the 70th and well understood. There exist techniques that analyse the structure of such a control flow structure and turn it into control flow statements of current programming language. The program code contained within the activities is then just embedded into these control structures. However, from the CASE tools evaluated within this chapter only Fujaba supports this kind of code generation.

Collaboration diagrams are a quite new idea and thus not yet studied, broadly. Accordingly, there only few code generation concepts. Often, the basic idea is to focus on the collaboration messages. The message text is restricted to valid statements of the target programming language. Then, the message numbers are interpreted like control flow instructions and turned into the corresponding control flow constructs of the target language. The graphical parts of the collaboration diagram are just neglected. While this is a simple and working approach, it provides only a very low level of abstraction compared with plain program code. Fujaba overcomes this problem by assigning a default semantics to the graphical collaboration elements that raises the level of abstraction, significantly.

There exist several extensions of sequence diagrams, that turn sequence diagrams into full fledged programming notations. Plain sequence diagrams offer just sequences of messages or statements. Branches and loops may be introduced by just annotating the messages with appropriate guards and sequence numbers as known from collaboration diagrams. Some other approaches use additional rectangle boxes covering complete horizontal slices of a sequence diagram. The corresponding area is then annotated with an additional branch or iteration label flagging the conditional or repeated iteration of the corresponding sequence diagram slice. Basically, these ideas correspond to the simple code generation concept for collaboration diagrams that exploits only the collaboration messages and make little use of the graphical elements.

For sequence diagrams, TogetherJ provides a sophisticated implementation using the sequence number approach. TogetherJ provides sophisticated round-trip engineering support for such sequence diagrams and method bodies. In TogetherJ a single sequence diagram may represent the bodies of multiple methods. The implementation of these methods may be generated one-to-one from the sequence diagram. Conversely, TogetherJ may derive a sequence diagram from a method that represents the method body and in addition the bodies of methods called within such method bodies. Such an analysis may be restricted to a certain nesting depth or to certain classes or packages. Due to our experiences, such a common view of several methods facilitates the analysis of method effects and of method call chains, significantly.

Ideally, a system is fully specified at the UML level and then generated, fully automatically. There should be no need to modify the generated code. Even debugging should be done at the UML level. However, currently there are many reasons, why code generated from an UML specification might be modified, manually. First of all, most UML tools do not cover the full functionality of an application.

Commonly, large parts of the application consists of empty method bodies, to be filled by the developers. Even the Fujaba system does not cover all aspects of a system. For example, graphical user interface parts are not yet supported. Even if a UML tool would cover all system parts, if debugging is still done at the code level, most likely "minor" modifications will be done on the fly, too. If debugging is supported at UML level, some developers may refuse to use the UML tool for certain reasons. In a distributed team the tool may just not be available at all sites. In addition, current UML tools provide only limited support for version and configuration management at the UML level. If multiple developers work at the same system they may coordinate their work via a common source code repository controlled by a sophisticated configuration management system with optimistic locking concepts. Such an optimistic locking concept requires a merging mechanism for concurrent changes at one system part / file. Such merging functionality is not yet supported by UML tools. Although, we have started to develop such a mechanism for Fujaba, for quite a while, there will exist a need to merge changes on a source code basis. Altogether, this work at the source code level bears a high risk that the source code will deviate from the original UML design. In practice, after some weeks of development at the source code level, the changes to the class structure are usually, dramatic.

Thus, for current CASE tools the ability to analyse changes at the source code level and to adapt the design documents accordingly is one of the most important features. We call this feature *round trip engineering* support. Without round trip engineering support, the UML tool can only be used to create initial code frames. From then on, the developers work at the code level and the system looses its contact to the original design, soon. Most sophisticated UML tools try to provide such a round trip engineering support. However, due to our experiences and common reports from all CASE tool users we have interviewed, this support works reliably only in TogetherJ. TogetherJ actually does not use its own model repository. TogetherJ always analyses the current source code in order to retrieve the current UML specification. This way, code and design cannot deviate. On the other hand this advantage is bought with a somewhat lower level of abstraction of the design level employed in TogetherJ. For example TogetherJ does not support bidirectional associations.

Fujaba tries to provide similar round trip engineering support. However, to be honest, this mechanism works not yet fully stable. Fujaba tends to duplicate or omit some methods sometimes. To fix this is current work, but this will need some time just because of the size and complexity of this functionality.

Rational Rose supports round trip engineering, too. In Rational Rose this is based to a large extend on additional pseudo comments embedded in the generated source code. This additional pseudo comments provide additional design informations like association adornment details and they separate generated code from manually added parts. Software developers usually hate the pseudo comments of Rational Rose, because they massively obscure the code. It just becomes hard to see the code between all the comments. In addition, all users report, that the round trip engineering feature is not stable. The tool crashes or the model is corrupted. The common advise was: don’t try. However, Rational Rose will probably improve this feature, soon.

5.2 Summary

Table 5.4: Tool evaluation overview

	① Editing and per diagram consistency	② Hierarchical diagram organization	③ Declaration / usage consistency	④ Scenario / specification consistency	⑤ Code generation / reverse engineering
Rational Rose RT	+	--	+	--	0
Rhapsody	+	--	+	--	0
TogetherJ	+	--	+	--	0 Although code generation is behind the standard, the reverse and round-trip engineering features of TogetherJ are exceptional. TogetherJ is the one and only tool that is actually able to keep design and code consistent.
Fujaba	+	0 The hierarchical organization of use-case diagrams and story boards is current work	++	0 In Fujaba, the implementation of this feature is current work.	++ Due to our knowledge, Fujaba is the only tool that provides sophisticated code generation for all kinds of behavior diagrams. Especially , the code generation from collaboration diagrams is unique.
Others	The public domain tool Argo employs a very sophisticated, interactive per diagram consistency check concept. Errors and incomplete elements are collected and analyses at a logical level. The problems are prioritized and turned into a todo list.			The SCED environment provides the most sophisticated support for synthesizing statecharts from sequence diagras. Its successor, the MAS system is even interactive. Multiple new tools are under development.	

Table 5.4 shows a coarse grained overview of our evaluation. We would like to conclude this chapter with a clear recommendation when to use which tool. Unfortunately, none of the tools provides the

desired functionality. None of them is fully usable in practice. Of course, we consider Fujaba as the tool with the most sophisticated concepts. However, Fujaba is just a research prototype and needs some further development in order to gain industrial strength. Rational Rose is the market leader. Actually, the rational tool suite provides a broad range of important functionality. It is probably a good choice for earlier project phases. However, Rational Rose has some weaknesses in its code generation concept and the round trip engineering support is fare beyond the one of TogetherJ. Thus, after the initial design phase, Rational Rose is no longer recommendable. This gap might be filled with TogetherJ which has very sophisticated round-trip engineering support. Although it provides only a limited level of abstraction, it is still very helpful in the design and coding phase and during further development. If sophisticated code generation concepts for behavior diagrams are required, Rhapsody might be the tool of choice.

Again, CASE tools are moving targets. Some of our conclusions may be out-dated when this work is published. We may have missed some features.

6 Conclusions

This work overcomes major deficiencies of existing methods and specification languages for object oriented applications. It provides (1) a rigorous technical development method facilitating the analysis and design of complex graph-like object structures, cf. Figure 6.1.

Figure 6.1: Story Driven Modeling

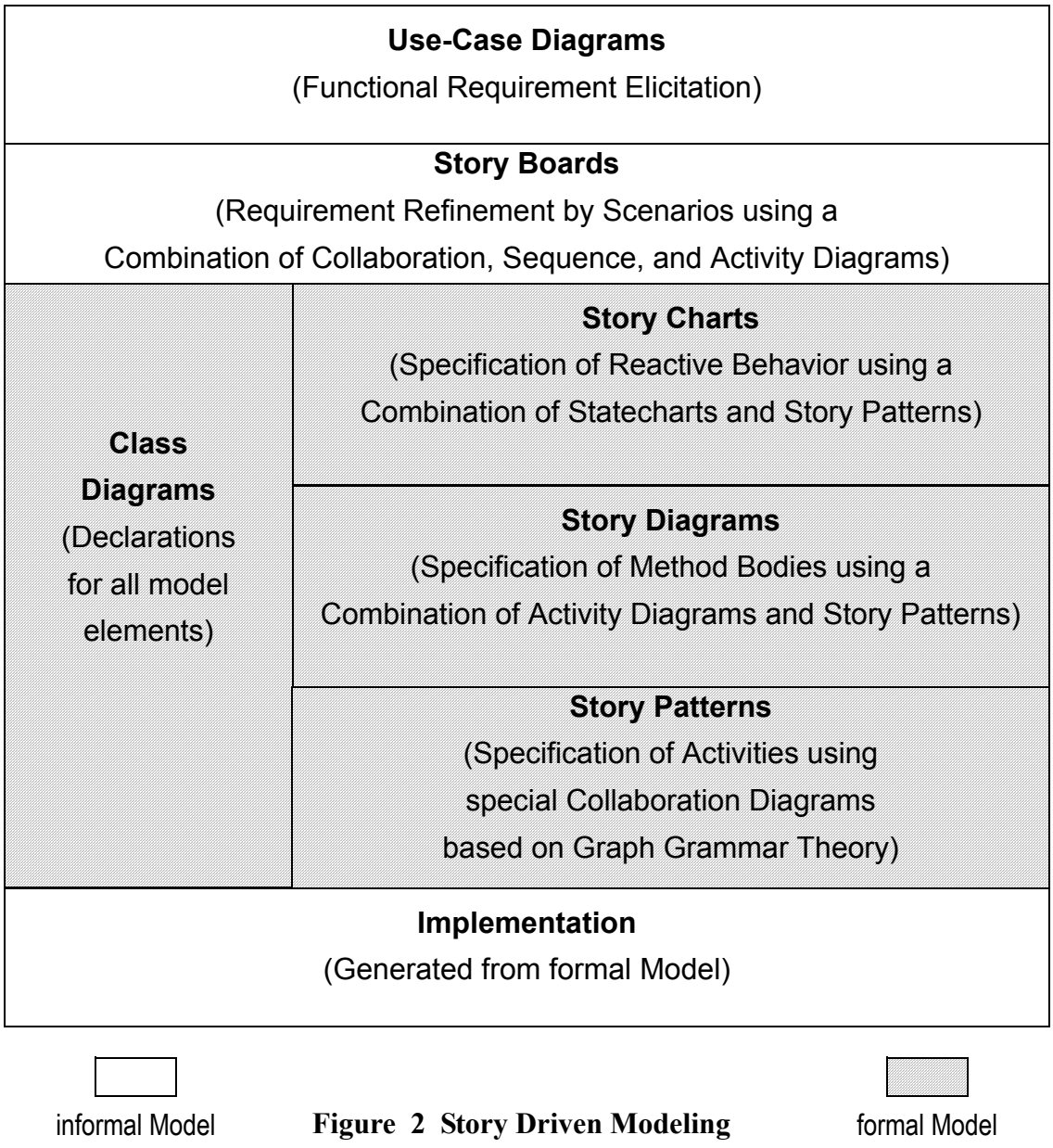


Figure 2 Story Driven Modeling

This rigorous development method is (2) based on a formal semantics description for UML structure and behavior diagrams. All diagrams of a specification are organized in a strict way. For each diagram it is defined which aspect of the overall system is modeled and how all diagrams fit together. This allows to check for the completeness and consistency of the overall specifications. Based on the formal semantics definition for UML structure and behavior diagrams, (3) this work provides fully automatic code generation concepts that cover the whole specification and application logic. These concepts are implemented as part of the Fujaba environment. Fujaba stands for "From UML to Java And Back Again". The generated code allows the validation of the specified behavior. This is supported by a dedicated simulation environment called "Dynamic Object Browser", Dobs. The Fujaba environment is public domain and open source. It may be downloaded from www.fujaba.de. Up to

May 2001 we have recorded about 8000 downloads. Currently, the implementation comprises about 380 000 LOC. It has been created and is maintained and further extended by a team of 10 to 20 developers starting January 1997. Fujaba has been created by applying the concepts proposed in this work in a logical boot strap. Since 2000, extensions of the Fujaba environment are mainly done using the Fujaba environment on its own implementation. Even the generators under development do no longer create source code directly but they translate higher level specifications into story diagrams and employ the existing code generators to derive an implementation. In addition, the Fujaba environment provides reverse and round-trip engineering support that allows to incorporate manual changes to the source code into the corresponding design documents.

In addition, SDM has been taught with great success in multiple courses at University of Paderborn and Braunschweig. It has been adapted for courses at various high-schools in the area of Paderborn. Tutorials about SDM have been and will be held at ICSE'99 in Los Angeles, ESEC'99 in Toulouse, FSE2000 in San Diego, ICSE2001 Toronto, and ESEC 2001 in Vienna. Several times, we have held industrial courses on SDM. SDM has been applied to several industrial and research projects. For example the development of a web based information system for the public transportation system of Paderborn and the course program planning system of the computer science department of Paderborn University, cf. [RT97]. The results of the Fujaba project have been published on many software engineering and graph grammar conferences and workshops [JZ97, JZ98, FNTZ98, NNSZ99, SZ99, SZ99b, KNNZ00, NNZ00, NZ00, NZ00b, SZ00, NWZ01, Zü01, Zü01b].

Although we have achieved some progress in the direction of rigorous software development, many current and future work remains. First of all, the support for early phases and for the transition from analysis to design needs improved tool support. So far, we mainly provide consistency checks. Current work incorporates the concepts of [MS00, MS01] into SDM and Fujaba. The idea is to analyse scenarios with respect to traces of signals and actions received and executed by certain kinds of objects. These traces are interpreted as possible words of a formal language. Then techniques known from the field of compiler construction are employed to derive a grammar that accepts all provided example words. This grammar is then turned into a finite state automata. This process results in a statechart or activity diagram for the considered kind of object. Employing reverse engineering techniques developed in our group [Klein99], such statecharts and activity diagrams may be turned into story charts and story diagrams.

Another branch of current work for improved support of early phases tries to provide code generation even for use-case and scenario diagrams. The idea is that each use-case although defines a test case. The corresponding scenarios are interpreted as possible executions of the defined functionality. Each use-case and scenario is connected to a method that should realize the described feature. The scenarios define starting situations and intermediate steps that should be executed if the implementation is called. The intermediate steps are turned into trace instructions embedded into the final source code. The use-case is turned into a test method that creates the start situation, calls the implementing method, collects the intermediate trace information, and compares this trace information with the example scenarios. This allows to detect inconsistencies between scenarios and specification or implementation.

Our formalization of the UML will serve as basis for future work on verification techniques for UML specifications. [HW95] proposes a technique that allows to state invariants and proof obligations as graphs that e.g. must always exist or must never exist. Then induction techniques are proposed that allow to proof that such properties are guaranteed by all story patterns applied to the given object structure. These techniques will allow to verify important system properties at the UML level of abstration. To facilitate this kind of verification, we will develop interactive tool support for induction steps.

Another branch of current research, addresses design patterns as proposed by [GHJV95]. We consider design patterns as another alternative to describe the semantics of certain classes, attributes, methods, and relations. Since a design pattern defines certain behavior aspects, it becomes possible to exploit these semantics information for code generation. Similarly, we try to provide reverse and round-trip engineering support for design patterns [NWZ01].

SDM facilitates the development of applications that employ complex graph-like object structures. Frequently such applications require persistency for their logical data. For technical reasons, this persistency is often realized using a relational database system. This relational database system may even exist already, and an object oriented application on top of the existing system is desired. To support this kind of application, Fujaba tries to pick up and incorporate the results of our Varlet project, [JSZ97b, JZ98b]. The Varlet project has developed automatic analysis support to retrieve design information from relation database schemes, access code, and relational data and to provide an appropriate object oriented schema for the contained data. Based on this work, we want to generate an object-oriented access layer for relational databases that allows to realize new functionalities using SDM and especially, story charts and story diagrams and the code generation facilities of Fujaba. Similarly, we have started to provide analysis and generation support for XML. This work shall facilitate to turn certain object structures into XML descriptions and vice-versa. Using style sheets such XML descriptions are easily turned into html files and forms. These html files and forms may be represented by usual WWW browsers. Filled forms may be transferred back into XML data and then turned into the corresponding object structures, automatically. Thus, the provided form content becomes available for further treatment with SDM and story diagrams. Together the relational database and the XML support would facilitate the construction of E-commerce applications that require some complex logical functionality at server site.

SDM already provides support for the realization of reactive systems. However, so far, a common object structure is assumed as basis for information exchange between multiple threads. In practice, different threads may run on different nodes with separated memory spaces. Therefore, we try to extend SDM with SDL block diagram elements [ITU96] and / or with ROOM capsules and communication channels [SGW94]. In addition, SDM shall address real-time problems. SDM employs a variant of statecharts where events are sequentialized and queued, where the consumption of events and the execution of actions may need time, and where the creation, transportation, and delivery of events may need time. Finally, we have to develop a sound theoretical basis allowing to reason about hard real time aspects like guaranteed reaction times and reaction at exactly defined clock times.

For the Fujaba environment the team support must be improved. As most existing CASE tools, so far Fujaba supports single users only. For teams of users, configuration management support and task coordination mechanisms are needed. Larger specifications will be developed by multiple team members in parallel. Since all these developers work on a single specification, we need the possibility to split a specification in separated parts that may be edited by different users, independently. In addition, we should allow that different developers edit the same specification (part), concurrently. We already have developed a merge algorithm for complex object structures that allows to combine the changes of different users to a single specification (part), afterwards, cf. [Rock00]. This work needs to be extended. Further extensions of Fujaba with task coordination mechanisms will support project management and project planning activities. Based on Use-Case diagrams and initial analysis and design documents, Fujaba may support the derivation of effort estimations and plans for development tasks. During project execution, Fujaba may serve as a process engine providing all users with agendas and coordinating information exchange and controlling project progres.

Finally, we plan to apply SDM to other industrial and research projects. This will continuously provide feedback for the improvement of method and tools.

Appendix A: Formalizing our usage of the UML

A.1 Introduction

As the title suggests, we are not going to formalize standard UML, but "only" the (large) parts of the UML as they are employed in our Story Driven Modeling approach, i.e., we formalize a cut-out of class diagrams, story patterns, story diagrams, and story charts or collaboration diagrams, activity diagrams, and statecharts, respectively. Within the IPSEN project, graph grammars have proven to be well suited for the formalization of object-oriented modeling languages and for the construction of CASE tools, cf. [ELNSS92, Nagl96, KSZ96, SZ96, SWZ96, SWZ96b, Zü96]. Graph grammars provide an operational formalization mechanism, that enables verification of system properties as well as automatic code generation. The formalization approach we use is inspired by the formalizations of the Progres language [Sch91, Zü96] and by other graph grammar approaches, cf. [Roz97]. Generally, graph grammars are a quite complex formalism. However, in a course on graph grammar engineering, cf. [Zü00], we have developed a simplified graph grammar formalism for didactic purposes based on a simple set-theoretic approach.

The object oriented data model has several features which are not part of most graph grammar approaches. In addition to plain graph models, the object oriented data model offers inheritance, aggregation, qualified associations, sorted associations, and ordered associations. Thus, we had to extend existing theoretic approaches by new means covering these object-oriented data model features. Another extension of plain graph models is that in the object oriented model methods are attached to objects and method calls may be polymorphic. This needs appropriate handling, too. However, it causes no major extensions of our formal semantics.

The formalization first defines the semantics of class diagrams in terms of the described set of all possible object structures. Second, the semantics of a story pattern or collaboration diagram is described using set operators on object structures. Third, the execution of the control flow in story or activity diagrams is defined using a small control flow interpreter. Similarly, the fourth step defines the semantics of story charts or statecharts.

A.2 The semantics of class diagrams

From the graph grammar theory point of view, class diagrams define a so-called graph schema. A graph schema provides sets of node labels and edge labels and attribute names that may be used to describe the elements of certain graphs. Thus, the first step is to define the term object oriented graph:

Definition A.1: Object oriented graphs

G := (SI, Ext) where

SI := (NL, EL, A, IsAs, Assocs, Attrs) where (Schema Info)

NL	finite set of node labels	
EL	finite set of edge labels	
A	finite set of attribute names	
IsAs	$\subseteq$ Rel (desc $\in$ NL, anch $\in$ NL)	where Rel := "relation of "
Assocs	$\subseteq$ Func ((el $\in$ EL) $\rightarrow$ (srcNl $\in$ NL, srcCard $\in$ MultiInfo := {one, many} assocType $\in$ P (AssocTypes := {qualified, aggregation, ordered}) tgtNl $\in$ NL, tgtCard $\in$ MultiInfo))	

Attrs $\subseteq$ Func ((A) $\rightarrow$ NL $\times$ BaseTypes)
where BaseTypes := {Boolean, Integer, String, Float, P(Integer), ...}

Ext := (N, E, nl, av) (Extension)

N finite set of nodes (node ids)

E Rel (src $\in$ N, el $\in$ EL, i $\in \varepsilon \cup$ R, q $\in \varepsilon \cup$ AttrValues, tgt $\in$ N)

nl Func ((N) $\rightarrow$ NL)

av Func ((N, A) $\rightarrow$ AttrValues := {true, false} $\cup$ *R* $\cup$ char* $\cup$ *R** $\cup$ ...)

An object oriented graph G consists of two parts a so-called schema information SI and an extension Ext. The *schema information* provides sets of node labels (NL), edge labels (EL), and attribute names (A) used to classify the elements of the extension. So far, these are elements of classical graph schemas. The relation IsAs models the first object oriented feature, namely inheritance. Since object oriented associations are slightly more complex than plain graph edges, we employ the additional relation Assocs to model these features.

For each edge label el $\in$ EL relation Assocs provides a tuple defining the label(s) allowed for source nodes (srcNl) and for target nodes (tgtNl). Associations may declare cardinality constraints within their roles. These cardinality constraints are reflected via srcCard and tgtCard. Note, we distinguish to-one and to-many associations, only. In UML association cardinalities may provide lower bounds like 0..1 and 0..n or several intervals like 1..4, 8..12 or exact numbers. To keep things simple, we decided to support only to-one and to-many associations directly in our data model and to cover the other possibilities using additional constraints extending the core data model.

In addition to cardinality constraints, associations may be classified as aggregation or composition relationship, as qualified associations or ordered associations or sorted associations. Qualified and ordered and sorted associations will be discussed together with the kind of edges that are part of an extension. So far, there exists no common understanding for the semantics of aggregation and composition relationships. [BRJ99] define that composition relationships describe „co-incident life time“. That means, composite objects are created in one shot and the components stick together without any changes and the composition is deleted as a whole. This idea stems from nested record or struct definitions known from Pascal or C or C++ that behave this way.

Due to our experience, in object oriented structures it is often more convenient to construct the composition in several steps within a certain subsystem. Similarly, one may want to exchange certain parts of the composition after the construction or one may want to save some parts from deletion for later reuse. Thus our core data model does not provide strict co-incident life time but supports the weaker idea of existence dependency, only. Composite objects may be constructed in several steps and the composition may be changed. However, deletion of a composite object is automatically forwarded to its „part“ objects. This semantics of existence dependency corresponds to the usual meaning of aggregation, too. Thus, our core data model supports aggregation directly. If required, strict composition may be expressed using additional constraints.

For each attribute name, function Attrs describes the declaring class and the corresponding attribute type. Only objects that belong to the declaring class or one of its subclasses (defined by the IsAs* relation) posses this attribute. Allowed attribute types are boolean, integer, string, and float. In addition one may use sets or lists or arrays of these types and such containers may be nested, again. However, an attribute must not hold a reference to another object nor a struct of different attribute values. For references one has to use associations and for structs one shall declare another class.

In our data model, the *extension of a graph* consists of 4 elements, a set of nodes or objects N, a set of edges E, a node labeling function nl, and an attribute value function av. N defines the set objects of a given graph G. Note, objects have their own identity and two objects with equivalent attribute values and equivalent links attached to them are still distinguishable.

Edges are tuples consisting of a source node src, an edge label el, possibly an index number i, possibly a qualifying attribute value q, and a target node tgt. Note, in opposite to objects, our edges do not have their own identity. The set of edges can contain only one tuple with the same values for its constituents. In case of a plain association a1 this means that there may exist at most one edge with label a1 that connects node n1 with node n2 in this direction. Adding a second tuple with the same constituents would not change the current graph. However, in case of a sorted or qualified association there may exist multiple edges with the same label connecting nodes n1 and n2 as long as they carry different values for their index i or their qualifier q.

Disclaimer: Note, our formal model for object oriented graphs does *not* (yet) restrict edges (or attribute values) to be type conform with the schema definition. Each edge has a label denoting a certain association definition and certain source and target nodes. The association definition contains certain requirements for the labels of source and target nodes of the corresponding edges. For example a carries edge must connect a Shuttle node with a Good node. More generally, the label nl1 of a source node n1 of some edge e1 with edge label el1 should be contained in the IsAs* set of node labels that conform to the source node label required by associtio el1. In addition, existing cardinality constraints must not be violated. Similar constraints must hold for the attribute value function. An object oriented graph where the extension conforms to all requirements stated in the corresponding graph schema (as one usually expects) may be called a *well-formed object oriented graph*. Unfortunately, it is not trivial to guarantee the well-formedness property of object oriented graphs for all kinds of complex graph modifications that will be defined in the remainder of this chapter. We will introduce the semantics of story diagrams as a relation between object oriented graphs. However, to show that a story diagram guarantees that its application to a well-formed object oriented graph produces again a well-formed object oriented graph needs still to be done. This requires a sound and complete type concept for our whole language, i.e. for most parts of the UML. This is a major effort that is beyond the limitted scope of this work. Thus, for this work we drop the formal requirement, that extensions have to conform to their graph schema. We try to provide this notion in future work.

Note, our kinds of edges are an extension of the Progres graph model that does not provide edge indices or qualifiers. However, other graph grammar approaches employ edge objects, where edges have their own identity and may carry arbitrary attributes. This kind of edge objects would be especially suited to model attributed associations. The hypergraph model even employs n-ary edges with an arbitrary number of targets. n-ary edges allow to model n-ary associations, easily. Our choice for representing edges as tuples was driven by our implementation strategy for associations. Fujaba implements associations as pairs of references within the corresponding classes. For to-one associations we use plain references. For to-many associations we use sets, lists, or hashtables that easily implement plain, sorted, ordered, or qualified associations, cf. chapter 3. This kind of implementation is not able to represent multiple (plain) edges of the same type and direction between the same pair of objects. A more powerful edge model would require a more complicated implementation, e.g. employing intermediate edge objects.

The node labeling function nl allows to look up the node label or class name for a given node or object n. This provides us with runtime type information that we will use e.g. for method look-ups and for runtime type checks. The attribute value function av stores and retrieves attribute values for a given node and attribute name. Note, integer and float attributes are not restricted to a certain maximal number or precision but integer attributes may store any positive or negative natural number and floats

may store any real number of any precision. Similarly, strings may have an arbitrary length. This facilitates our theory, since we do not need to deal with overflow problems or e.g. with dirty zeros. However, our standard implementation of attributes will use the types boolean, int, float, and String of the corresponding programming language, Java in our case. Thus, such an implementation may face overflow and precision problems and therefore be incorrect.

Provided with this graph model we consider UML class diagrams and UML object diagrams just as a graphical rendering of the schema information and of the extension of object oriented graphs, respectively:

Definition A.2: Rendering object oriented graphs as class diagrams and object diagrams

Let G := (Sl, Ext) be an object oriented graph with

Sl := (NL, EL, A, IsAs, Assocs, Attrs) and

Ext := (N, E, nl, av)

Rendering schema information

R1: A node label C1 ∈ NL is rendered as a UML class with name "C1".

R2: An attribute a ∈ A with Attrs (a) = (C1, T) is rendered as an attribute declaration a : T within the attribute compartment of class C1.

R3: Each tuple (SubclassB, SuperclassA) ∈ IsAs is rendered as an arc with hollow arrow head from class SubclassB to class SuperclassA.

R4: Each tuple from the Assocs relation is rendered as an association between the corresponding classes showing the corresponding labels and cardinality constraints.

Rendering extensions

R5: Each node n ∈ N with nl(n) = C1 is rendered as an object n : C1.

R6: Each tuple (n, a → valx) ∈ av is rendered as an attribute equation a==valx shown in the attribute compartment of object n.

R7: Each tuple (sn, el, i, q, tn) ∈ E is rendered as a link between objects sn and tn with label el possibly followed by qualifier q in square brackets and/or a dot and index i, e.g. el.1, el[q], el[q].1.

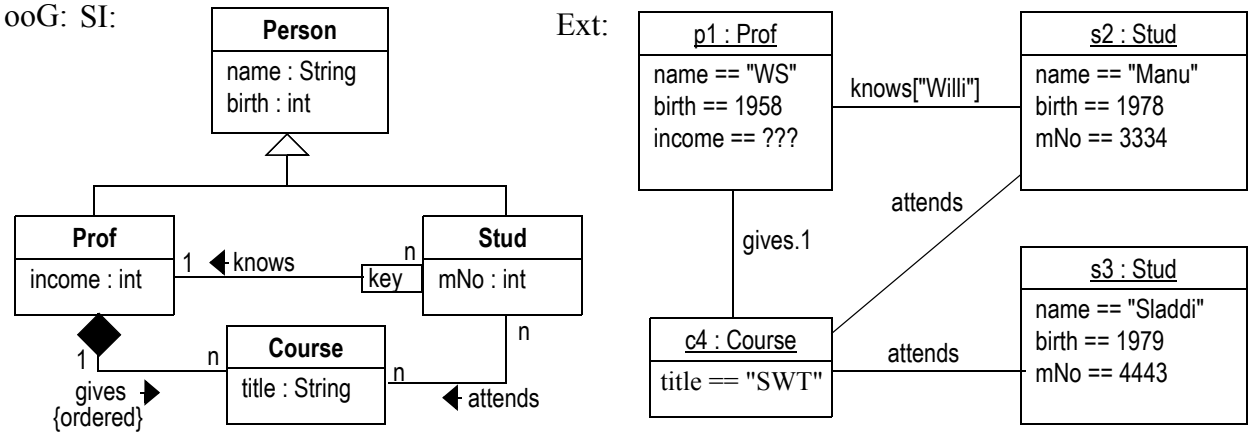
Note, we consider class diagrams as a graphical rendering of our formal model. Our formal model defines the set of all diagrams that are valid according to our formal model. This also defines the set of all valid graphical renderings of class diagrams and object diagrams (besides layout properties). Any other graphical picture is just a graphical picture, it is not a class diagram nor an object diagram.

Example A.3 shows an example class diagram and an example object diagram in our formal model and rendered in UML notation.

Example A.3: Class Diagram and Object Graph

ooG = (Sl, Ext)
  Sl = (NL, EL, A, IsAs, Assocs, Attrs)
    NL = {Person, Prof, Stud, Course}
    EL = {knows, gives, attends}
    A = {Person.name, Person.birth, Prof.income, Course.title, Stud.mNo}
    IsAs = {(Prof, Person), (Stud, Person)}
    Assocs = { (attends → Stud,many, {}, Course, many),
               (knows → Stud, many, {qualified}, Prof, one),
               (gives →Prof, one, {aggregation, ordered}, Course, many)) }
    Attrs = {(Person.name → Person, String), (Person.birth → Person, int),

(Prof.income → Prof, int), (Stud.mNo → Stud, int), (Course.title → Course, String)}
Ext = (N, E, nl, av)
N = {p1, s2, s3, c4};
E = {(s2, attends, ε, ε, c4), (s3, attends, ε, ε, c4), (p1, gives, 1, ε, c4), (s2, knows, ε, "Willi", p1)};
nl = {(p1→Prof), (s2→Stud), (s3→Stud), (c4→Course)};
av= {(p1,Prof.name→"WS"), (p1,Prof.birth→1958), ...}



We admit, that the rendering of our formal model as the usual class and object diagrams has been discussed rather short. However, we hope that this introduction to our data model suffices to yield a common understanding of our data model and how it corresponds to UML class and object diagrams. Note, we did not yet consider methods. This will be done in chapter A.5.

A.3 Formalizing story patterns

As discussed in chapter 2.3, activity 5, story patterns are an adaption of usual UML collaboration diagrams. A simple way to assign a semantics to UML collaboration diagrams is to exploit the collaboration messages only. The message text may be restricted to some (formal) programming language. This is actually the current formalization approach of the object management group OMG which has issued a call for proposal for an action semantics. Such an action semantics will provide a formal programming language for actions embedded in statecharts, activity diagrams, and for collaboration messages embedded in collaboration or sequence diagrams. While such an action semantics provides a formalization of basic actions, the numbering schema of collaboration messages is easily interpreted as a control flow specification. However, as discussed in chapter 2.3 such an approach provides a very low level of abstraction, only, compared to usual programming languages. In addition, such an approach ignores the graphical elements of collaboration diagrams, totally. In contrast, our approach assigns a standard semantics to the graphical elements of collaboration diagrams. This standard semantics for graphical elements simplifies the specification with collaboration diagrams, dramatically. As an example one may revisit the conventional collaboration diagram shown in Figure 9 on page 25 and the semantically equivalent diagram shown in activity 1 at the top of Figure 10 on page 29. We call collaboration diagrams with such a standard semantics for their graphical elements *story patterns*.

We will formalize story patterns using the theory of graph rewrite rules. Generally, a graph rewrite rule consists of a pair of graphs, the left-hand side and the right-hand side, cf. e.g. the Progres approach [Zü96]. The graph of the left-hand side describes a „before“ situation, i.e. a cut-out of the current object structure that is going to be modified. The right-hand side graph describes the „after“ Situation, i.e. how the cut-out of the current object structure should be changed by the rule. Due to our experiences, frequently these two graphs are quite similar. Graph rewrite rules may look-up quite complex graphs consisting of 10 to 20 nodes, but usually they perform only little changes most fre-

quently just adding or deleting one node and some edges and attributes. For such rules the separated notation of left- and right-hand side is quite uncomfortable. One has to draw a quite complex graph twice, in order to show only little modifications. In addition, the reader of such a rule has to compare two complex graphs in order to determine the differences, i.e. the performed changes. Therefore, story patterns combine the left-hand and right-hand side of the depicted graph rewrite rule into a single picture. In a story pattern the performed changes are explicitly denoted using «destroy» and «create» tags. This facilitates to read (and write) complex rewrite rules that perform only some modifications. In case the graph rewrite rule performs dramatic changes of the left-hand side, this notation is more complicated than usual graph rewrite rules, however, in story diagrams one may easily split such complex rules in two subsequent story patterns one modeling the left-hand side (including «destroy» tags) and a second modeling the right-hand side, the remaining actions.

Although a story pattern shows a single graph, only, to apply graph grammar theory for the semantics definition of story patterns we split story patterns into a pair of graphs, i.e. into a left-hand and a right-hand side. First we need some auxiliary definitions:

Definition A.4: Common shorthands

- GraphClass (SI) denotes the set of all graphs with schema information SI
- let SI be a given schema information and GC := GraphClass(SI)
- as a shorthand a graph G ∈ GC may be given by its extension Ext, only
- Let G = (Ext=(N, E, nl, av)) ∈ GC , then we denote
  - N_G := N the node set of G
  - E_G := E the edge set of G
  - nl_G := nl the node labeling function of G
  - av_G := av the attribute value function of G

The semantics definition for story patterns starts with "basic" story patterns that employ only some story pattern language elements. Once the semantics of these basic story patterns is defined we will introduce more complex story pattern features step-by-step.

Definition A.5: Basic story patterns

Let GC be a GraphClass over SI.

A basic story pattern is a pair of graphs (LG, RG) where LG, RG ∈ GC and LG and RG are consistently marked,

i.e. $nl_{LG}|_{N_{LG} \cap N_{RG}} = nl_{RG}|_{N_{LG} \cap N_{RG}}$

Let grr := (LG, RG). Then we denote:

- $DelN_{grr} := N_{LG} - N_{RG}$ the set of nodes deleted by grr
- $DelE_{grr} := E_{LG} - E_{RG}$ the set of edges deleted by grr
- $CoreN_{grr} := N_{LG} \cap N_{RG}$ the core nodes of grr
- $AddN_{grr} := N_{RG} - N_{LG}$ the set of nodes created by grr
- $AddE_{grr} := E_{RG} - E_{LG}$ the set of edges created by grr

A basic story pattern or graph rewrite rule consists of a pair of graphs (LG, RG) belonging to the same graph schema (or class diagram). Graph LG is the so-called left-hand side of the story pattern describ-

ing the before situation. Graph RG is the so-called right-hand side of the story pattern describing the after situation. Note, LG and RG may have a common subgraph, i.e. common nodes and edges. This so-called core-graph describes unchanged elements serving as context of the graph rewrite step. The core-graph allows to connect created elements to existing elements or in graph grammar terms, the core-graph allows to embed the right-hand side elements.

Informally, a story pattern is applied by replacing (a match of) the left-hand side by (a copy of) the right hand side. All elements that are part of the left-hand side LG but not part of the right-hand side RG are deleted. Elements that occur on the right-hand side RG, only, are to be created. Thus $DelN_{grr} := N_{LG} - N_{RG}$ denotes the nodes of the story pattern that are to be deleted during an execution. Similarly, $DelE_{grr} := E_{LG} - E_{RG}$ denotes the edges that are explicitly deleted. Core nodes contained in $CoreN_{grr} := N_{LG} \cap N_{RG}$ remain unchanged. $AddN_{grr} := N_{RG} - N_{LG}$ and $AddE_{grr} := E_{RG} - E_{LG}$ are created. Note, a graph may contain only edges that connects nodes within its node set N, cf. Definition A.1. Thus, deleting a node n implies that all edges using n either as target or as source must be deleted, too, in order to yield a valid graph, again.

Example A.6 shows a simple example story pattern in formal notation according to Definition A.5 and its rendering as an UML collaboration diagram:

Example A.6: A basic story pattern, formally and graphically

$N_{LG} = \{c1, c2, s3, s4\}$

$E_{LG} = \{(s3, attends, \varepsilon, \varepsilon, c1), (s3, attends, \varepsilon, \varepsilon, c2), (s4, attends, \varepsilon, \varepsilon, c2)\}$

$nl_{LG} = \{c1 \rightarrow Course, c2 \rightarrow Course, s3 \rightarrow Student, s4 \rightarrow Student\}$

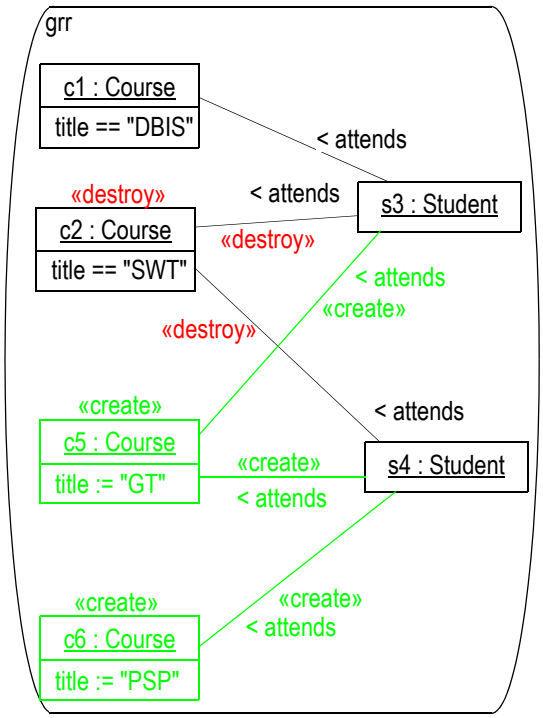
$av_{LG} = \{(c1, Title) \rightarrow "DBIS", (c2, Title) \rightarrow "SWT"\}$

$N_{RG} = \{c1, s3, s4, c5, c6\}$

$E_{RG} = \{(s3, attends, \varepsilon, \varepsilon, c1), (s3, attends, \varepsilon, \varepsilon, c5), (s4, attends, \varepsilon, \varepsilon, c5), (s4, attends, \varepsilon, \varepsilon, c6)\}$

$nl_{RG} = \{c1 \rightarrow Course, s3 \rightarrow Student, s4 \rightarrow Student, c5 \rightarrow Course, c6 \rightarrow Course\}$

$av_{RG} = \{(c1, Title) \rightarrow "DBIS", (c5, Title) \rightarrow "GT", (c6, Title) \rightarrow "PSP"\}$



- $DelN_{grr} = \{c2\}$
- $DelE_{grr} = \{(s3, attends, \varepsilon, \varepsilon, c2), (s4, attends, \varepsilon, \varepsilon, c2)\}$
- $CoreN_{grr} = \{c1, s3, s4\}$
- $AddN_{grr} = \{c5, c6\}$
- $AddE_{grr} = \{(s3, attends, \varepsilon, \varepsilon, c5), (s4, attends, \varepsilon, \varepsilon, c5), (s4, attends, \varepsilon, \varepsilon, c6)\}$

The left-hand side of our example story pattern grr consists of the nodes c1, c2, s3, and s4 and their connecting edges and their attribute conditions. Node c2 is marked by a «destroy» tag, indicating its deletion. Formally, this is modeled by excluding c2 from the core graph. Thus, the core graph contains only nodes c1, s3, and s4 and the corresponding edges and attribute conditions. This elements form

the context of the described graph rewrite step. The right-hand side of grr consists of the core graph plus the created elements (marked by a «create» tag). These are nodes c1, s3, s4, c5, and c6 and the corresponding edges and attribute conditions and assignments.

The execution semantics of a basic story patterns is now easily defined by the following steps:

1. Choosing a handle
2. Deletions
3. Creations

Choosing a handle that fits to the left-hand side of the story pattern means to find a subgraph of the current host graph that is isomorphic to the left-hand side of the story pattern. Thus, in order to formalize this step we need to introduce the notions of subgraph and graph isomorphism:

Definition A.7: Subgraph

Let SG, G ∈ GC

SG is a subgraph of G (or as infix operator SG <= G)

:<=>

- $N_{SG} \subseteq N_G$
- $unQual(E_{SG}) \subseteq (unQual(E_G) \cap (N_{SG} \times EL \times N_{SG}))$
  and there exists an *injective* function origE: E_{SG} -> E_G such that
  $\forall\ e := (sn, el, i, q, tn) \in E_{SG}$ holds $\exists\ e' := (sn, el, i', q', tn) \in E_G$
  such that
  e' = origE (e) and
  i != e => i == i' and
  q != e => q == q'
  where
  unQual (E) := { (sn, el, tn) | (sn, el, i, q, tn) ∈ E}

- $nl_{TG} = nl_G|_N$ // compatible node labeling
- $\forall n \in N_{SG}, a \in A$ where $av_{SG}(n, a) = X$ implies $av_G(n, a) = X$

A graph merely consists of sets of nodes and edges and attributes. Thus, a subgraph merely consists of subsets of these nodes, edges, and attributes. First, the node set of the subgraph SG must be included in the node set of G. Similarly, one could just define that the edge set of SG must be a subset of the edges of G: $E_{SG} \subseteq E_G$. Note, SG is a graph. This already imposes the condition, that the edges of SG connect nodes of SG, only. Thus, if SG omits some node n of G, edges of G attached to n must not be part of SG, by definition.[3]

For object oriented graphs, we employ a more complex notion of subgraphs with respect to edges than just $E_{SG} \subseteq E_G$. The left-hand side of a story pattern may look-up links belonging to qualified or ordered associations without providing values for the qualifier or index. Such a link in a story pattern shall fit to any link in the host graph with the same label, independent of the actual qualifier or index. Story patterns are applied to subgraphs of a host graph, that are isomorphic to the left-hand side of the story pattern. Isomorphic merely means "equal structure but different naming". Thus, if the left-hand side of a story pattern does not provide a qualifier or index for a link, we must be able to choose sub-

3. The same holds for attribute values.

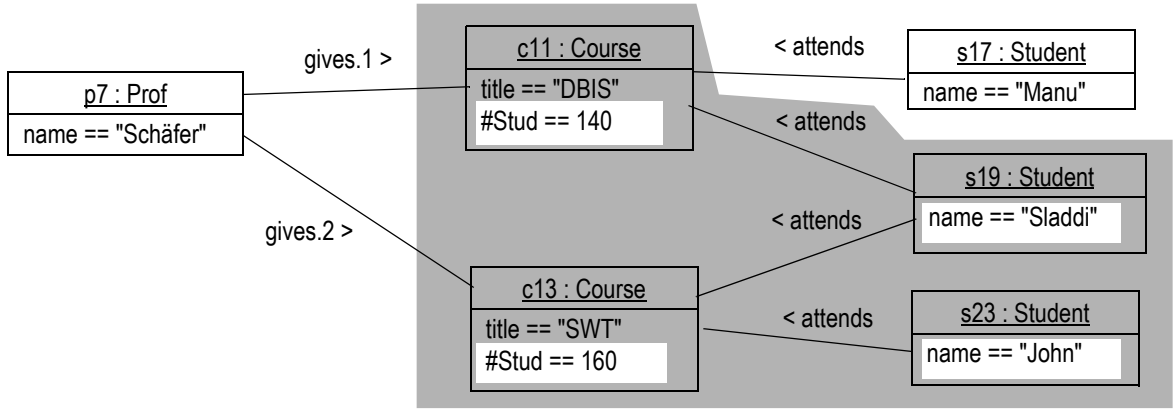
graphs of the current host graph that do not contain the qualifier or index of an edge. Therefore, our subgraph definition employs the operator unQual (E) which turns the 5-tuple edges of object oriented graphs into 3-tuples containing only source node, edge label, and target node of the original edges. Thus, unQual (E_{SG}) $\subseteq$ unQual (E_G) just demands that, neglecting qualifiers and indices, the subgraph may contain only edges that already exist in the original graph. In addition, our definition explicitly demands that subgraph edges connect nodes of the subgraph only: the set ($N_{SG} \times EL \times N_{SG}$) denotes all possible (un-qualified) edges between nodes of the subgraph SG.

Edges within the subgraph SG may omit qualifiers or indices of their originals in graph G using the value ε for the corresponding tuple column. However, if an edge in subgraph SG contains an qualifier or index it must be the same as the one in the original graph G. Thus we require that it must be possible to provide an *injective* function origE that uniquely identifies an original edge e' for each edge e in the subgraph SG. And, if e contains an index or qualifier not equal to ε then it must be equal to the original value.

The node labeling function of the subgraph just has to be equal to the node labeling function of the host graph, restricted to the nodes of subgraph SG. Similarly, the attribute value function is defined only for nodes contained in the subgraph. Note, in addition, the attribute value function of the subgraph may omit arbitrary attribute values for arbitrary nodes by just being undefined for these combinations of nodes and attribute names.

To summarize, a subgraph SG of a given host graph G is constructed by choosing a subset of nodes, edges, and attribute values. In addition, for chosen edges one may drop certain edge qualifiers and indices.

Example A.8: Choosing a subgraph



N_G = { p7, c11, c13, s17, s19, s23 }

E_G = { (p7, gives, 1, ε, c11), (p7, gives, 2, ε, c13), (s17, attends, ε, ε, c11),
(s19, attends, ε, ε, c11), (s19, attends, ε, ε, c13), (s23, attends, ε, ε, c13) }

l_G = { p7 -> Prof, c11 -> Course, c13 -> Course, s17 -> Student, s19 -> Student,
s23 -> Student }

av_G = { (p7, Person.name) -> "Schäfer", (c11, Course.title) -> "DBIS",
(c11, Course.#Stud) -> 140, (c13, Course.title) -> "SWT", (c13, Course.#Stud) -> 160,
(s17, Person.name) -> "Manu", (s19, Person.name) -> "Sladdi",
(s23, Person.name) -> "John" }

N_{SG} = { c11, c13, s19, s23 }

E_{SG} = { (s19, attends, ε, ε, c11), (s19, attends, ε, ε, c13), (s23, attends, ε, ε, c13) }

l_{SG} = { c11 -> Course, c13 -> Course, s19 -> Student, s23 -> Student }

av_{SG} = { (c11, Course.title) -> "DBIS", (c13, Course.title) -> "SWT" }

In example A.8 we have chosen a subgraph SG consisting of the nodes c11, c13, s19, and s23 and the connecting attends edges. Subgraph SG contains the title attributes of nodes c11 and c13, but the other attributes are omitted. Another subgraph SG2 containing nodes p7 and c11 could contain edge (p7, gives, 1, ε, c11) or an edge (p7, gives, ε, ε, c11) omitting the index or no edge at all.

Note, for a given finite graph G the number of possible subgraphs is finite, too. Graphs consists of sets of related entities sets do have a finite number of subsets. Thus, in any given situation there exist only a finite number of possibilities to choose a handle where a given story pattern could be applied and it is "easy" to enumerate all these possibilities.

Once we know the set of all possible subgraphs of a given graph we have to decide which of these subgraphs "fit" to the left-hand side of a given story pattern:

Definition A.9: Graph isomorphisms

Let G1, G2 $\in$ GC := GraphClass (SI)
We call a **bi-jective** function match: N_{G1}-> N_{G2} a graph isomorphism
:<=>

- match (N_{G1}) = N_{G2} where match(N) := { match(n) | n $\in$ N }
- match (E_{G1}) = E_{G2}
  where match(E) := { (match(sn), el, i, q, match(tn)) | (sn, el, i, q, tn) $\in$ E }
- $\forall$ n $\in$ N_{G1} holds nl_{G1}(n) $\in$ IsAs * (nl_{G2} (match (n)))
  // node label are assignment compatible
  where
  IsAs (nl) denotes the set of descendant node labels of nl as defined in the schema information SI
  and IsAs * denotes the transitive closure of this function.

- $\forall$ n $\in$ N_{G1}, a $\in$ A holds av_{G1}(n, a) = X implies av_{G2}(match (n), a) = X

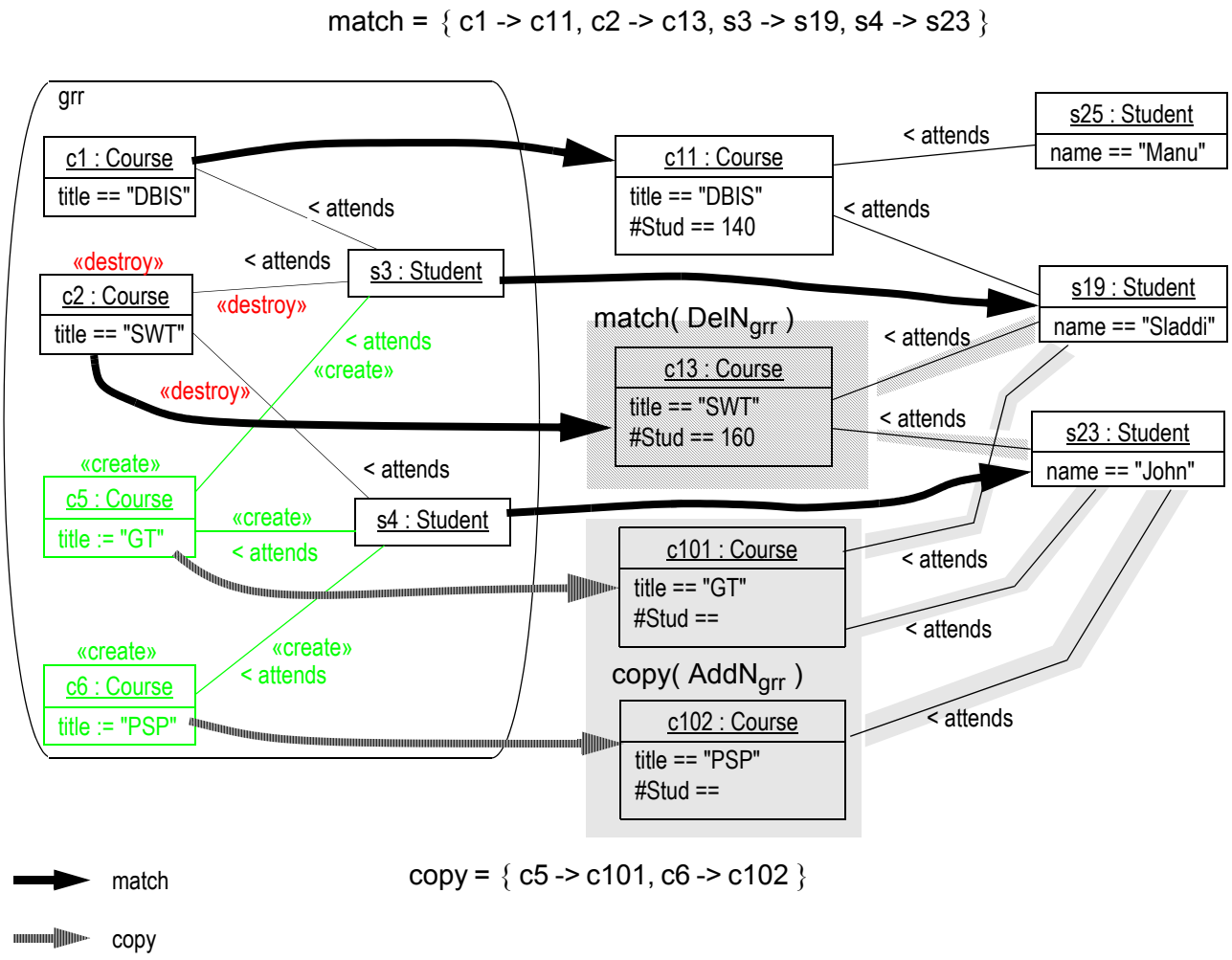
- ISOs(G1, G2) denotes the set of all possible graph isomorphisms for N_{G1} and N_{G2}

Isomorphisms are structure preserving functions. In case of graphs, structure preserving first of all means that the edges and attribute values in the image graph must correspond to the edges and attribute values in the original graph. In addition, we require that the node labelings are compatible. Note, in plain graphs we would require equal node labelings for the original and the image node. For object oriented graphs we relax this requirement. We want e.g. that a node of type Person contained in the left-hand side of a story pattern may match to any subclass of class Person, like Stud or Prof, too. Thus, we define that it suffices if the label nl1 of a node n in graph G1 (e.g. the left-hand side of a story pattern) is a direct or indirect superclass of the node label nl2 of the image of node n1 (i.e. the matching node in the host graph). To compute the set of subclasses of a given class we employ the IsAs relation of the corresponding schema information. Using the transitive closure of entries contained in the IsAs relation we compute the starting node label itself (applying IsAs zero times), the direct descendants (applying IsAs one time) and indirect descendants (applying IsAs multiple times).

Requiring compatible node labels, only, allows us to use nodes of a certain superclass like Person to match all kinds of objects that belong to class Person directly or indirectly. This is very convenient for writing story patterns because otherwise one would have to deal with each subclass of class Person

individually. On the other hand, it has the drawback that our notion of isomorphism is not symmetric. Generally, one would expect that if G1 is isomorphic to G2, i.e. structurally equivalent, then G2 should be structurally equivalent to G1 i.e. G2 should be isomorphic to G1, too. This does not hold for our definition of graph isomorphism. Let G1 be a graph consisting of a single node n1 of type Person and G2 a graph consisting of a single node n2 of type Prof. Then function match := { (n1, n2) } is a valid isomorphism from G1 to G2 since Person is a superclass of class Prof. We say a Prof is a valid match for a Person. But the reverse function $match^{-1}$:= { (n2, n1) } is *not* a valid graph isomorphism from G2 to G1, since n1 is only a Person and we are looking for a node of type Prof. However, giving up the symmetry property for isomorphism, our definition of isomorphism allows us to deal with inheritance in object oriented graphs, easily.

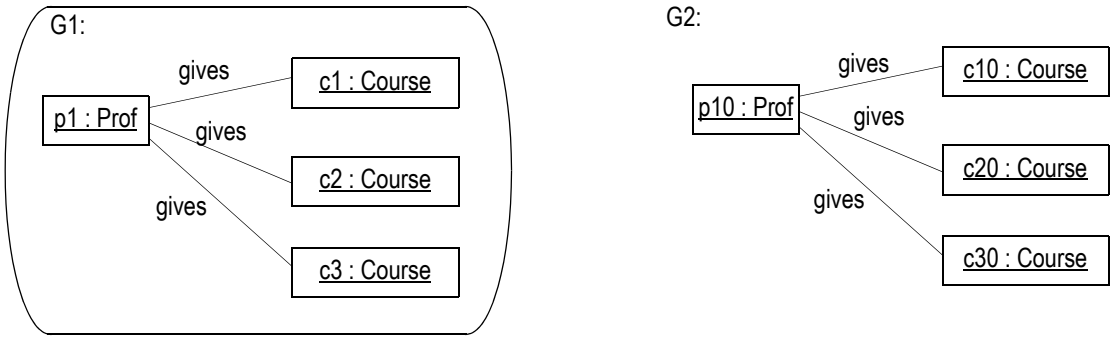
Example A.10: An example for match and copy



Example A.10 shows our sample story pattern and a sample host graph and one possible isomorphism match between the left-hand side of the story pattern and a subgraph SG of the shown host graph. The chosen subgraph SG is similar to the one of example A.8. Note, course c1 may be mapped on node c11 because they have the same type and their attribute title contains the same value. Node c1 does not contain a value for attribute #Stud. Thus, we had to choose a subgraph of the current host graph that does not contain the #Stud attribute, either.

Note, in general there may exist several isomorphisms between two given graphs G1 and G2. In example Example A.11 each course of graph G1 may be matched against any course of graph G2. This results in 3! possible matches. For two graphs, each consisting of just n nodes with the same label, there exist n! possible different matches. However, the number of possible matches is still finite and one may enumerate them, "easily".

Example A.11: Multiple graph isomorphisms



match 1 = { p1 -> p10, c1 -> c10, c2 -> c20, c3 -> c30 }

match 2 = { p1 -> p10, c1 -> c10, c2 -> c30, c3 -> c20 }

match 3 = { p1 -> p10, c1 -> c20, c2 -> c10, c3 -> c30 }

match 4 = { p1 -> p10, c1 -> c20, c2 -> c30, c3 -> c10 }

match 5 = { p1 -> p10, c1 -> c30, c2 -> c10, c3 -> c20 }

match 6 = { p1 -> p10, c1 -> c30, c2 -> c20, c3 -> c10 }

Provided with the notion of subgraphs and of graph isomorphisms, we are now able to define the semantics of a basic story pattern. Note, in general there are multiple possibilities to apply a story pattern to given graph. First, the given graph may contain multiple subgraphs that are isomorphic to the left-hand side of the story pattern. Second, there may exist multiple isomorphisms between the left-hand side and the chosen subgraph. Each of these different possibilities may result in a different result graph. On the other hand, the application of a story pattern to two different graphs can result in the same graph, by accident. Thus, the semantics of a story pattern is best defined as a relation between two graphs:

Definition A.12: Semantics of basic story patterns

Let LG, RG, G1, G2, IG $\in$ GC and grr := (LG, RG) be a story pattern.

Sem[grr] $\subseteq$ GC $\times$ GC or in other notation Sem[grr] : GC -> P(GC)

where (G1, G2) $\in$ Sem[grr] or in other notation G2 $\in$ Sem[grr] (G1)

:<=>

1. (Search)

   $\exists$ SG $\in$ GC such that SG <= G1 and $\exists$ match $\in$ ISOs(LG, SG) // we call SG a handle

2. (Deletion)

   $\exists$ IG <= G1 such that

   $N_{IG} = N_{G1}$ - $parts_{G1}$* (match ($DelN_{grr}$))

   where $parts_{G1}$ (N) := { tn | $\exists$ (sn, el, i, q, tn) $\in$ (E_{G1} - match ($DelE_{grr}$))

   where sn $\in$ N and aggregation $\in$ $assocType_{G1}$(el) }

   $E_{IG} = (E_{G1}$ - match ($DelE_{grr}$)) $\cap$ ($N_{IG} \times EL \times R \times$ AttrValues $\times N_{IG}$)

$nl_{IG} = nl_{G1}|_{N_{IG}}$

$av_{IG} = av_{G1}|_{N_{IG} \times A}$

3. (Creation)

(new nodes)

$\exists\, N_{new}$ such that $N_{new} \cap N_{G1} = \varnothing$ and $\exists$ bijective function copy
such that copy ($AddN_{grr}$) = N_{new}

where match determines copy uniquely, i.e.

match = match' => copy = copy' and

$N_{G2} = N_{IG} \cup N_{new}$

E_{G2} = (E_{IG} - replace (E_{IG}, E_{new})) $\cup$ E_{new}
where E_{new} := { (copyMatch (sn), el, i, q, copyMatch (tn)) |
$\exists$ (sn, el, j, q, tn) $\in$ $AddE_{grr}$ and
i == 1 + (max k in { (sn, el, k, q, tn") $\in$ E_{IG} }) if j != ε
i == j else } // so far add at the end of the list, only
where copyMatch (n) := match (n) if n $\in$ N_{IG}
copy (n) if n $\in$ N_{new}
and where replace (E_{IG}, E_{new}) := // to-one associations
{ (sn, el, i, q, tn) $\in$ E_{IG} | tgtCard (el) == one and (sn, el, i, q, tn') $\in$ E_{new}
or srcCard (el) == one and (sn', el, i', q', tn) $\in$ E_{new} }

nl_{G2}(n) = nl_{RG}(nr) if n = copy(nr) or = nl_{IG}(n) else

av_{G2}(n, a) = av_{RG}(nr, a) if (n = copy(nr) or n = match(nr)) and if av_{RG}(nr, a) is defined
or av_{IG}(n, a) otherwise

A pair of graphs (G1, G2) is element of the semantic relation Sem (grr) if G2 may result from the application of story pattern grr to graph G1. A story pattern grr is applied to a given graph G1 as follows. First, one chooses a subgraph SG of the given graph G1 that is isomorphic to LG, the left-hand side of the story pattern. In addition, one chooses an isomorphism match that matches the nodes of LG to the nodes of the subgraph SG. We call SG a handle for the application of the story pattern grr.

The chosen handle SG uniquely determines an intermediate graph IG that results from G1 by removing elements that are matched by the left-hand side LG but are not part of the right-hand side RG. IG is constructed in 4 steps. First, we delete nodes. $DelN_{grr} := N_{LG} - N_{RG}$ describes the set of nodes that are explicitly deleted by story pattern grr, cf. Definition A.5. Actually, not the nodes in $DelN_{grr}$ are deleted but their matches within host graph G1, i.e. match ($DelN_{grr}$). Usually this are all nodes that are deleted. However, object oriented graphs may contain aggregation relationships. As discussed in chapter A.2 we interpret aggregation as existence dependency. This means, when a parent object is deleted all kid objects that are still connected to the parent are deleted, too. Note, the deletion of a kid object k may cause the recursive deletion of kids of k. In our definition this is reflected using function parts. Function parts looks-up the schema information of edges attached to nodes passed as argument. In case the keyword aggregation is element of the assocType of an attached edge, the target node becomes part of the result. However, if the corresponding aggregation edge is explicitly deleted by the story pattern, this rescues the part from dieing with its parent. Building the transitive closure parts*

generates all nodes that are parts of $DelN_{grr}$ directly or indirectly (including zero applications of parts, i.e. $DelN_{grr}$ itself). Thus, we delete all nodes that are to be deleted explicitly and all nodes that are "parts" of these nodes directly or indirectly.

Next we delete edges. $DelE_{grr} := E_{LG} - E_{RG}$ defines the set of edges that are explicitly deleted by story pattern grr. Again, actually the images of these edges are deleted. We compute these images by applying function match to the source and target nodes of the edges in $DelE_{grr}$. In addition, we have to take care of edges attached to nodes that have been deleted. In order to yield a valid graph, such edges need to be deleted, too. $E_{possible} := (N_{IG} \times EL \times R \times AttrValues \times N_{IG})$ denotes the set of all edges that a valid graph with node set N_{IG} could contain. Thus intersecting the set of remaining edges with the set of all allowed edges $E_{possible}$ deletes all edges attached to deleted nodes, as required.

The node labeling function nl_{IG} is just the restriction of nl_{G1} to the remaining nodes N_{IG}. The same holds for the attribute value function av_{IG}. Altogether this defines the intermediate graph IG.

In the third story pattern execution step we add new elements to the intermediate graph IG until graph G2 results. This again requires 4 steps. First we create new nodes. $AddN_{grr} := N_{RG} - N_{LG}$ describes the set of nodes that are explicitly created by story pattern grr, cf. Definition A.5. Actually, not the nodes in $AddN_{grr}$ are created but we create new nodes N_{new} which are copies of them. In order to determine which node $n' \in N_{new}$ corresponds to which node $n \in AddN_{grr}$ we define function copy. Special care is required in choosing the new nodes N_{new}. Of course, one must not use nodes that are already part of the graph, i.e. no elements of N_{IG}. We could reuse some of the nodes that we just have deleted. Or we could use an arbitrary set of new nodes (the node labeling function will be defined later). There may exist an infinite number of possible sets N_{new}. Each of these sets would finally create a different result graph G2. Thus the semantics of a story pattern Sem (grr) could become an infinite relation. However the resulting graphs G2 would differ in the identity of the new nodes N_{new}, only. This is not intended. Thus we do not allow an arbitrary choice of the node set N_{new} but we require that N_{new} is uniquely determined by the chosen handle. This is achieved by requiring that equal functions match result in equal copy functions. The resulting node set N_{G2} is then defined as the disjoint union of N_{IG} and N_{new}.

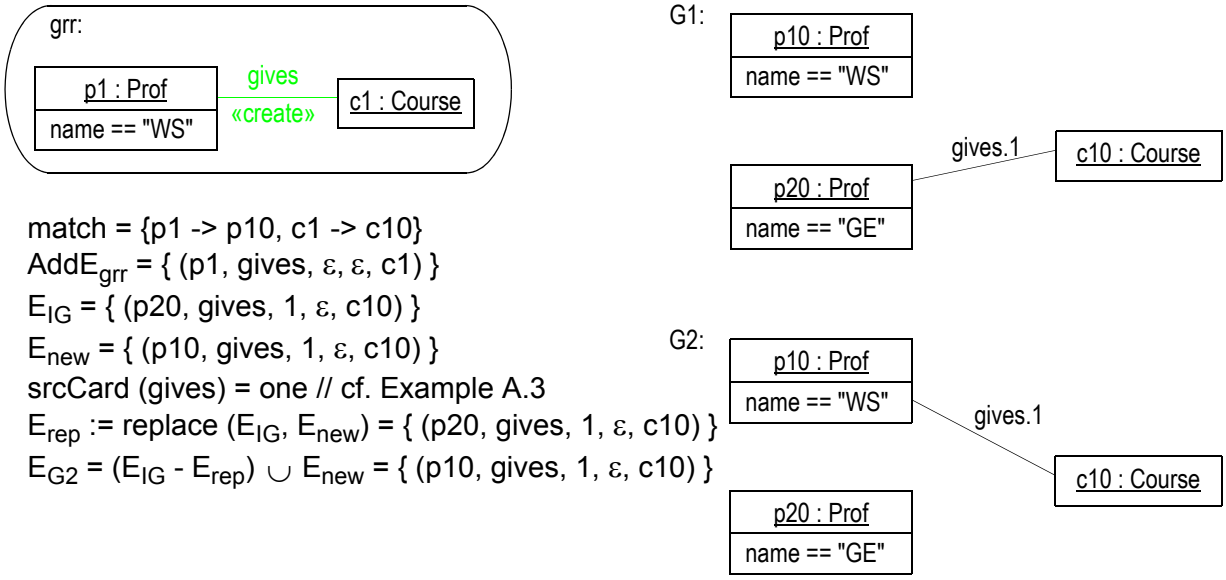
Next, we add edges. $AddE_{grr} := E_{RG} - E_{LG}$ defines the set of edges that are explicitly added by story pattern grr. Again, actually "copies" of these edges are added. We construct the set of new edges E_{new} from $AddE_{grr}$ by replacing the nodes in $AddE_{grr}$ by either their matches or their copies in N_{G2}.

In addition we have to deal with ordered associations. In our approach we employ indices to define an order for a set of neighbors reached via an ordered association. Adding a new neighbor always requires to provide a position within the list of existing neighbors where the new neighbor shall be inserted. That means, one has to provide an index i for the corresponding edges. Frequently, one may want to append the new neighbor to the list of existing neighbors. This can be achieved using an index that is greater than all existing indices. However, such an index can not be provided by the story pattern itself, since the length of the list of neighbors is not yet known when the rule is written. Instead, the insertion index should be computed at runtime. Full story patterns provide language means to compute such indices from existing neighbors, cf. chapter A.6.9. One may insert new neighbors directly before or after known other neighbors or one may add new neighbors at the beginning or at the end of the existing list. We will define these language elements later.

In a basic story pattern one either has to provide an explicit index for list edges or the default case applies, i.e. the new neighbor is appended to the list of existing neighbors. We achieve this by computing the index i of new edges as the maximum of all existing edges with the same edge label leaving the same source node, increased by 1. This is done only, if the story pattern does not provide an explicit index j for the corresponding edge. Otherwise, the explicit index j is used.

In a plain graph it would now suffice to construct E_{G2} as the union of E_{IG} and E_{new}. In an object oriented graph this needs more care due to the cardinality constraints of to-one associations, cf. Example A.13.

Example A.13: Replacing a to-one link



match = {p1 -> p10, c1 -> c10}
$AddE_{grr}$ = { (p1, gives, ε, ε, c1) }
E_{IG} = { (p20, gives, 1, ε, c10) }
E_{new} = { (p10, gives, 1, ε, c10) }
srcCard (gives) = one // cf. Example A.3
E_{rep} := replace (E_{IG}, E_{new}) = { (p20, gives, 1, ε, c10) }
$E_{G2} = (E_{IG} - E_{rep}) \cup E_{new}$ = { (p10, gives, 1, ε, c10) }

In Example A.13 we apply story pattern grr to graph G1 using the (only possible) match p1->p10 and c1->c10. According to grr we have to add a gives edge between nodes p10 and c10. However, course c10 is already given by professor p20. According to the class diagram shown in Example A.3, gives is a to-one associations with respect to class Prof. Thus a course may be given by at most one professor. Assigning a new professor to course c10 should therefore replace the edge to the old professor.

In our semantics definition we achieve this behavior using function replace. Function replace looks up the source and target cardinality of edges in E_{new}. In case of a target cardinality one, any existing edge in E_{IG} with the same source node, label, index, and qualifier is going to be replaced by a new edge. In case of a source cardinality of one, any existing edge with the same label is going to be replaced, independently of the index or qualifier. Note, our data model allows indices and qualifiers for the forward (left-to-right) direction of associations, only. In forwarded direction, multiple edges with the same label but different indices or qualifiers are allowed. An old edge is replaced only if a new edge with the same source node, label, index, and qualifier is added. In the reverse (right-to-left) direction, cardinality of one means that a given node might be the target of at-most one edge of this type without respect to qualifiers and indices.

Thus, the set E_{G2} of edges of the resulting graph is constructed from the set E_{IG} of edges of the intermediate graph by subtracting the edges which are going to be replaced and by adding the computed set of new edges, afterwards.

The node labeling of graph G2 is taken either from the intermediate graph IG or from the right-hand side of the story pattern. Similarly, the attribute value function is taken from the intermediate graph IG. However, in case the right-hand side RG provides a (new) value for some attribute, the attribute value function is (re)defined, accordingly. Note, attribute values given in the story pattern automatically replace possibly existing old attribute values.

To summarize, our definition of the semantics of story patterns takes care of the special features of object oriented graphs like qualified and ordered associations, inheritance, aggregations, and cardinality constraints. Choosing a subgraph as handle for the application of a story pattern allows to omit edge indices or qualifiers. This allows to write story patterns that may look-up specific indices and

qualifiers or that match edges of the corresponding type independently of their indices and qualifiers. Our definition of graph isomorphism takes care of inheritance. A node of a certain superclass A in a story pattern may match nodes of class A or any of its direct or indirect subclasses. The semantics definition of the deletion step takes special care of aggregations by using the parts function. If a node is deleted explicitly, all its direct or indirect "parts" are deleted, too. Note, this implements existence dependency but not co-incident lifetime, since there are no restrictions to disconnect parts from their parents or to add new parts, later on. Adding edges deals with ordered associations, specifically. In the default case, we automatically compute an index for new elements added to such a list, that is computed as the maximal existing index plus 1. In chapter A.6.9 we will discuss language features, allowing to add new elements to such a list between existing elements or in front of them. Finally, adding edges is defined to take special care of to-one associations. If an object is allowed to have at-most one neighbor, assigning a new neighbor to it automatically replaces the old neighborship, i.e. the old neighbor edge is deleted.

A.4 Basic story diagrams

For complex operations, a single story pattern may not suffice. One may want to iterate some substep(s) or to execute different substeps in different situations. In the UML, such control flow issues may be modeled using activity diagrams. Unfortunately, UML activity diagrams specify the actual actions as pseudo code or natural language descriptions, only. As discussed in chapter 2.3, we replace these informal action descriptions by story patterns or collaboration diagrams. The resulting combination is called *story diagrams*. In the next chapter, we will use story diagrams for the specification of method bodies. Within a consistent and complete system specification, each method declared in a class diagram is specified by exactly one story diagram. To achieve well defined method specifications, in this chapter we provide an inductive definition of all possible story diagrams and their semantics. In each step we provide a textual as well as a graphical notation for the defined construct.

The most elementary form of a story diagram consists of a single activity containing a single story pattern:

Definition A.14: One-Pattern Story Diagrams

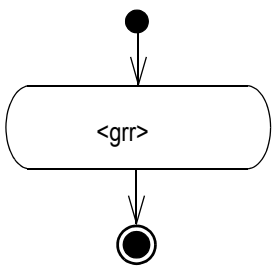
Let Grr := (LR, RG) be a story pattern and G a graph, then

try (grr) is a story diagram with

try (grr) is applicable to G by definition if and only if $\exists$ SG <= G, match $\in$ ISOs(LG, SG)

Sem [try (grr)] := { (G1, G2) | (G1, G2) $\exists$ Sem [grr]
or (G1 = G2 and grr is not applicable to G1) }

Graphically, try (grr) is shown as:



where <grr> represents the graphical notation of story pattern grr.

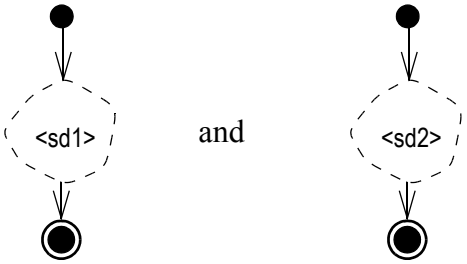
Thus, any story pattern may be used as a story diagram containing a single activity showing just the story pattern. Note, for story diagrams we introduce the notion of *applicability*. We will use the notion of applicability to define branching, later. Note, a story pattern grr may not have a match in a given graph G1. In that case, no pair of graphs (G1, G2) is in the semantics of the story pattern. However,

for a story diagram this means only, that the story pattern was not applicable. In that case the given graph is just not changed, i.e. the pair (G1, G1) is in the semantics of try (grr). This allows us e.g. to try some story pattern grr1 and if it fails to go on with some other story pattern grr2.

The most simple control structure is the sequential composition of two story diagrams:

Definition A.15: Sequential composition of story diagrams

Let sd1, sd2 be story diagrams with graphical depictions



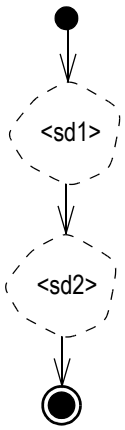
then

sd1; sd2 is a story diagram with

sd1; sd2 is applicable if sd2 is applicable and

Sem [sd1; sd2] := { (G1, G3) | ∃ G2 such that(G1, G2) ∈ Sem [sd1]
and (G2, G3) ∈ Sem [sd2] }

Graphically, the sequential composition of sd1 and sd2 is shown as:



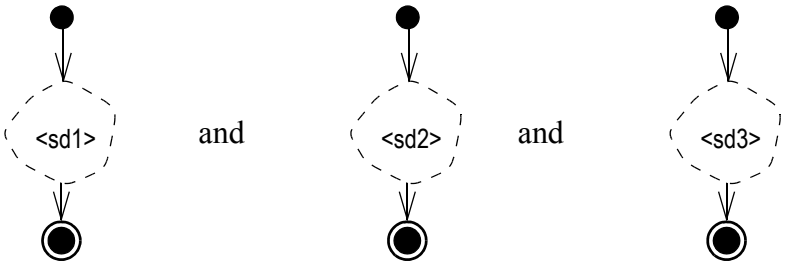
Note, the graphical representation of story diagram always has exactly one start activity ● and exactly one stop activity ◉. In the graphical notation of the sequential composition of two story diagrams sd1 and sd2 the transition reaching the stop activity of sd1 and the transition leaving the start activity of sd2 are glued together and the intermediate stop and start activity are omitted. This results in a story diagram showing again exactly one start activity and exactly one stop activity.

The semantics of the sequential composition corresponds to the sequential application of the story diagrams. The sequential composition is applicable iff the second story diagram is applicable.

In addition, we need some conditional control flow:

Definition A.16: If-composition of story diagrams

Let sd1, sd2, and sd3 be story diagrams with graphical depictions



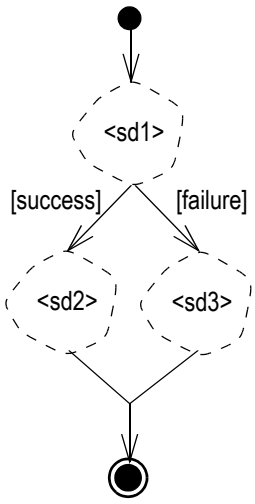
then

if sd1 then sd2 else sd3 end is a story diagram with:

if sd1 then sd2 else sd3 end is applicable iff
sd1 is applicable and sd2 is applicable
or sd1 is not applicable and sd3 is applicable

Sem [if sd1 then sd2 else sd3 end] :=
Sem [sd1; sd2] if sd1 is applicable
Sem [sd1; sd3] else

Graphically, the if-composition of sd1, sd2, and sd3 is shown as:

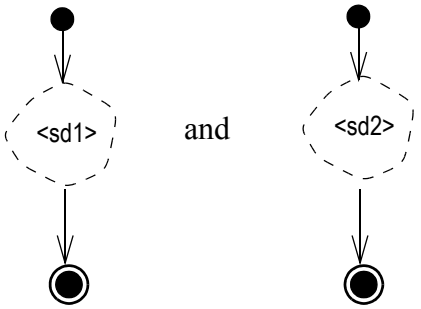


Note, in the graphical notation of the if-composition the transition leaving sd1 is split into a [success] and a [failure] transition leading to sd2 and sd3, respectively. Similarly, the transitions leaving sd2 and sd3 are glued together, guaranteeing that the if-composition has exactly one leaving transition. If no ambiguity arises, the two leaving transitions may be connected to the first activity of a subsequent activity, individually.

The semantics of the if-composition depends on the applicability of the first story diagram. If it is applicable then the [success] branch is executed, otherwise the [failure] branch is executed.

Definition A.17: While-composition

Let sd1 and sd2 be story diagrams with graphical depictions



then

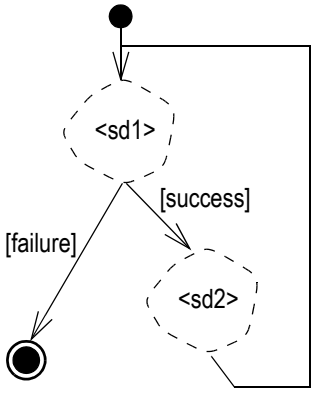
while sd1 do sd2 end is a story diagram with

while sd1 do sd2 end is never applicable and

Sem [while sd1 do sd2 end] :=

Sem [sd1]	if sd1 is not applicable
Sem [sd1; sd2, while sd1 do sd2 end]	else

Graphically, the while-composition of sd1 and sd2 is shown as:



Note, the transition leaving story diagram sd2 may also lead to the first activity of sd1, if no ambiguities arise.

The semantics of the while composition is recursively defined. If sd1 is not applicable the recursion terminates. Otherwise, sd1 and sd2 are executed and the while-composition is employed recursively. If sd1 is not applicable it is nevertheless executed one time. Story diagram sd1 could e.g. represent a sequential composition of two subdiagrams sd11;sd12. In that case sd1 is not applicable if its last subdiagram sd12 is not applicable. However, the first subdiagram sd11 may be applicable and modify the graph. Thus, sd1 may have effects although it is not applicable.

Similarly, a while-composition is defined to be never applicable. The while-composition will terminate as soon as the story diagram sd1 in its condition is no longer applicable. Thus, the last subdiagram executed by a while-composition has not been applicable by definition and therefore the while composition is never applicable. Note, applicable corresponds to the success of the last activity, only. Previous steps may have been executed successfully.

A.5 Methods

In this chapter we will introduce story diagrams as the implementation of methods declared in class diagrams. This will include the definition of polymorphic method call operations including parameter handling, recursion, and local variables.

Definition A.18: Object oriented specifications

OOSpec := (SI, SDs, Methods, bind, main) with

SI := (NL, EL, A, IsAs, Assocs, Attrs) a schema information, cf. Definition A.1

SDs a set of story diagrams

Methods a set of method names

bind Func (NL × Methods) → SDs

main ∈ Sds the main story diagram

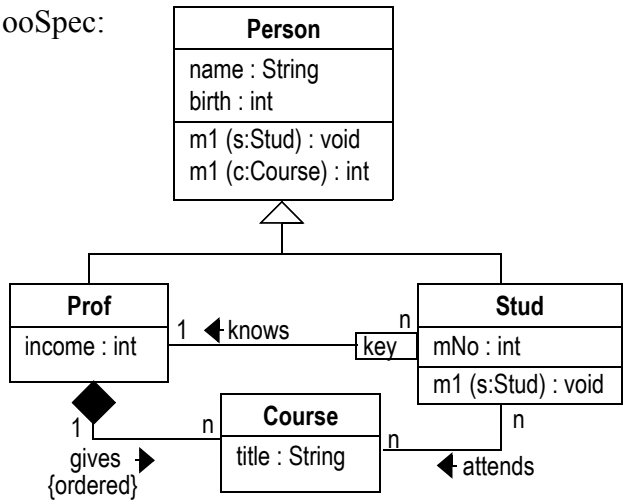
Sem [OOSpec] := Sem [main]

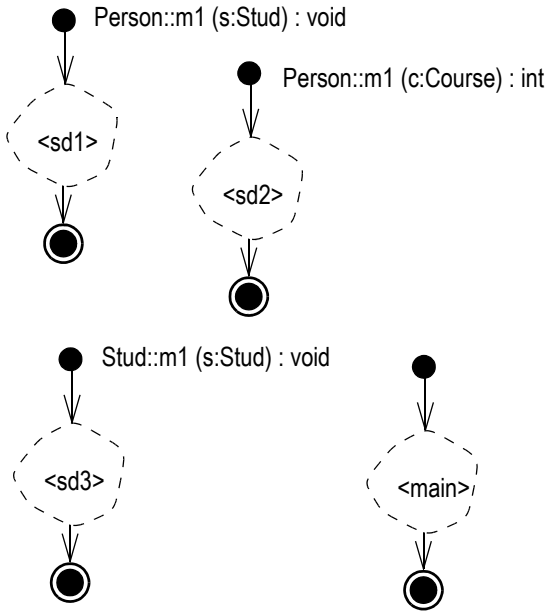
So far, an object oriented specification consists of a schema *SI*, a set of story diagrams *SDs*, a set of method names *Methods*, and a method lookup function *bind*. In addition, an object oriented specification contains one *main* story diagram that is called to execute the specification. Thus, the semantics of an object oriented specification is defined through the semantics of its main story diagram.

Note, object oriented languages commonly allow overloading and polymorphic method calls. Overloading means that one class may provide several methods with the same plain method name but different parameter signatures. Polymorphic method calls mean that subclasses may provide their own implementation for inherited method (names) and that the implementation which is called is determined by the runtime type of the target object. To deal with overloading, we employ method names that encode the parameter types and the result type. For example a method m1 (s:Stud) : void would be encoded by name m1_Stud_void. Polymorphic method calls will be handled using the bind function. The bind function takes a node label nl, i.e. the runtime type of the target object, and a method name and provides the corresponding method implementation, i.e. the corresponding story diagram. Graphically, the method names are shown in the method compartment of the corresponding classes in the usual UML notation. The binding of story diagrams to methods is indicated by attaching the corresponding method declaration to the start activity of the story diagram.

Example A.19: An object oriented specification in UML notation

ooSpec:





SI like in Example A.3

SDs := {sd1, sd2, sd3, main}

Methods := { m1_Stud_void, m1_Course_int}

bind := {(Person, m1_Stud_void) → sd1, (Person, m1_Course_int) → sd2,
(Prof, m1_Stud_void) → sd1, (Prof, m1_Course_int) → sd2,
(Stud, m1_Stud_void) → sd3, (Stud, m1_Course_int) → sd2 }

main := main

Note, function bind has to reflect existing inheritance relationships. This means, if a parent class A declares some method m_T and if function bind assigns story diagram sd to this combination of class name and method name then function bind has to provide similar assignments for all subclasses of A if the subclass does not provide its own implementation.

Note, our definition does not yet deal with possible name clashes caused by multiple inheritance. We already restricted multiple inheritance in class diagrams to the cases that correspond to the concepts of Java. In Java clashes of inherited method implementations cannot occur since multiple inheritance is allowed for interface classes, only, and interface classes cannot provide method implementations. In languages like Eiffel or C++ such implementation clashes could occur. However, such languages deal with this problem at compile time. For example, if an Eiffel class D inherits two different implementations for a method m(), then the user has to provide a new implementation for this method within class D in order to resolve the ambiguity. Thus, in a consistent specification method implementation clashes do not occur and function bind is well defined.

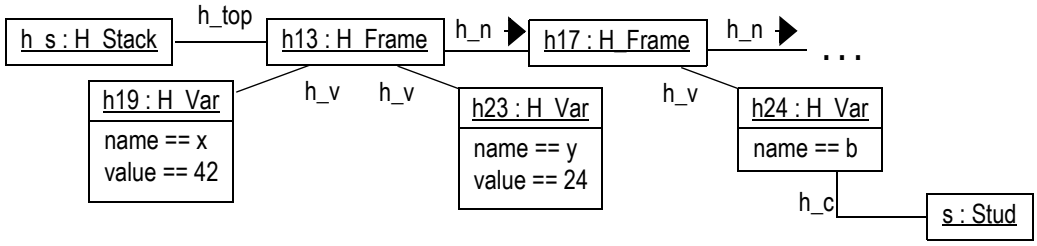
Provided with the notion of methods, we are now able to define the semantics of method calls. Our methods may have parameters and local variables. Of course, we allow recursive and reentrant method calls. Thus, to implement method calls we need something like a *procedure call stack*. However, we do not support nested method declarations as e.g. in Pascal. Thus, our procedure call stack needs no static link, cf. [ASU86].

To handle global variables and static methods we just assume that such elements are attached to our unique stack object. In this way we provide, e.g. mathematical operations and other library functions.

So far, our semantics definition employs story patterns as basic operations. Thus, it is quite natural to model the required procedure call stack as a special extension of our graphs and to define the opera-

tions modifying the procedure call stack via additional story patterns. Without loss of generality, we assume that user provided graph schemata may not use node or edge labels beginning with "H_" or "h_". This allows us to employ some help structures within a given object oriented specification that begin with "H_" or "h_". Such help structures cannot interfere with user defined names. Thus a possible procedure call stack could look like in:

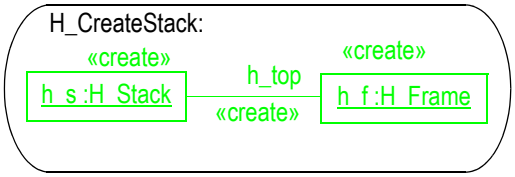
Example A.20: A sample procedure call stack



We employ exactly one node with label H_Stack. This node provides an h_top link to the top-level call frame. Nodes of type H_Frame represent procedure call frames. Call frames provide h_v links to contained local variables and parameters which are represented using H_Var nodes. Each H_Var node contains a name attribute indicating the name of the corresponding variable and a value attribute containing the current attribute value. Instead of an attribute value, an H_Var node may provide an h_c link to another node in the graph.

Definition A.21: Procedure call stack initialization

Without loss of generality the first activity of the main story diagram of any object oriented specification must be the story pattern H_CreateStack:



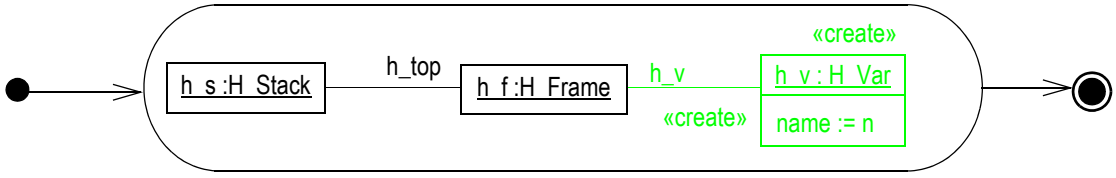
Thus, the main story diagram of a specification always first has to create an initial procedure call stack consisting of an H_Stack node and the top level call frame. Provided with this auxiliary call stack subgraph we are able to define the declaration of local variables and the execution of method calls.

Definition A.22: Declaration of local variables

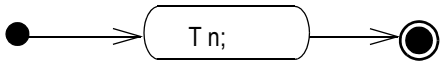
Let T be a node label or an attribute type and n a unique name, then

T n; is a story diagram with

Sem [T n;] := Sem [declare_T_n] where declare_T_n is the following story diagram



Graphically, a variable declaration is just shown as the content of an activity shape:



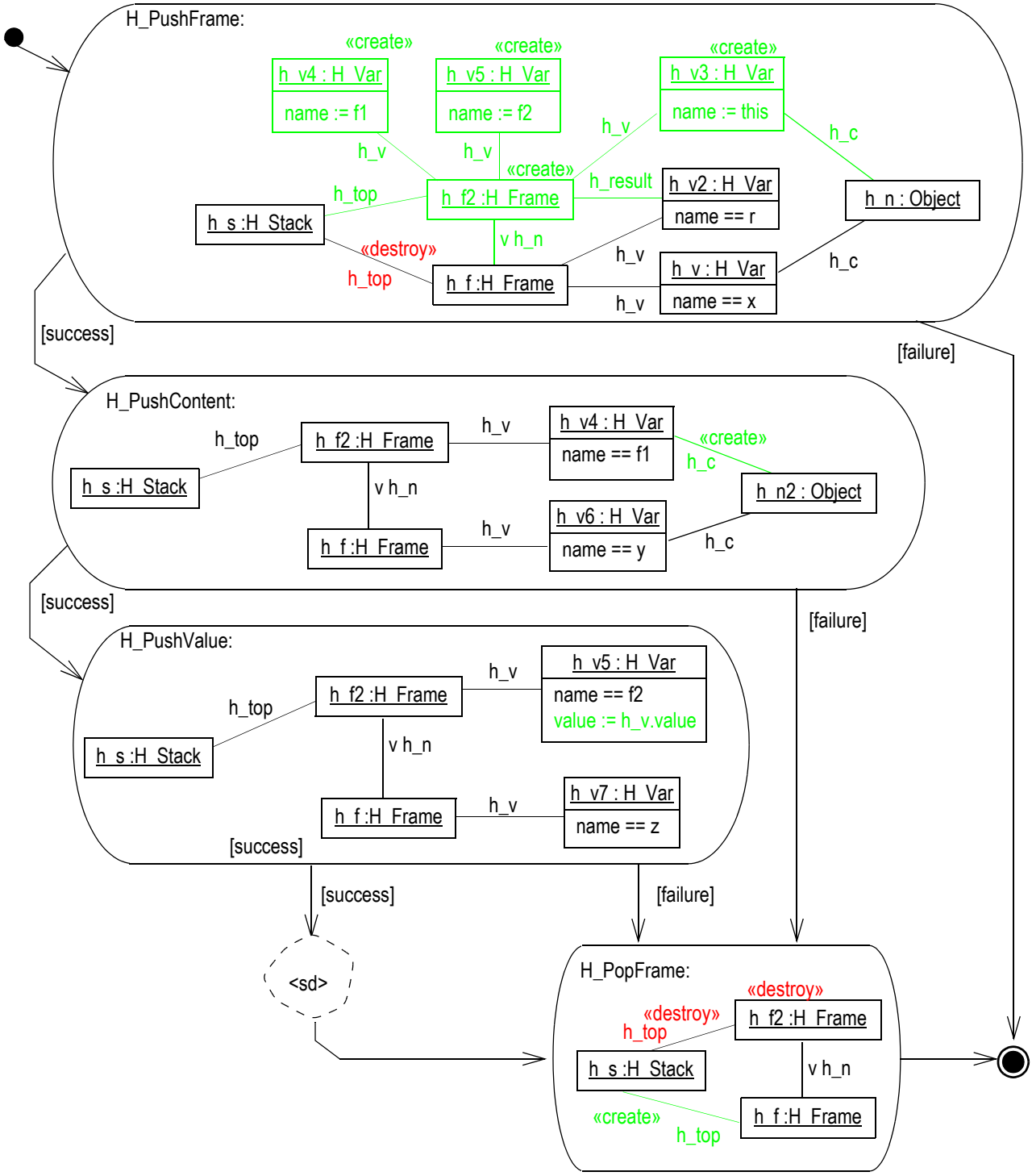
A variable declaration just creates an H_Var node at the current procedure call frame. Note, the H_Var node just stores the variable name but not its type. The static type of a variable is important for context sensitive checks at compile time. At runtime this information is not necessary.

Definition A.23: Method calls

Let r, x, y, z be local variables and m (f1 : T1, f2 : T2) : T3 a method declaration creating method name m_T1_T2_T3, then

r = x.m (y, z); is a story diagram with

Sem [r = x.m (y, z);] :=
the semantics of the following story diagram



	if the var node representing local variable x has an h_c link to a node h_n and l (h_n) = nl and bind (nl, m_T1_T2_T3) = sd holds and the provided actual parameters exist and fit to the formal parameters
Sem [nop]	otherwise, i.e. if x has no h_c link or nl does not provide an implementation for m_T1_T2_T3 or one of the employed variables does not exist.

To call method m_T1_T2_T3 on variable x we first have to look-up the corresponding implementation sd. Therefore, we lookup the current value of variable x by following the h_c link to its content node h_n. Then we use the labeling function l to derive the runtime type of node h_n and function bind to identify the implementing story diagram sd.

If we are able to compute the method implementation sd, the first activity of the call execution H_PushFrame creates a new procedure call frame on our call stack and the h_top link is redirected to the new call frame. Next, we add the standard variable this to the new call frame that is used to store the invocation target, i.e. the currently active object. Therefore, the variable this gets an h_c link to node h_n, the content of variable x. In our example, the return value of the called method is assigned to variable r. Within our call stack, the target variable h_v2 of the assign statement is marked via a special h_result link attaching it to the new call frame h_f2. In addition, we add variables for all formal parameters to the new call frame.

Once the new call frame is established, we pass the actual parameter values to the formal parameters. In our example there are two actual parameters y and z. Actual parameter y is an example for a variable that contains an h_c link to a graph node. For such parameters story pattern H_PushContent copies the h_c link from the actual to the formal parameter. Note, this step might fail if the actual parameter does not contain an h_c link, i.e, if the actual parameter is a null-pointer. In that case the activity H_PushContent is not applicable and we just proceed with the next activity. This would leave the formal parameter without an h_c link, too, which would correctly reflect the actual parameter value. Actual parameter z is an example for a parameter of a plain attribute type like int or boolean. For such parameter story pattern H_PushValue copies the value of the actual parameter variable to the formal parameter variable. At this point of the execution the new procedure call frame is well established and we execute the method body sd. Note, by definition sd has exactly one entry and one exit transition and thus it is easy to embed sd in the help activities.

Once story diagram sd is executed, the method is ready to return to its caller. This is done using story pattern H_PopFrame. Story pattern H_PopFrame destroys the top procedure call frame h_f2 and redirects the h_top link to the previous call frame h_f determined by the corresponding h_n link. This re-establishes the caller's environment of local variables.

Note, the execution of a method call may fail, because variable x does not refer to an object. This corresponds to a method invocation on a variable with value null. Modern programming languages deal with this situation via a runtime exception concept, cf. [Java]. In our semantics definition, we have not yet introduced such an exception concept. Instead we deal with the situation by just ignoring the method call and doing nothing.

In addition, the method invocation could fail, because the runtime type of the invocation target does not provide an implementation for the called method or because the call does not provide the correct number and types of actual parameters. However, these kinds of errors may be checked statically at compile time as part of the context sensitive semantics. Here, we assume that the program is correct with respect to the context sensitive semantics, i.e. such problems must not occur.

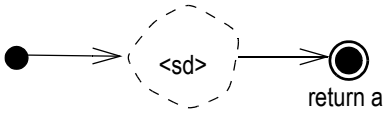
Note, the story patterns H_PopValue, H_PopContent, and H_PopFrame destroy the top level call frame object but not all of the variable objects attached to such call frames. This leaves garbage nodes

of type H_Var within the graph. These garbage nodes do not interfere with the further execution of the specification since our help story pattern only look-up H_Var nodes that are still attached to procedure call frames via h_v links. However, in the next chapter we will define additional language means for story patterns that easily allow to collect and remove all these garbage nodes.

To conclude the definition of method invocations we introduce a short-hand notation for dealing with return values:

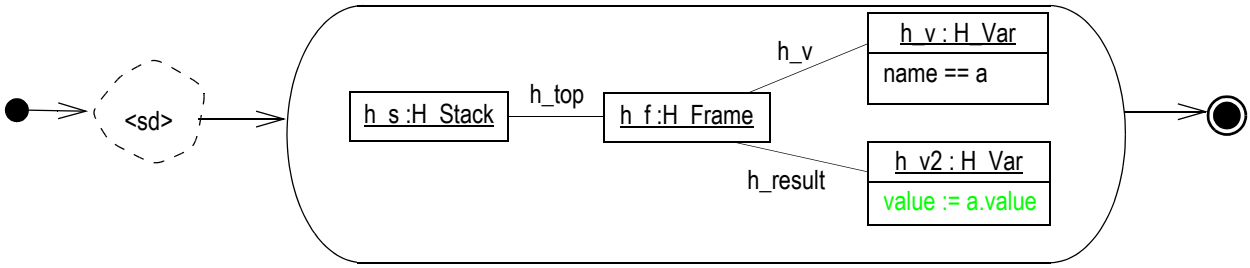
Definition A.24: Return clause

A story diagram may provide a return clause:



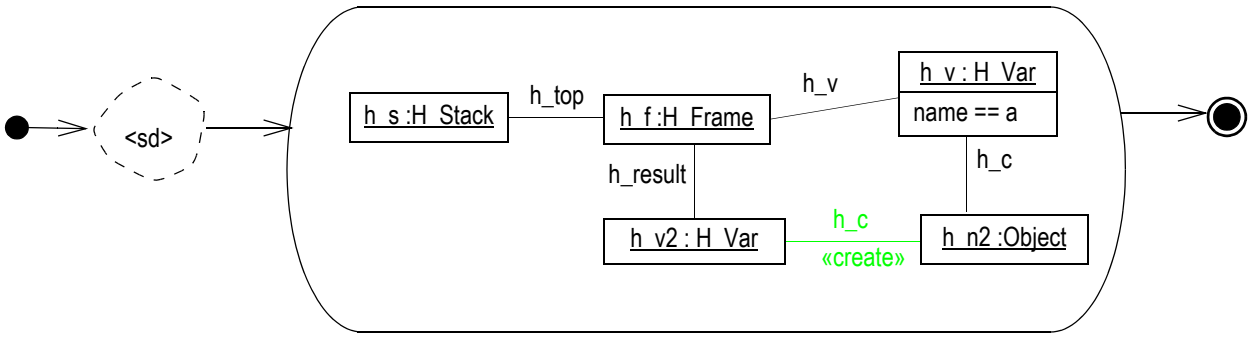
Such a return clause is a short-hand notation for an explicit story pattern copying the value or the content of variable a to the assign target marked via the h_result link.

Sem [return a] := the semantics of



if a has a simple attribute type like int or boolean,

or the semantics of



otherwise.

Note, in principle this definition allows to continue the execution of a story diagram after a return operation. We have defined the semantics in this way, since the semantics of sequential, if-, and while-composition of story diagrams require that it is always possible to combine existing story diagrams via these controls structures. However, if a story diagram with a return clause is going to be embedded in some surrounding story diagram, first the return clause short-hand has to be replaced by the explicit assignment to the h_result variable. Note, intermediate assignments to the result variable are no problem, since the called method cannot access the local variables of its caller in any way but only the result variable.

We conclude this chapter with the definition of constants and attribute expressions:

Definition A.25: Constants and attribute expressions

Constants like natural numbers, boolean values, strings, or real numbers may be used as actual parameters or in attribute expressions.

A constant k is a short-hand for a story pattern that assigns the corresponding value to a internal help variable h_k and where this help variable h_k is used instead of the constant.

We define that the unique stack object h_s :H_Stack provides method implementations for all common mathematical operators and functions like e.g. +, *, ^2, <=, substring, ... The use of such an operator or function within a mathematical expression is a a short-hand for a call to the corresponding method where the result is assigned to a unique help variable and where the operator and its arguments are replaced by this help variable. For example:

Sem [x = y.f (a +2) * b] :=

```
Sem [   h_k2 = h_s.H_k2 ();
        h_1 = h_s.plus (a, h_k2);
        h_2 = y.f (h_1);
        x = h_s.mult (h_2, b) ]
```

Note, this definition of the semantics of mathematical expressions is not very elaborated. However, formalizing mathematical expressions is well understood (and tedious) and would not add value to the results of this work.

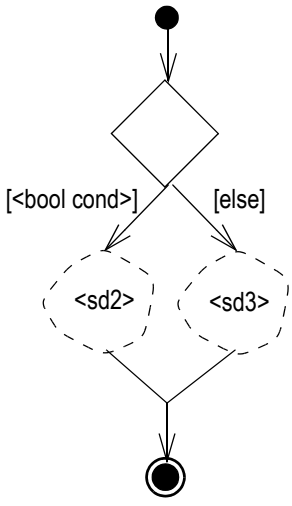
Definition A.26: Using boolean expressions as transition guards

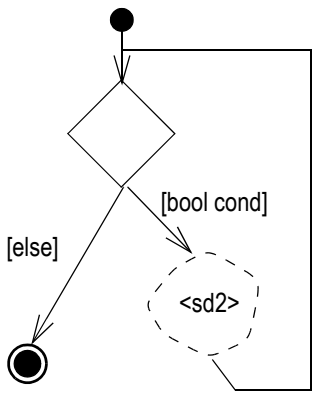
The boolean operators "and" and "or" are left associative and perform short circuit evaluation.

Boolean conditions may be used as the first story diagram of an if- or while-composition. In that case a boolean condition is considered as "applicable" iff it evaluates to true.

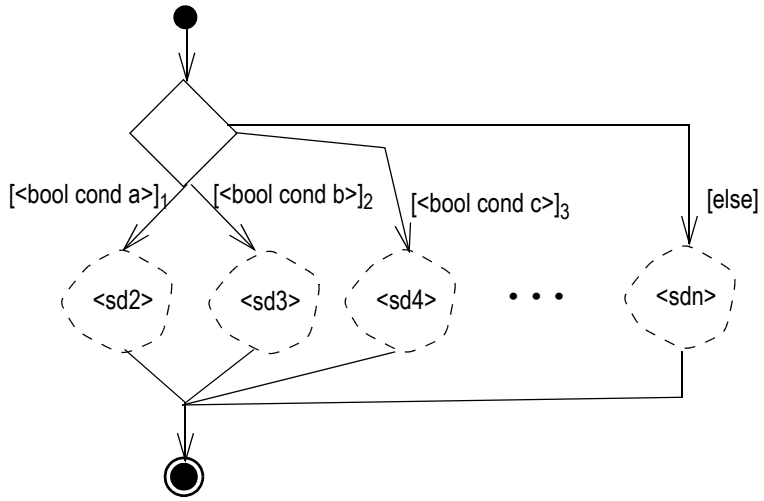
Story diagrams without a return clause may be used as parameters for boolean operators. A story diagram without a return clause evaluates to true if it is applicable. It evaluates to false, otherwise.

Graphically, a boolean condition used as first story diagram of an if- or while-composition may be shown using a diamond shaped activity where the leaving success transition is replaced by the boolean condition in square brackets and the leaving failure transition is replaced by the keyword else in square brackets:





In case of a chain of if-then-else-if compositions one may use a single diamond shape with multiple outgoing transitions. However, in this case ordering numbers subscribed to the boolean conditions define the order of evaluation for the different alternatives:



Note, UML activity diagrams require that multiple outgoing transitions leaving a given activity shall employ mutually exclusive guard conditions. In general, it is not possible to check this feature statically, i.e. at compile time. In addition, our conditions may call user defined methods that may have side effects. Therefore, the sequence of evaluation for different guard conditions, or in other terms the exact nesting of the depicted if-compositions, has semantical relevance. We solve this problem by extending the syntax of transition guards by ordering numbers that define the order of evaluation. The first guard that is fulfilled fires. If the guards are actually mutual exclusive and do not have side effects, our semantics meet the UML semantics.

A.6 Additional story pattern and story diagram features

So far, we have defined the semantics of basic story patterns and basic story diagrams. Theoretically, these language features have the same computational power as Turing machines. This chapter will propose some additional language features that facilitate a more convenient modeling of frequently occurring situations.

A.6.1 Bound variables

One frequent situation is, that an object structure modification is too complex for a single story pattern. Thus one splits the operation into several simple story patterns that are embedded into a common story diagram. Usually, the first story pattern chooses a subgraph to be modified and one wants to apply the subsequent steps to the same subgraph. So far, each story pattern chooses the subgraph to be modified, individually. Thus, we need appropriate means in order to path handles from one story pattern application to the next. A direct way to achieve this is to create a unique cursor node within the first story pattern that marks certain parts of the handle via appropriate links. Subsequent story patterns can look-up this cursor node in their left-hand side and thus guarantee to modify the same subgraph of the host graph. In case of recursion, the marker node could be attached to the current procedure call frame node and looking up the marker could include the unique stack node and the top call frame.

To deal with such markers is tedious and error prone. In addition, such situations occur frequently. Thus, we have decided to provide additional language features that allow to pass parts of a match from story pattern to others more conveniently. We introduce implicit variables that store all nodes matched

by the left-hand side of a story pattern or created by the story pattern execution. A special short-hand notations allows subsequent story patterns to look up the contents of these variables easily in order to modify the same subgraph.

Definition A.27: Implicit story pattern variables

1.: For each story diagram sd, that serves as a method implementation, we implicitly add an initial story pattern that creates H_Var nodes at the top level procedure call frame for all objects employed in any story pattern in sd. In the name attribute of the H_Var nodes we store the names of the corresponding objects in the story patterns.

2.: Each user provided story pattern grr is implicitly extended in the following way:

2.1.: We add the unique stack node and the top call frame and the connecting h_top link to the left-hand and right-hand side of the story pattern.

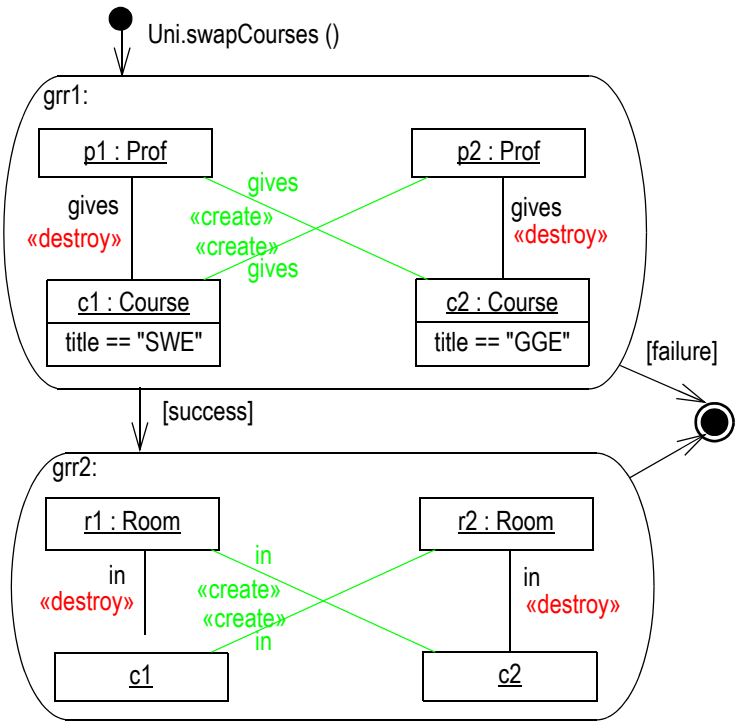
2.2.: For each object n in the story pattern we add an H_Var node with equivalent name attribute value to the left-hand and right-hand side of the story pattern.

2.3.: For each object $k \in CoreN_{grr} \cup AddN_{grr}$ we add an h_c link connecting k and the corresponding H_Var object to the right-hand side of grr.

Let $bounds \subseteq N_{LG}$ then grr:bounds is the story pattern grr with a left-hand side where the elements of bounds are implicitly linked to their corresponding H_Var object with an h_c link. We call such story pattern objects *bound variables*.

Graphically, bound variables are depicted as a box that shows the object identifier, only, but not its type, cf. objects c1 and c2 in grr2 in Example A.28. This is a short hand notation for adding h_c link between the objects and its variable in the left-hand side of the story pattern.

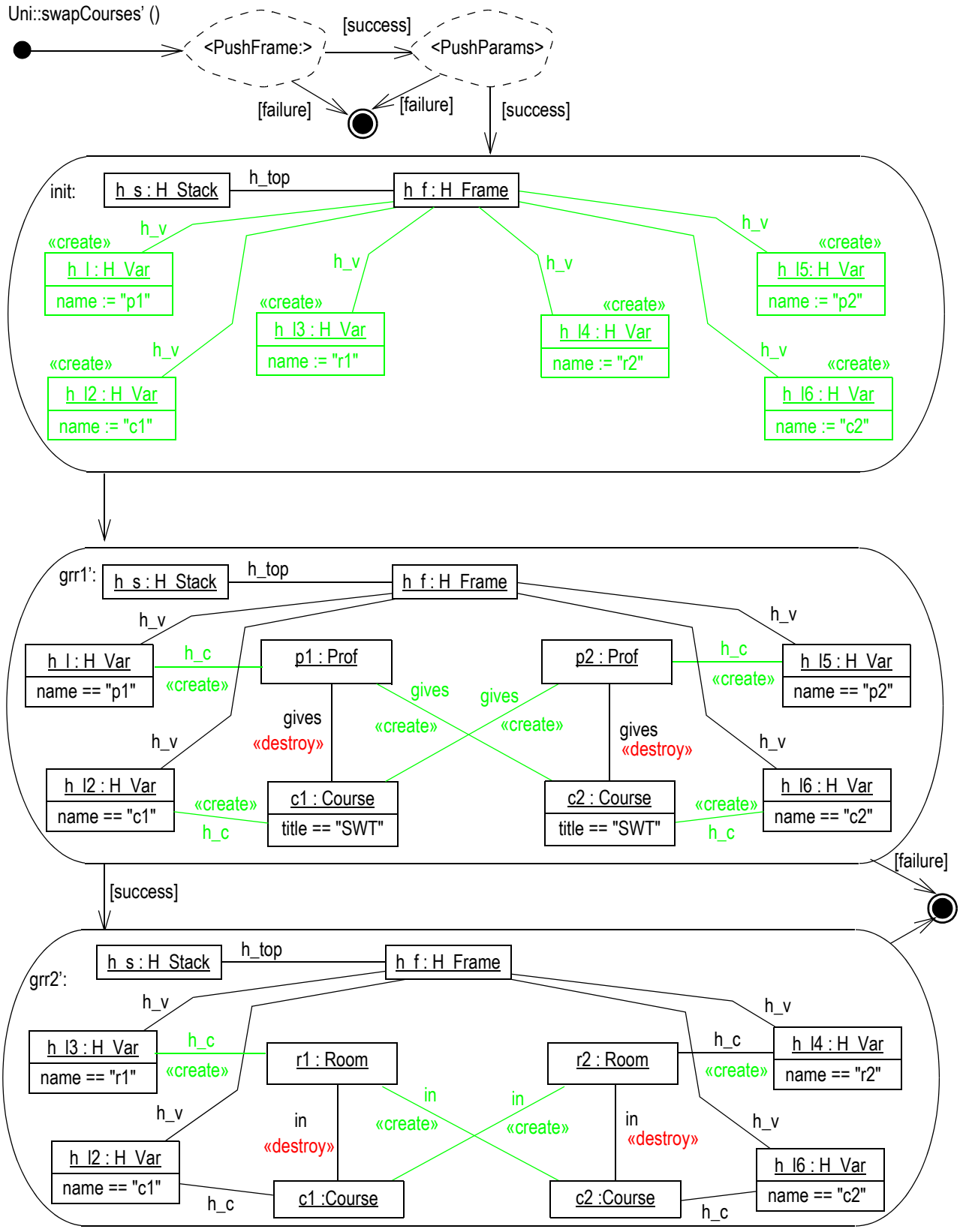
Example A.28: Bound variables in story patterns



Example A.28 shows an example story diagram that looks-up two courses with titles "SWE" and "GGE". The story diagram has the task to swap these two courses, i.e. to swap their professors and their rooms. The first story pattern grr1 swaps the professors by adapting the corresponding gives links, appropriately. The second story pattern grr2 swaps the rooms by adapting the corresponding in

links. Note how the second story pattern uses the just introduced notion of bound variables in order to look-up the courses to be modified.

Example A.29: The implicit handling of H_Var objects of Example A.28



Example A.29 shows the story diagram of Example A.28 with its implicit H_Var node handling. The additional initial story pattern init creates 6 H_Var nodes for the 6 different objects employed in the

story diagram and attaches them to the top level call frame. The first user defined story pattern grr1' is extended by the occurring H_Var nodes. As an implicit side effect, story pattern grr1' creates h_c links between the matched objects and the corresponding H_Var nodes. In the second user defined story pattern grr2' we look up the courses marked in grr1'. Note, we employ only one unique H_Stack node that has only one h_top link that identifies the top call frame, uniquely. On each call frame the names of the employed variables are unique, e.g. "c1" and "c2" describe the corresponding H_Var nodes, uniquely. Similarly, the h_c links identify the content of the variables, unambiguously. Altogether this guarantees that the second user defined story pattern grr2' is applied to the same courses as story pattern grr1'.

At the first glance the introduction of implicit variables for all objects in any story pattern of a story diagram may seem oversized. However, in practice this concept has proven to be very useful. It allows to pass parts of a handle from one story pattern to another without crowding the specification with artificial marker nodes. This simplifies the employed story patterns significantly. One may for example compare the size of the story diagrams in Example A.28 and A.29.

A.6.2 Iterated story pattern

Story patterns are inherently nondeterministic. Generally, a given graph may contain multiple subgraphs that fit to the left-hand side of the graph rewrite rule. So far, our semantics defines that each of these possible matches creates a different result graph. In our implementation, applying a graph rewrite rule to a given graph delivers one of the possible result graphs defined via the semantics relation. Which result graph is delivered is nondeterministic. Such nondeterministic choices can lead to unexpected and nonrepeatable system behavior. Thus, such situations need special care. One way to avoid such a nondeterminism is to provide a left-hand side that has only one unique match within the given host graph. This may be achieved using unique marker nodes or key valued attributes or node valued parameters passing a certain "place" for the application of the rule, cf. chapter A.5. In other situations a story pattern may match to multiple "places" within a given graph by purpose and one wants to apply the rule to all possible matches. In simple cases, one can achieve this using the while-composition. However, a while composition may easily create a termination problem if the rule can be applied to the same subgraph several times. For example, if a rule just adds some elements to a certain subgraph or increments some attribute value then the rule may probably be applied to the same subgraph arbitrary often. To avoid this, the rule has to delete some elements of its left-hand side or to modify some attributes. Such modifications mark a "place" as already visited. The next application of the rule may check these marks and it will no longer accept the same subgraph but it will choose a new subgraph. Taking care of such "visited" markers is quite tedious and thus we provide a dedicated control structure for this purpose, the iterated story pattern:

Definition A.30: Iterated story patterns

Let grr:bs = (LG, RG) be a story pattern with bound variables bs, then

iterate (grr:bs) is a story diagram with

Sem [iterate (grr:bs) do sd2 end] := Sem [initGrr;
while (newMatch (grr:bs)) do sd2 end;
resetGrr;
fail;]

where

initGrr creates a special H_Var node h_grrVisited on the current procedure call frame. Within the value attribute of h_grrVisited we will store a set of vectors where each vector identifies one match of the left-hand side of grr that has been visited.

newMatch (grr:bs) extends the execution of the story pattern grr:bs by the following two steps. First, the subgraph that is chosen for the application of grr:bs must not be contained in the set of already visited subgraphs stored in variable h_grrVisited. To store a given subgraph in h_grrVisited, one just orders the nodes in LG and creates a vector of nodes from the subgraph that are matched by the nodes in LG in this fixed order. To check whether a subgraph is already visited one produces the same vector and checks if it is already contained.

Second, if a new match is found, the corresponding node vector is added to the h_grrVisited set and the rule is executed as usual.

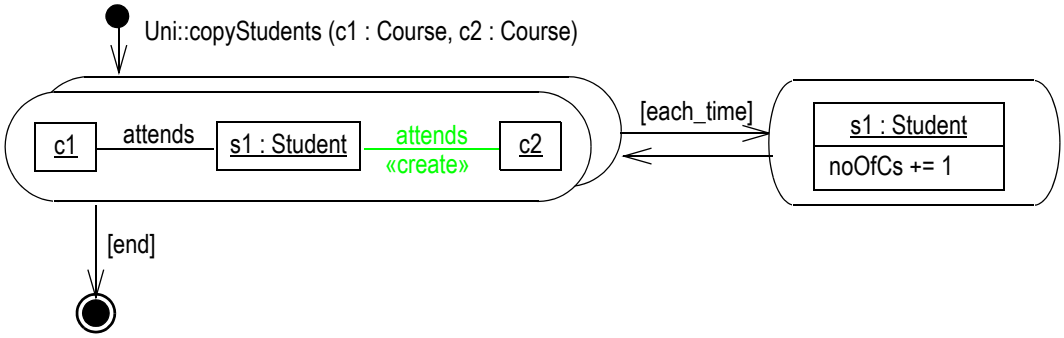
resetGrr just deletes the h_grrVisited variable created by initGrr

fail is a pseudo story pattern that is never applicable

Graphically, an iterated story pattern is depicted using two stacked activity shapes and the special transition guards [each time] and [end].

Note, we use the pseudo operation fail as last operation of an iterated story pattern. This ensures that the iterated story pattern is never applicable and thus that its behavior corresponds to the while-composition.

Example A.31: An iterated story pattern



Thus, an iterated story pattern grr employs some special, story pattern specific variable h_grrVisited. The execution of the story pattern itself is extended to accept only new subgraphs as matches that are not yet stored in h_grrVisited. In addition, for each application the new match is added to h_grrVisited. This is repeated until, no new match is found. On termination, we reset variable h_grrVisited, since the iterated story pattern may be embedded in another loop and thus it may be executed again. Note, after each execution of grr the story diagram in the body of the iterated story pattern is executed. This story pattern may again contain an iterated story pattern. Each iterated story pattern uses its specific h_xyVisited variable. Thus, the handling of already visited matches cannot interfere. We call this semantics of an iterated story pattern the *fresh-matches* semantics.

Note, the fresh-matches semantics definition does not solve the termination problem for the general case. We automatically exclude that a given rule is applied to the same subgraph several times. Since a given graph has only a finite number of subgraphs this should drain the loop, finally. However, the execution of grr or of the story diagram in the body of the iterated story pattern may create new subgraphs that fit to grr. This may lead to a termination problem.

One could solve this termination problem choosing another semantics for iterated story patterns. Instead of storing visited matches one could compute all possible matches upfront and then apply grr to one of the stored matches after the other. We call this the pre-select semantics. Using pre-select semantics one has to take care that the application of grr to one match can destroy some other matches. Thus, for each stored match one has to check if it still exists and if it still fits to the rule. However, this is possible and would solve the termination problem.

We did not choose the pre-select semantics for two reasons. First, as before the execution of a rule grr may create some new possible matches for the rule. Since these matches did not exist when all old matches were stored, these new matches are not covered by the iterated story pattern. This contradicts to the general idea of the iterated story pattern construct that means "apply grr to all matches". From this idea one may draw the expectation that after the application of the iterated story pattern a given graph G should not contain matches for grr that have not been considered. As an example, a story pattern may specify garbage collection by destroying nodes that have no parent node. After applying this graph rewrite rule in an iterated story pattern one may expect that no orphans exist, anymore. This would not hold for the pre-select semantics. It does hold for our fresh-matches semantics.

The second argument for choosing the fresh-matches semantics was an implementation and efficiency issue. For the pre-select semantics we felt it is necessary to store all possible matches up-front. This may require book keeping data proportional to the size of the given graph. For the fresh-matches semantics we found an implementation that just systematically searches through the given graph and thereby it avoids to choose the same match several times without a lot of book keeping.

To conclude there are good arguments for both, for the fresh-matches and the pre-select semantics. The pre-selects semantics avoids the termination problem, safely. This is probably bought by additional book keeping overhead. The fresh-matches semantics covers newly created matches. This may cause a termination problem. However, on termination of a story pattern the graph contains no unvisited match.

Note, how often an iterated story pattern is applied to a given graph may depend on the order in which matches are chosen. For example there may exist a single node that participates in all matches and that is destroyed by exactly one of the matches. If the destroying match is chosen as the first match, the iterated story pattern is executed only one time. This holds for both semantics. Thus, one should use this construct with care.

Note, the differences between the fresh-matches and the pre-selected semantics of iterated story patterns are very subtle. Similarly, the order in which different matches are visited should not matter. Otherwise unexpected behavior may occur, sporadically. To avoid these kinds of vague semantics one should not write iterated story patterns that create new matches or that destroy more than one match at a time. If this rule is regarded, the semantics becomes clear and simple. Due to our experience, it is easy to respect this rule in practice and iterated story pattern have proven to be very handy in various situations.

Example A.31 shows a simple example for a safe use of an iterated story pattern. The iterated story pattern searches for all students attending course c1. For each such student an attends link to course c2 is created. In addition, the body activity reached via the [each time] transition invokes the notify method on each matched student. Note, a simple while-composition of the same operations would have created a non-terminating loop. Each new application of the graph rewrite rule could just match the same student x and add it to course c2 again and again. Without the iterated story pattern construct we would have to check for each student if he or she already attends c2. The next sections will show how such a check could be specified, easily. However, this could create an efficiency problem since the while-composition would start the search for matches anew for each iteration. This may check (and reject) the same subgraphs within each iteration causing O (n^2) matches to be considered where n is the number of possible matches in the given graph. Our implementation of the iterated story pattern just performs one systematic search through the graph considering only O(n) matches.

A.6.3 Boolean constraints

Story patterns are especially suited to look-up complex object patterns. However, frequently one wants to execute a story pattern only if the handle fulfills some additional constraints. Like in UML, we notate such additional constraints as boolean expressions in curly braces. Later on, we will allow extended OCL expressions as boolean constraints, so far we restrict ourselves to usual boolean expressions that are built using the standard operators provided by our unique stack object. However, such expressions may invoke user defined methods, e.g. with boolean return values.

For the semantics definition, we extend the iterated story pattern with a condition part that is applied to each match before that match is accepted:

Definition A.32: Story patterns with boolean constraints

Let grr:bs = (LG, RG) be story pattern with bound variables bs.

A story pattern may be extended by a set BCs of boolean expressions. Textually, we write grr:bs|BCs. Graphically, the boolean constraints are shown in curly braces within the story pattern.

Sem [try (grr:bs|BCs)] :=

```
Sem [   boolean h_grrSearching = true;
        iterate test_grr:bs do
                if h_grrSearching and BC1 and ... and BCn
                then
                        try ( grr:NLG ); h_grrSearching = false;
                end
        end ]
```

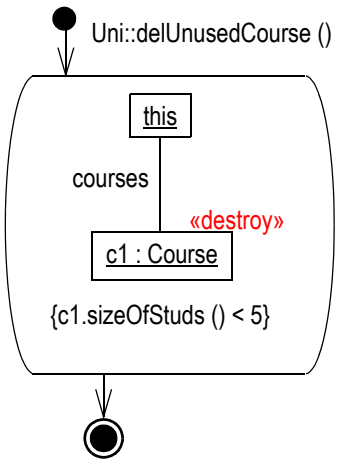
where

test_grr:bs := (LG, LG):bs. Note, rule (LG, LG):bs has identical left-hand and right-hand sides and thus it does not modify the given host graph. However, if rule (LG, LG):bs is applicable the corresponding variable binding is performed as described in Definition A.27.

BC_i := the story diagram that evaluates the boolean constraint $bc_i \in BCs$ for all constraints in BCs

and := the usual boolean and operator provided by our standard stack object. Note, this operator is evaluated left to right using short circuit evaluation.

Example A.33: Using boolean constraints



A story pattern with boolean constraints is executed by searching for a subgraph that fits for the basic story pattern and that additionally allows to evaluate all boolean constraints to true. However, the boolean constraints may call operations that in turn perform complex computations.

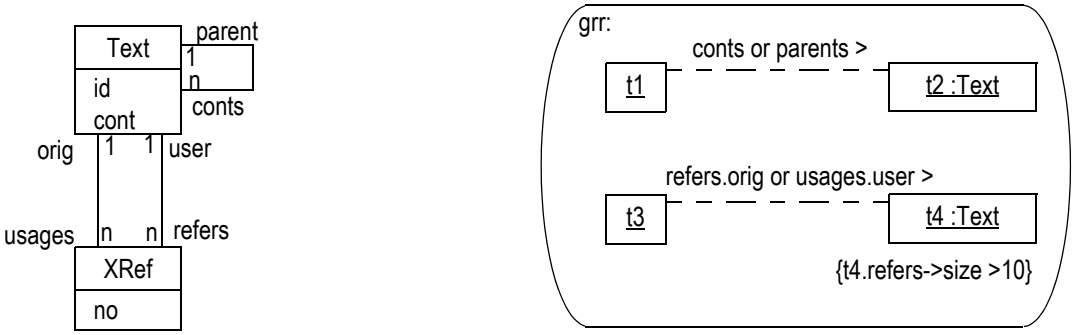
Thus, for a story pattern containing constraints we first construct a look-up story pattern test_grr. This look-up story pattern has equal left-hand and right-hand sides and thus it does not modify the current graph. However, test_grr is still considered as a user provided story pattern and thus the implicit extension handling the H_Var objects is applied to it, too. Thus test_grr has the effect of choosing a handle and storing the chosen nodes in the corresponding H_Var objects. Subsequent operations within the boolean constraints may look-up objects of the left-hand side of the story pattern grr as bound objects.

In Example A.33 we use the constraint {c1.sizeOfStuds () < 5} to restrict the application of method Uni::delUnusedCourse to course nodes c1 where the corresponding method sizeOfStuds returns a number less than 5, i.e. to courses with less than five participants.

A.6.4 Constraints and navigations using the OCL

In UML, boolean constraints may use OCL expressions in addition to plain boolean operations. OCL provides operators that may be applied to sets or bags of objects and that compute sets or bags of neighbors or attribute values. In story patterns one may use OCL expressions as parts of boolean constraints or as so-called navigational expressions, i.e. as a kind of complex path via a series of links from one node to another:

Example A.34: Using OCL for navigational expressions



Example A.34 shows a small class diagram modeling hierarchical text fragments with cross references. Story pattern grr employs two navigational expressions. Navigational expressions are depicted as dashed lines connecting two nodes of the left-hand side of a graph rewrite rule. This dashed line is labeled with an (extended) OCL expression, e.g. conts or parents and with a direction marker e.g. >.

Definition A.35: Semantics of OCL navigational expressions

Let grr = (LG, RG) be a story pattern with x, y $\in N_{LG}$ and
let from x to y via <oclExpr> be a navigational expression, then

Sem [grr | from x to y via <oclExpr>] := Sem [grr | { y Œ x.<oclExpr> }]

Graphically, the OCL navigational expressions is shown as a dashed line from x to y labeled with the OCL expression <oclExpr> and with an arrow head depicting the reading direction of the OCL expression.

Basically, OCL navigational expressions correspond to boolean constraints related to the corresponding source and target variable. However, our implementation will use OCL navigational expressions

to compute candidates for the target from the source node. In Example A.34 node t2 may be computed by applying the OCL expression conts or parents to node t1.

Within OCL expressions we allow all kinds of operators provided by OCL. The semantics of these operators is satisfyingly defined in [UML97]. Thus, we just assume, that our unique H_Stack object provides appropriate methods for these operators. However, we repeat the most important OCL operators in order to guarantee a common understanding and to provide sufficient knowledge for the understanding of our examples for those that are not familiar with OCL. In addition, we have extended OCL by three new operators.

Definition A.36: Informal introduction of some OCL operators

- '.' becomes applicable to sets and bags of objects:
  - t1.refers computes the set m1 of XRef objects attached to t1
  - t1.refers.orig starting from m1 the bag b1 of reached orig texts is computed
  - t1.refers.no starting from m1 the bag of corresponding no attributes is computed

  Note, OCL operations frequently return a bag or a sequence instead of a plain set. A bag or a sequence may contain the same element multiple times. Assume for example that t1.refers delivers two XRef object r1 and r2 that both hold the same value 8 within their no attribute. Using bags the expression t1.refers.no computes Bag{8, 8} containing value 8 two times. Applying e.g. the sum operator to this expression delivers t1.refers.no->sum = 16. In case t1.refers.no would have computed a set, the attribute value 8 would have been contained one time, only, and the sum would have computed 8 instead of 16.

  Applying the '.<assoc>' operator to a single element will create the collection that corresponds to the used association assoc, i.e a sequence in case of an ordered or sorted association and a set, otherwise. Applying the '.<assoc>' operator to a collection usually creates a bag, in order to deal with duplicated elements, properly.
- '->' allows access to collection methods:
  - t1.refers->size number of elements in t1.refers
  - t1.refers->isEmpty boolean operator if t1.refers is empty
  - t1.refers.no->sum sum of all no attributes
  - collection->includes (object)
    true if object is an element of collection, false otherwise
  - collection->notEmpty is collection not the empty collection?
- collection converting operators
  - collection->asSet turns collection into a set (without duplicate values)
  - collection->sortedBy (self.<attrExpr>)
    turns collection into a sequence. self.<attrExpr> returns a key for each object which has to provide a less-than operation which is used as sorting criteria.

  Note, the operator ->asSet might be used to turn bags into usual sets that contain each element at most one time.

- Iterative operations:
  - collection->sum
    The addition of all elements in collection. Elements must be of a type supporting addition (Integer and Real)
  - collection->count(object)
    the number of occurrences of object within collection
  - collection->exists (<oclexpr>)
    Results in true if <oclexpr> evaluates to true for at least one element in collection.
  - collection->forAll (<oclexpr>)
    Results in true if <oclexpr> evaluates to true for each element in collection. Otherwise it results in false.
  - collection->select(<oclexpr>)
    The subcollection of collection for which <oclexpr> is true
  - collection->iterate(x; acc = <init-expr> | <expr-including-x-and-acc>)
    Corresponds to following Java-like pseudocode:

```
iterate(x : T; acc : T2 = value)
{
        acc = value;
        for (   Enumeration e = collection.elements() ;
                e.hasMoreElements(); )
        {
                x = e.nextElement();
                acc  = <expression-with-x-and-acc>
        }
}
```

  Note, most other -> operators are defined in terms of the ->iterate operator, cf. [UML97]
- Some operators that are more handy than the original OCL operators
  - collection <= collection2
    true if collection is a subset of collection2, false otherwise
    Replaces collection2->includesAll (collection)
  - collection or collection2
    the union of collection and collection2
    replaces collection -> union (collection2)
  - collection and collection2
    the intersection of collection and collection2
  - collection - collection2
    the difference of collection minus collection2
- transitive closure
  - collection.(<oclexpr>) * := collection->asSet
    if collection.(<oclexpr>)->asSet <= collection
    or (collection or collection.<oclexpr>).(<oclexpr>) *
    otherwise
    the *set* of objects reachable from collection by applying <oclexpr>

zero or more times:
collection or collection.<oclexpr>

- collection (<oclexpr>) +
  collection.<oclexpr>.(<oclexpr>) *

  Note, the transitive closure operator * computes a *set* of objects as a result, not a bag. This allows the recursive definition given above. The idea is that the iterated expression <oclexpr> is applied as long as new elements are reached. All intermediate results are collected. Using bags instead of sets, elements that are visited again via a loop in the graph structure would accumulate and the transitive closure would not terminate. In contrast, a set stops to grow at latest, when it contains all elements of the current graph.

  Note, these transitive closure operators are a real extension of OCL. OCL does not provide this kind of operation.

- user defined methods
  - collection.m1 (...) corresponds to:
    collection->iterate (x, acc = Bag{} | acc := acc or x.m1 (...) }

    Note, user defined methods are just applied to each element of the collection and the results are collected and finally returned. Method m1 may return a single element or already a collection. Note, if collection contains an element x multiple times, than m1 is called multiple times on x. If an element y is returned several times than the result will contain y, multiple times, too.

    Note, the entry type of a collection is not always clear. Assume a set studs of students and a set profs of professors. Which is the entry type of studs or profs? And is method m1 provided by all elements of the resulting collection? One can solve this problem requiring that the class hierarchy forms a mathematical lattice. In that case it is always possible to find the closest common ancestor of two classes and to use this as the entry type for a set constructed by the union operator. However, these kinds of problems belong to static typing and compile time checkings. For the execution semantics we just define that for each object it is checked wether it provides method m1 and if not we return the empty bag.

Note, all OCL operators may be applied to single elements, too. In that case the single element is just converted into a set containing the single element.

Note, the introduced OCL operators may not only be used within OCL navigational expressions, but also within usual boolean constraints and attribute expressions.

A.6.5 Negative graph elements

Frequently, one wants to exclude the application of a story pattern if certain elements exists in the neighborhood of the rule handle. Therefore we introduce so-called negative nodes and edges.

Definition A.37: Negative nodes

Let grr : bs | BCs = (LG, RG) be a story pattern with bound variables bs and boolean conditions BCs.

Some nodes and edges in LG may be contained in a special set $NCs \subseteq N_{LG} \cup E_{LG}$, a set of negative nodes and edges, attached to grr which is textually written as
grr : bs | BCs ! NCs.

Note, negative edges *must not* be attached to negative nodes!

Sem [try (grr : bs | BCs ! NCs)] := Sem [try (grr_0 : bs | (BCs and BNCs))]

where

grr_0 := (LG_0, RG_0) where LG_0 := LG - NCs and RG_0 := RG -NCs

BNCs := { grr_x | x ∈ NCs }

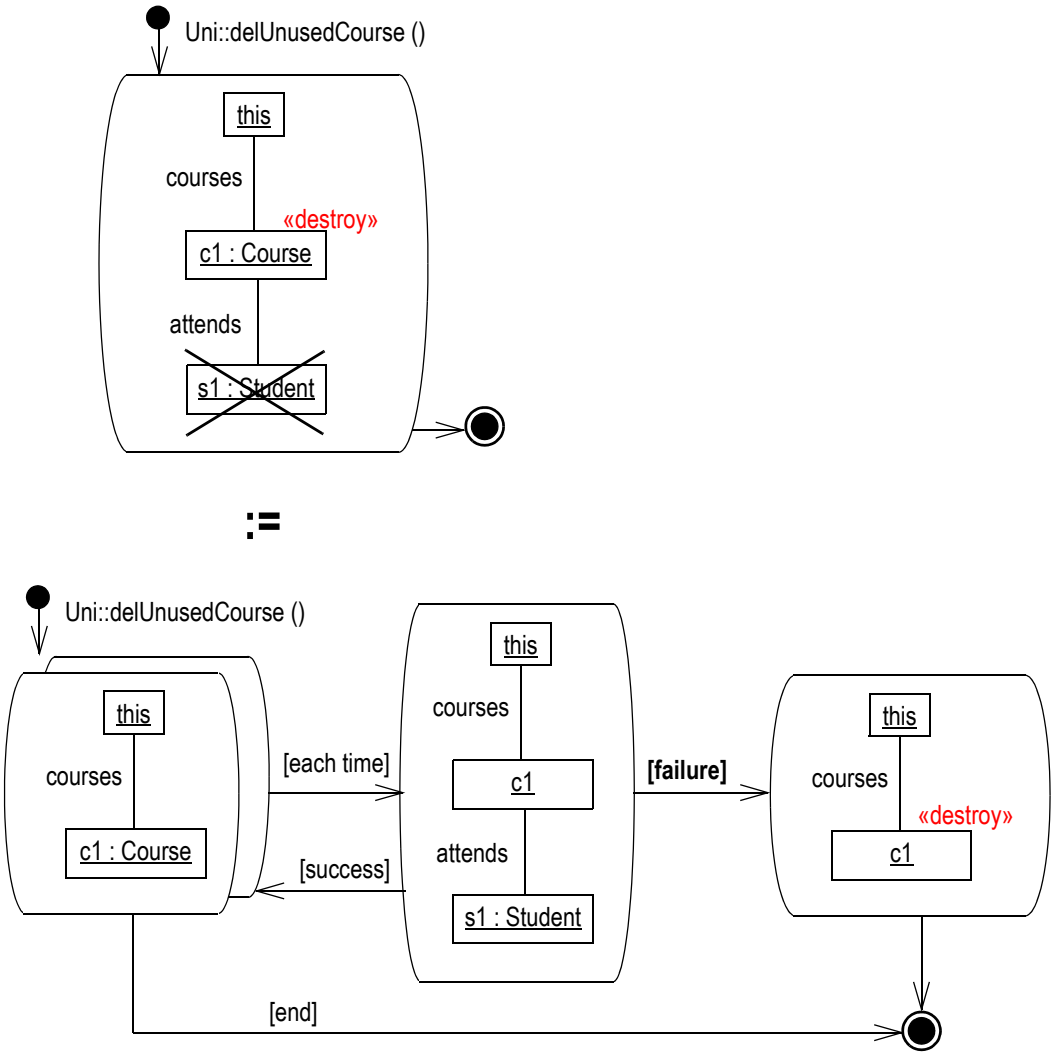
where

grr_x := ! try ((LG - (NCs - x), LG - (NCs - x)) : N_{LG0})

Note, subtracting a node from a graph means subtracting it from the set of nodes and deleting all attached edges and restricting the node labeling and attribute value function. Subtracting an edge just removes the edge.

Graphically, elements contained in NCs are crossed out with a bold X:

Example A.38: Using negative pattern elements



Thus, negative story pattern element are just turned into special boolean constraint operations attached to a reduced variant grr_0 of the original story pattern grr. The reduced variant grr_0 is derived from grr by just removing all negative elements. In addition, the negative elements are turned into special boolean constraints, one negative constraint operation for each negative element. For each such negative constraint operation, we reduce the story pattern grr by removing all negative conditions but the one we want to test. In addition, we employ bound variables for the "positive" nodes in order to test exactly the handle that has been identified in the first step. Graph modifying effects of such a negative constraint operation are removed by using identical left-hand and right-hand sides. Finally, the not operator '!' is applied to the negative constraint operation. Thus, the test for a negative element actually tests whether the current handle can be extended to match the negative element, too. This is

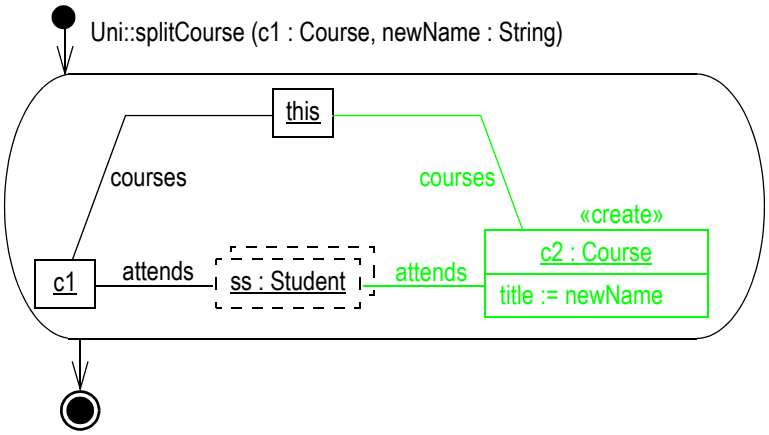
forbidden and the not causes the constraint operation to fail. If we found a subgraph that has no extension for the negative elements then the negative constraint operation evaluates to true. As defined for boolean constraints in general, once all usual boolean constraints and all negative constraints are passed, rule grr_0 is executed (i.e. rule grr reduced to the positive nodes). Otherwise, the iterated story pattern looks up the next possible handle for grr_0. This is repeated until the first handle is found that passes all negative conditions.

In Example A.38 there is only one negative element, namely node s1. Thus, the depicted story pattern searches for a course c1 that has no student s1 attending it. The matching course is then deleted.

A.6.6 Multi object variables

Iterated story patterns allow to apply a single graph rewrite rule to a set of subgraphs within a given host graph. Another frequently occurring situation is that one wants to copy or redirect a set of edges from one node to another node. To facilitate this frequently occurring task, story patterns provide so-called *multi object variables*. Graphically, multi object variables are shown using two stacked dashed boxes. Example A.39 shows a story pattern with a multi object node ss. The example story pattern adds a new course to the this object. The multi object node has the task to match the set of all students attending course c1 and to create attends edges between the new course c2 and each of these students.

Example A.39: Using multi object variables



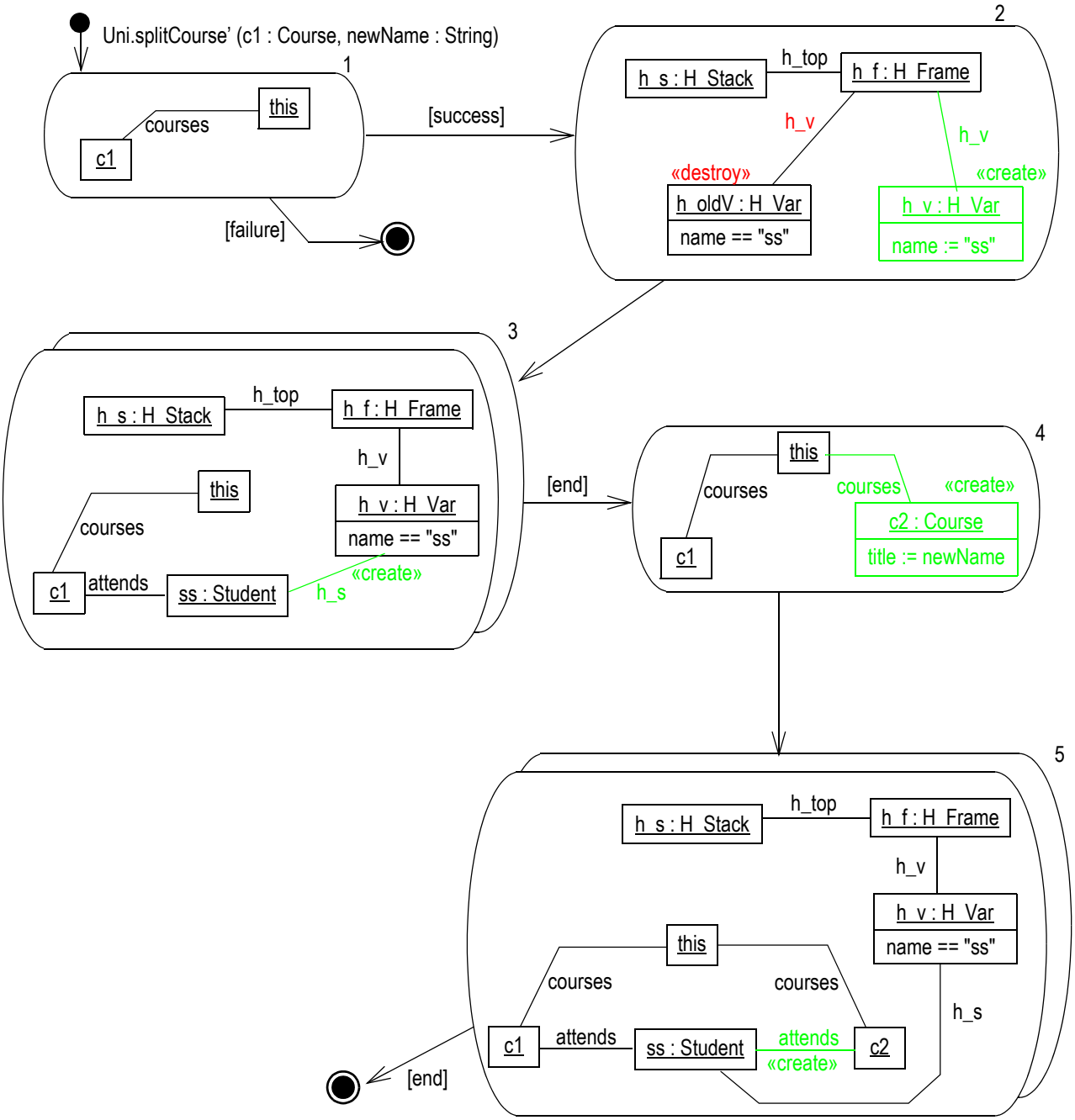
Multi Object nodes allow to create edges for a set of nodes or to delete edges attached to a set of nodes or to assign a new value to attributes of a set of nodes. One may also delete the whole set of nodes. Later on, we will define how to invoke a method on a set of nodes which means to call the method on each node, one after the other.

The semantics definition for multi object variables is a little bit complicated. Basically, a story pattern with a multi object variable is executed in 5 steps, cf. Example A.40. In step one, we choose a handle for the core rule, i.e. the rule without the multi object node. In our example, this is trivial, since nodes this and c1 are already bound variables. Step two just reinitializes variable ss. In step three, we mark all nodes matching the multi object variable ss. This is done using an iterated story pattern, that creates h_s edges between nodes fitting to the multi object variable and the corresponding H_Var node. In step four, we execute the core rule at the chosen handle. The latter is achieved using bound variables for all nodes in the left-hand side. In step five, the effects for the multi object variable are executed. Again, we use an iterated story pattern to look-up all nodes marked by h_s edges and we employ a graph rewrite rule that executes only the modifications of the multi object variable.

In case of multiple multi object variables, step 2, 3, and step 5 are executed for each of the multi object variables, separately. Note, a given host graph node must not be matched by different multi object variables, because the modifications performed by the different multi object variables could contra-

dict to each other. For example one multi object node could delete the contained nodes while another attaches new edges to the contained nodes. To avoid such conflicts, our semantics definition will add nodes that would fit for several multi object variables to the "first" multi object variable, only. This creates the problem that the order in which the multi object variables are considered has semantical relevance. To solve this problem, the user may define the order in which different multi object variables are considered. Graphically, this is done using index numbers subscribed to the dashed boxes. In chapter A.6.8 we will introduce alternative language means that allow a single host graph node to be matched by different nodes of the story pattern, if the performed modifications do not conflict.

Example A.40: Executing story patterns with a multi object variable



The formal definition also deals with the problem that a given story pattern may contain both, multi object variables and negative elements:

Definition A.41: Multi object variables

Let grr : bs | BCs ! NCs be the graph rewrite rule (LG, RG) with bound variables bs and boolean constraints BCs and negative elements NCs and
let Sets $\subseteq$ (N_{LG} - NCs) be a list of multi object nodes,

then

Sem [try (grr : bs | BCs ! NCs * Sets)] :=

Sem [try (grr_0 : bs | (BCs and SetCs and SetBNCs));
SetXs]

where

grr_0 := (LG_0, RG_0) := (LG - Sets - NCs, RG - Sets - NCs)

SetCs := fill_grr_x1 and ... and fill_grr_xn for xi $\in$ Sets
where
fill_grr_xi := reset_xi; iterate fill_one_grr_xi : NLG_0 do nop end; nop

reset_xi := a simple story pattern that kills the H_Var object corresponding to xi and that creates this H_Var object, anew. The purpose is to kill old h_s edges.

fill_one_grr_xi := (LG_fill_one_grr_xi, RG_fill_one_grr_xi) : N_{LG_0} | noSetMatch (xi)

LG_fill_one_grr_xi := LG - NCs - (Sets - xi)

RG_fill_one_grr_xi := LG_fill_one_grr_xi $\cup$ (h_xi, h_s, ε, ε, xi)
where h_xi is the H_Var node corresponding to xi

noSetMatch (n) :=n $\notin$ { s | s is contained in the match of some multi object variable xi $\in$ Sets }
if n is a node
true otherwise

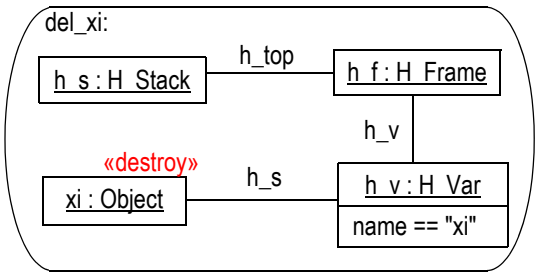
SetBNCs := { grr_y | y $\in$ NCs }
where
grr_y := ! try ((LG - Sets - (NCs - y), LG - Sets - (NCs - y)) : N_{LG0} | NoSetMatch (y))

nop := ($\varnothing$, $\varnothing$) the empty rule, changes nothing and is always applicable

SetXs := try (exec_grr_x1); ... ; try (exec_grr_xn) for xi $\in$ Sets
where

exec_grr_xi := iterate one_grr_xi do nop end

one_grr_xi :=



if xi $\in$ delN (grr)

(LG_one_grr_xi, RG_xi) : N_{RG_0} otherwise

LG_one_grr_xi := (RG_xi - one_xi_addEs) $\cup$ one_xi_delEs
and $av_{LG\_one\_grr\_xi}$:= $\varnothing$

one_xi_addEs := { (src , el, i, q, tgt) $\in$ $AddE_{grr}$ | src = xi or tgt = xi }

$AddE_{grr} := E_{RG} - E_{LG}$

$one\_xi\_delEs := \{ (src , el, i, q, tgt) \in DelE_{grr} \mid src = xi \text{ or } tgt = xi \}$
$\cap\ (N_{RG\_xi} \times EL \times R \times AttrValues \times N_{RG\_xi})$

$DelE_{grr} := E_{LG} - E_{RG}$

$RG\_xi := RG - NCs - (Sets - xi)$

Note, the story pattern must not contain edges connecting two multi object variables or a multi object variable and a negative node.

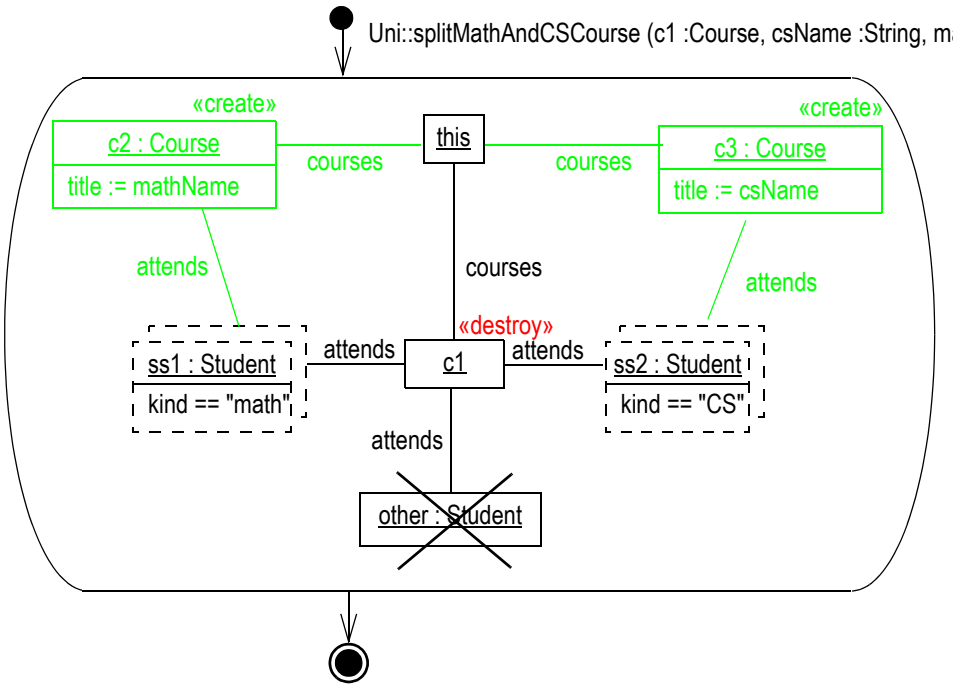
The semantics definition for multi object variables follows the idea out-lined in Example A.40. For step one we employ story pattern grr_0 : bs. Story pattern grr_0 : bs is derived from the original story pattern grr : bs by just removing all multi object variables and all negative nodes and edges. Story pattern grr_0 is equipped with some special boolean constraints. First the original boolean constraints BCs are tested. Second, we employ set conditions SetCs derived from the multi object variables. Third, we use extended negative constraints SetAndBNCs.

The set conditions SetCs correspond to step 2 of Example A.40. For each multi object variable xi, we generate a special operation fill_grr_xi. Operation fill_grr_xi first employs story pattern reset_xi that removes any old h_s edges from the corresponding H_Var object. The rest of operation fill_grr_xi corresponds to an iterated story pattern that matches all possible candidates for multi object variable xi and marks them via h_s edges. This is done using story pattern fill_one_grr_xi. Story pattern fill_one_grr_xi is derived from the original rule grr in two steps. The left-hand side LG_fill_one_grr_xi corresponds to the left-hand side of the original rule minus all negative elements and minus all multi object variables but with multi object variable xi remaining. Within fill_one_grr_xi the multi object variable xi is handled as a normal node. In rule fill_one_grr_xi all objects but xi are bound variables. Thus, the resulting rule matches the same handle as rule grr_0 and tries to find new candidates for node xi, only. Note, the candidates for xi shall not yet have been matched by any (other) multi object variable. This is enforced by the additional boolean constraint noSetMatch (xi) that just says, xi shall not (yet) be part of the match of any multi object variable. The right-hand side of fill_one_grr_xi is just a copy of the left-hand side with one additional edge (h_xi, h_s, ε, ε, xi). Thus each execution of rule fill_one_grr_xi creates one h_s edge between a new candidate for xi and the corresponding H_Var object. Rule fill_one_grr_xi is employed as an iterated story pattern, thus all possible candidates for xi are marked. The final nop operation of the whole fill_grr_xi construct just ensures, that fill_grr_xi is always applicable. Thus, the boolean condition fill_grr_xi never fails.

Note, the special conditions fill_grr_xi are employed as boolean constraints within the execution of rule grr_0. As boolean constraints they should not have side-effects. We violated this rule, in order to simplify the semantics definition for multi-object variables. This allows us to rely on the semantics definition for boolean constraints, otherwise we would have had to repeat that definition, here.

Once all candidates for all multi object variables are marked with h_s edges we are ready to consider the negative elements. This needs special care. There is a semantical difference whether we match multi-object variables, first, and try to extend such a match for a negative element afterwards or if we work the other way round:

Example A.42: Using multi object variables together with negative nodes



Example A.42 shows a variant of Example A.39. In Example A.42 we split course c1 into two new courses c2 and c3. All students ss1 that study mathematics (kind == "math") shall attend c2 and all computer science students shall attend c3. The negative node other ensures that no student is lost. This means, the operation must not be executed if there is some student that is neither covered by ss1 nor by ss2. This corresponds to the semantics defined for multi-object variables, since we look-up the matches for multi object variables, first, and consider the negative elements, second. If we would have considered negative elements, first, then the above rule would be applicable for courses that have *no* students, only.

The special negative boolean constraints SetBNCs take care of the situation discussed above: operations that try to extend the current match to a match for a negative element must not use an object matched by a multi-object variable for this purpose. SetBNCs are constructed like usual negative constraints. In addition, multi-object variables are removed from SetBNCs operations. Instead, the special boolean constraints noSetMatch (y) are added to the SetBNCs operation that tries to extend the current match towards a match for a negative node y. The noSetMatch (y) constraints ensures that the match for y is not taken from one of the multi object variables.

If a match for the core rule grr_0 is found that fulfills all constraints the core rule is executed as defined in Definition A.32. Thus steps 1 to 3 of the handling of multi-object variables are done, cf. Example A.40. What remains is step 4, i.e. the execution of the effects attached to multi object variables. This is defined by the SetXs operations. SetXs is a sequence of exec_grr_xi operations, one exec_grr_xi for each multi object variable.

The exec_grr_xi operations are iterated story patterns employing one_grr_xi rules. A one_grr_xi rule handles one node marked by an h_s edge. The one_grr_xi rules distinguish two cases. First, the multi-object variable may be destroyed. In this case we employ a simple rule del_xi that just looks-up an object marked by an h_s edge and destroys it. Since this rule is employed as an iterated story pattern all nodes matched by the multi object variable are destroyed.

If the multi object variable is not destroyed it might modify attributes and destroy edges and create edges. Note, the core rule grr_0 is already executed. Thus, nodes (and edges) of the left-hand side of grr_0 may already have been deleted. Thus the rule modeling the effects of an multi object variable is

derived from the right-hand side of grr. The right-hand side RG_xi of rule one_grr_xi is constructed from RG by removing all negative elements and all multi object variables but xi. The left-hand side LG_one_xi of rule one_grr_xi starts with RG_xi, too. Then, we remove those edges that need to be created and we add those edges that need to be deleted. In addition, we remove all attribute conditions from the left-hand side, since those conditions are already checked and we need the attribute assignments modelled in the right-hand side by av_{RG_xi}, only.

The set of edges one_xi_addEs that need to be created by one_grr_xi are those that are created by grr and that are attached to xi.

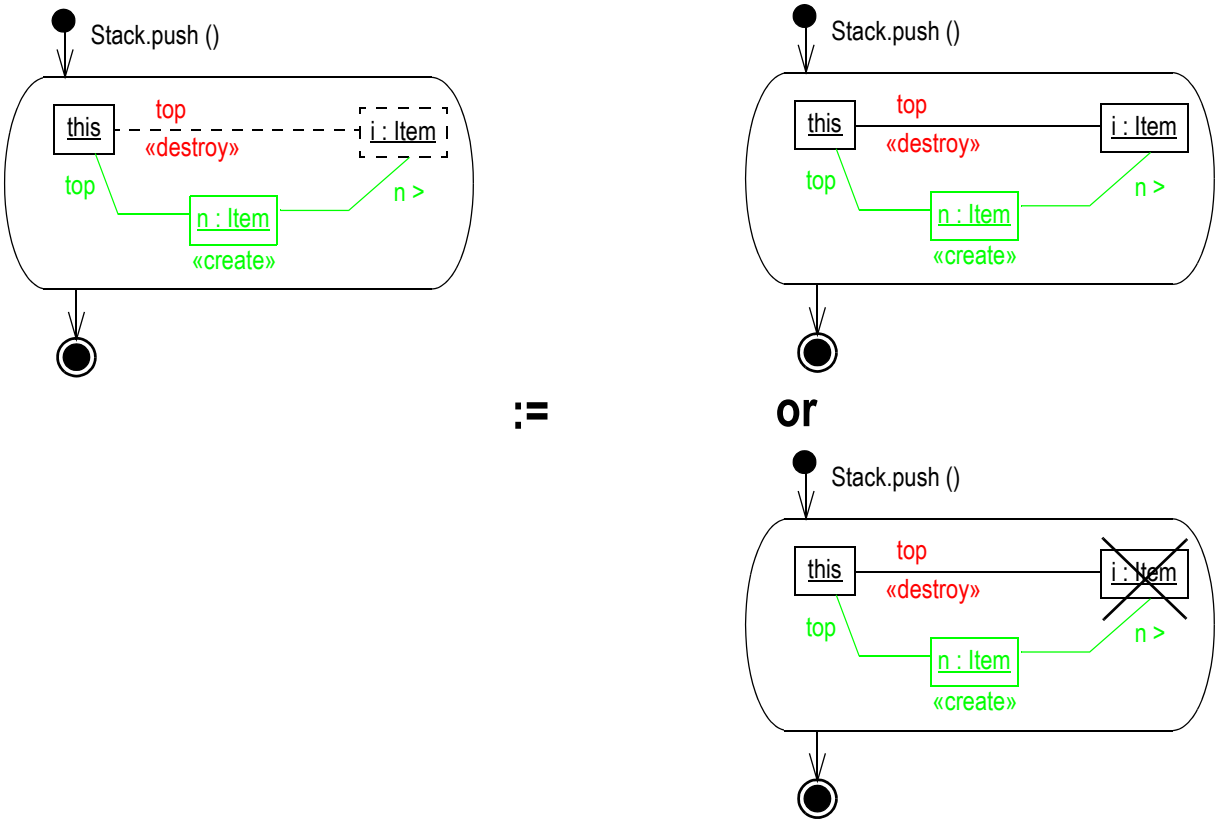
The set of edges one_xi_delEs that need to be deleted by one_grr_xi, explicitly, are those that are deleted by grr and that are attached to xi and where the other node is not deleted by grr. The latter condition is ensured by intersecting the set of possible one_xi_delEs with the set of all edges possible between nodes of RG_xi.

The semantics definition for multi object variables is quite complex. In addition, there are many pitfalls in the use of multi object variables e.g. if an object could be matched by different multi object variables or by an multi object variable and a negative node. However, the most common use-case for multi object variables is the redirection of bundles of edges or the deletion of a set of neighbor nodes. Commonly, one uses only one multi object variable per rule. Using multi object variables and negative nodes together that may have common matches is quite seldom. For the remaining simple use-cases the semantics of multi object variables is quite simple and due to our experiences multi object variables have proven to be very handy and facilitate to deal with their use-cases, significantly.

A.6.7 Optional graph elements

Optional graph elements are like multi object nodes that may contain at most one element. Optional graph elements are frequently used when there may be one object but it may not exist, either. Example A.43 shows a typical example, the push operation for a stack. Optional graph el elements are shown using dashed lines and borders. In our example node i is optional. The push operation has to consider two cases. First, the stack is empty and one just creates the new item n and attaches it via a top link to the stack object this. Second, the stack contains already an item i. In this case, the old top link has to be deleted and an n link from the new item to the old item has to be created in order to setup the stack structure. Optional elements allow to deal with such situations in a single rule, easily. Thus, story pattern Stack.push works for empty stacks as well as for stacks containing elements. If an element exists, variable i is bound to that value and the rule is executed as if i would be a normal element. If no element exists, we subtract the node i from the rule and only the remaining rule is executed.

Example A.43: Stack.push for an empty and a non-empty stack



Definition A.44: Optional story pattern elements

Let grr : bs | BCs ! NCs * Sets be the story pattern (LG, RG) with bound variables bs and boolean constraints BCs and negative constraints NCs and multi object variables Sets and let Opts $\subseteq$ $N_{LG} \cup E_{LG}$,

then

try (grr : bs | BCs ! NCs * Sets ? Opts) is a story diagram with

Sem [try (grr : bs | BCs ! NCs * Sets ? Opts)] :=

$$\bigcup_{x \in Opts} \quad (Sem [try (grr : bs | BCs ! NCs * Sets ? (Opts - x))] \cup Sem [try (grr - x : bs | (not\_x and BCs) ! NCs * Sets ? (Opts-x))])$$

where

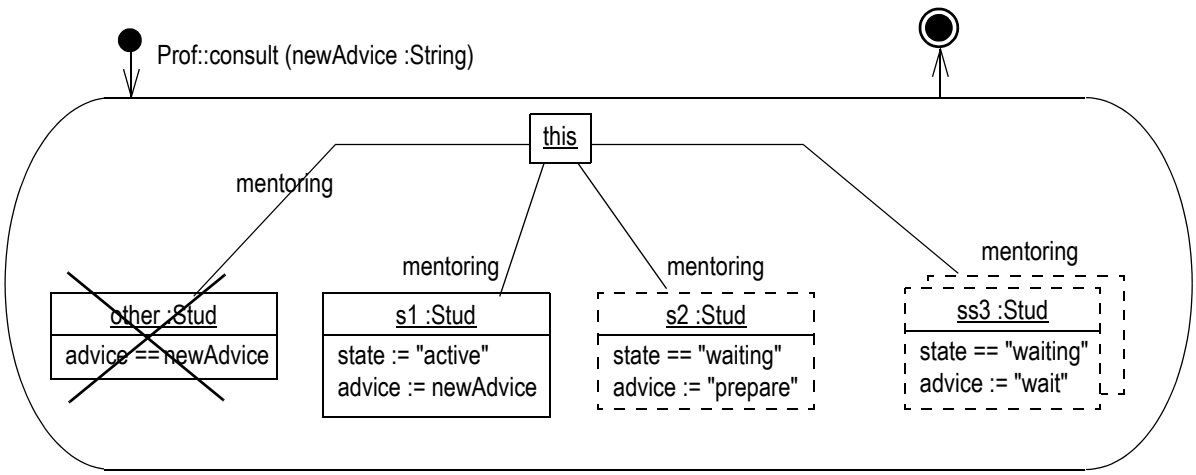
not_x := ! try ((LG - NCs - Sets - (Opts -x), LG - NCs - Sets - (Opts -x)) : N_{LG})

We call x $\in$ Opts an optional story pattern element. Graphically, optional story pattern elements are shown using dashed shapes or lines. cf. node i in Example A.43.

Edges attached to optional nodes are drawn as dashed lines, too. However, such edges must not be part of the set of optional rule elements. Edges explicitly contained in Opts may connect usual nodes, only.

This formal definition splits a rule containing an optional element x into two rules. One rule employs the optional element just as a normal element, i.e. only Opts - x optional elements remain. The second rule handles the case that it is not possible to match the optional element x. In this case we apply rule grr - x, i.e. we remove element x from the left-hand and right-hand side of grr. However, rule grr - x may be applied only where the full rule grr is not applicable. This could be achieved by turning element x into an negative element, i.e. using NCs $\cup$ x as set of negative elements. But, this simple approach would interfere with the handling of multi-object nodes, cf. Example A.45.

Example A.45: Combining optional, negative and multi object nodes in one rule



Rule Prof::consult assigns some newAdvice to the corresponding attribute of student s1. In addition, the rule advices some other "waiting" student s2 to prepare himself or herself for an advice, all other "waiting" students get the advice to wait further. In addition, rule Prof::consult ensures that each student gets his or her individual advice by the negative node other that forbids the existence of another student which already got the advice newAdvice.

Note, to achieve the described behavior, optional nodes need to be considered before multi object variables and before negative nodes. If we would try to find matches for the multi object variable ss3 first, then ss3 would cover all students in state waiting and no match for the optional element s2 would be left. Since optional elements are not mandatory for the rule execution, the rule would still be applicable but no student s2 would get the prepare advice. As discussed in chapter A.6.6, negative elements are considered after multi object variables and consequently they are considered after optional elements, too.

To summarize, due to our experience, handling optional elements before multi object variables (and before negative elements) is the most meaningful semantics. However, this creates a problem for our semantics definition. For the rule that omits the optional element x we want to guarantee that it is not applied where the match could be extended to cover x. We are not allowed to use the negative conditions NCs $\cup$ x in order to achieve this, because such a usual negative element is handled after other optional elements and after multi object variables. Such an other element could match the candidate for x and the negative condition would fail to detect that there would have been such a candidate. To avoid this, we employ the special boolean constraint not_x. The boolean constraint not_x is constructed like the constraint for a negative element, cf. Definition A.37. However, the special constraint not_x is inserted in front of the other constraints as part of the usual boolean constraints BCs, see Definition A.44. The definition for multi object variables handles such usual boolean constraints before it computes the matches for the multi object variable, cf. Definition A.41. This achieves the desired behavior for optional elements.

Note, our definition covers one optional element x at a time. We use our definition recursively by calling it with Opts - x optional elements. Since one element is removed, the recursive definition will terminate, finally.

Note, in some cases the choice of the optional element x to be considered next may have semantical relevance. If there exists a node in the given host graph that is the only candidate for two different optional elements opt1 and opt2 and if our recursive definition chooses opt1 first, then opt2 will not find a match, because we require injective matches, i.e. all elements of the rule are matched to different elements of the graph, cf. Definition A.9. Since we are not able to match object opt2 our definition

removes it from the rule. Thus, all effects, like e.g. an attribute modification, attached to opt2 are neglected. If our recursive definition chooses opt2 first, opt1 will not find a match anymore and the effects attached to opt1 will be neglected. Thus the sequence in which the optional elements are considered may cause different resulting graphs.

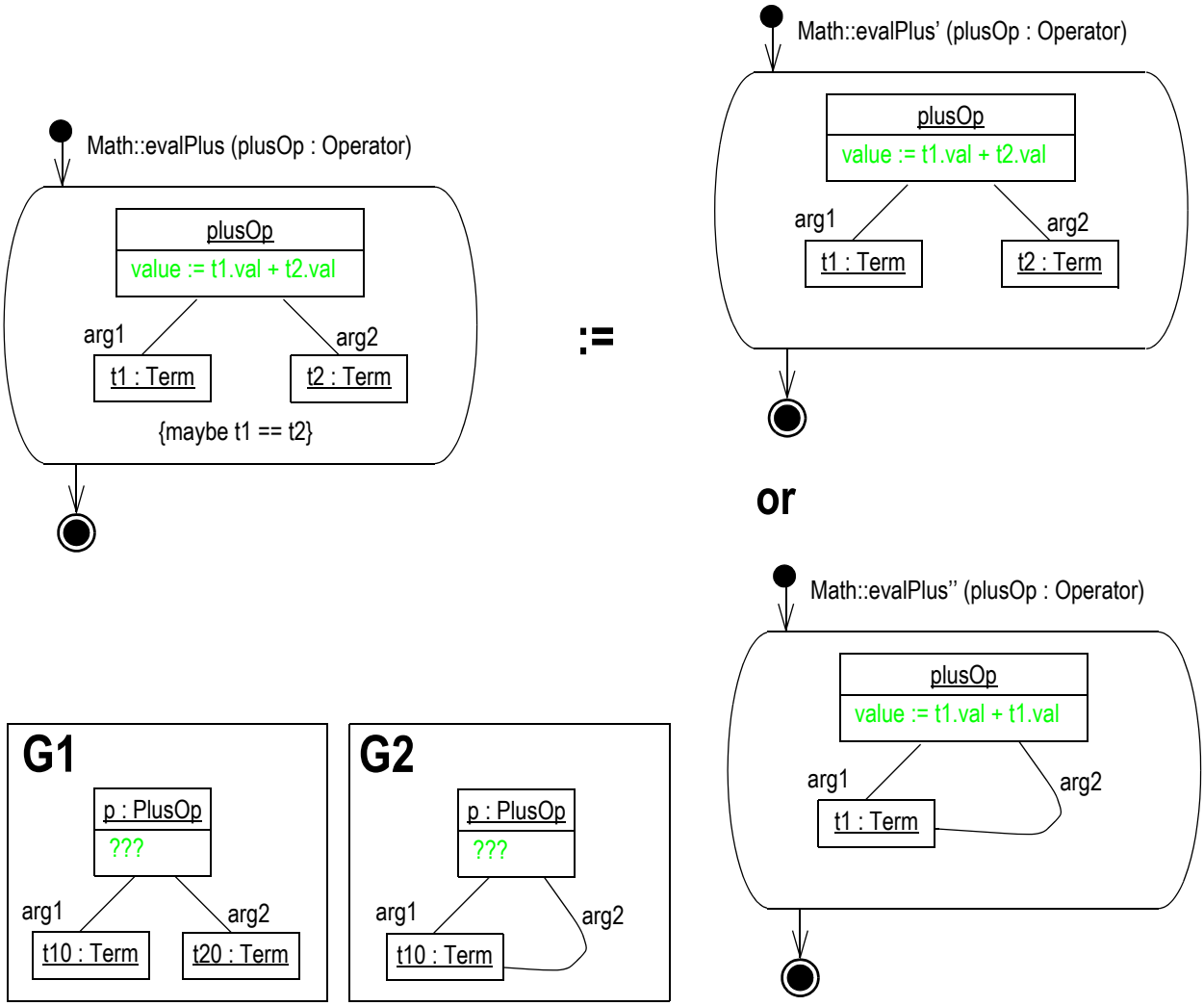
The above effect can occur only for two optional objects of equal types or where the one type is a(n indirect) supertype of the other. This may be checked statically. In case of such a potential conflict, the user has to define a sequence in which the optional elements shall be considered. Textually, this is done considering Opts as a list instead of a set. Graphically, the order may be indicated using indices at the conflicting optional elements, if necessary.

A.6.8 Maybe clauses

In Definition A.9 and Definition A.12 we restricted story patterns to so-called injective matches. This means, different objects in the graph rewrite rule must match different objects in the current host graph. This rule was regarded in the definitions for negative nodes, multi object variables, and optional nodes, too, cf. Definition A.37, Definition A.41, and Definition A.44. Other graph grammar approaches allow so called non-injective matches, cf. [Roz97]. In a non-injective match one node of the host graph may serve as image for several nodes in the rule. Non-injective matches are often difficult to understand and may cause unexpected effects. Thus, we have chosen injective matches as the default semantics for our approach. However, in certain situations it is very handy to allow that two different rule nodes may match the same node in the current host graph.

A very typical area that employs non-injective matches is term-graph rewriting, i.e. the evaluation of functional expressions that are represented as expression trees. In this area it is very common to represent common subterms only once which turns expression trees into directed acyclic expression graphs. In graph G1 of Example A.46 the terms t10 and t20 may represent the same (complex) subexpression. Thus, graph G1 could be simplified as shown in graph G2 where the subexpression is represented only once. From the operator p's point of view, in both cases there exists a subterm reachable via an arg1 and an arg2 edge, respectively. Our rule Math::evalPlus evaluates the plus operator by adding the values of the two argument terms. Usually, rule Math::evalPlus would not be applicable to graph G2, since we require different matches for nodes t1 and t2. To allow that t1 and t2 may match the same node, one may provide a so-called folding clause. Graphically, a folding clause is shown as a boolean constraint with the reserved keyword maybe followed by an equation over two or more nodes of the left-hand side.

Example A.46: Maybe clause handling professors in joinCourses



The semantics of such a maybe clause is defined as follows:

Definition A.47: Maybe clauses

Let grr : bs | BCs ! NCs * Sets ? Opts be a story pattern with bound variables bs and boolean constraints BCs and negative elements NCs and multi object variables Sets and optional elements Opts and
let folds $\subseteq N_{LG} \times N_{LG}$ be a set of folding pairs,

then

grr : bs | BCs ! NCs * Sets ? Opts # folds is a graph rewrite rule with folding with

Sem [grr : bs | BCs ! NCs * Sets ? Opts # folds] =

$$\bigcup_{(x,y) \in folds} (\text{Sem [try (grr\_x\_y : bs | BCs ! NCs * Sets ? Opts \# (folds - (x, y)))]} \cup \text{Sem [try (grr : bs | BCs ! NCs * Sets ? Opts \# (folds - (x, y)))])}$$

where

grr_x_y := the rule grr where all occurrences of y are replaced by occurrences of x

Note, a story pattern consists of two graphs which consist of sets of nodes and edges and attribute values. If the renaming of y to x creates double entries in one of these sets, these double entries are unified, automatically. In case of attribute values in the right-hand side

different attribute assignments could result for the same node and attribute name. This is forbidden, statically.

Similarly it is forbidden to fold nodes x and y where one is destroyed and the other not (identification condition).

There are some special cases where nodes of different kinds are folded:

1. $x \in N_{LG}$ - NCs - Sets - Opts, $y \in$ NCs: In this case the folding definition applies to the grr_y help rule of Definition A.37 that deals with negative element y. The or of the folding condition becomes the boolean or within the BNCs.
2. $x \in N_{LG}$ - NCs - Sets - Opts, $y \in$ Sets: The folding condition applies to the fill_one_grr_xi rule and the one_grr_xi rule of Definition A.41 that deal with y. Graphically, this folding condition is notated as {maybe $x \in y$}.
3. $x \in N_{LG}$ - NCs - Sets - Opts, $y \in$ Opts: The folding condition applies to that help rule of Definition A.44 that contains the optional node y as a normal node.
4. $x \in$ NCs, $y \in$ NCs: This case makes no sense, since negative nodes are handled one by one and there is no (help) rule that contains both negative nodes. Thus, this case is forbidden.
5. $x \in$ NCs, $y \in$ Sets: In this case drop just the corresponding SetBNCs condition in Definition A.41. Graphically, this folding condition is notated as {maybe $x \in y$}.
6. $x \in$ NCs, $y \in$ Opts: Apply Definition A.44 to y, first and than proceed like two cases above.
7. $x \in$ Sets, $y \in$ Sets: Folded elements may be contained in both sets. Usually, this is forbidden by the noSetMatch (y) constraint for the fill_one_grr_y rule. This noSetMatch (y) constraint shall no longer respect node x. This applies to the symmetric case, too. Graphically, this case is notated as {maybe x intersects y}
8. $x \in$ Sets, $y \in$ Opts: The folding condition applies to that help rule of Definition A.44 that contains the optional node y as a normal node. Once the optional node becomes a normal node, case 2. applies.
9. $x \in$ Opts, $y \in$ Opts: The folding condition applies to that help rule of Definition A.44 that contains the optional node y as a normal node. Once the optional node becomes a normal node, case 3. applies.
10. In cases where x and y have swapped roles, symmetric definitions apply.

Thus, a rule with a folding clause is replaced by two rules without that folding clause. The first rule just omits the folding clause. The second rule is derived by renaming all occurrences of one folded element by the other folded element, see also Example A.46. In case of multiple folding clauses this step is repeated until all foldings are resolved.

Note, the renaming step above glues the two folded nodes together within the left-hand and the right-hand side. In some cases this may cause unexpected effects. Consider a clause folding nodes x and y. The rule could delete node x and keep node y alive. This corresponds to the so-called identification condition. Different graph grammar approaches deal with this situation in different ways. Some define that the matched node is deleted, some define that the matched node survives. Like in the Progres approach, we forbid such conflicts statically and check it at compile time, since due to our experience such a situation most often reflects a specification error, i.e a situation not considered by the user.

If some of the folded nodes are negative, optional, or multi valued, special conditions apply. Basically, the folding is forwarded to the help rules that do not longer contain such special nodes.

A.6.9 Dealing with ordered (and sorted) associations

The object oriented data model provides ordered and sorted associations. So far, our story patterns do not utilize this additional concepts explicitly. In this chapter we will introduce special operators to look-up the first, last, next, previous, indirect next, or indirect previous objects in sorted and ordered associations and to provide insertion positions on the creation of edges for ordered associations.

Definition A.48: Identifying the position of objects in story patterns

Let grr = (LG, RG) be a graph rewrite rule and el be an ordered or sorted association and s, t $\in N_{LG}$ and e = (s, el, i, q, t), e2 = (s, el, i2, q2, t2) $\in E_{LG}$ and G be the current host graph, then

first (e) := let match (e) = (match (s), el, i', q', match (t)) $\in E_G$ be the match of e, then
$\forall$ e' = (match(s), el, j, r, u) $\in E_G$ must hold
i' <= j — if el is ordered or
match (t) <= u — if el is sorted according to <=

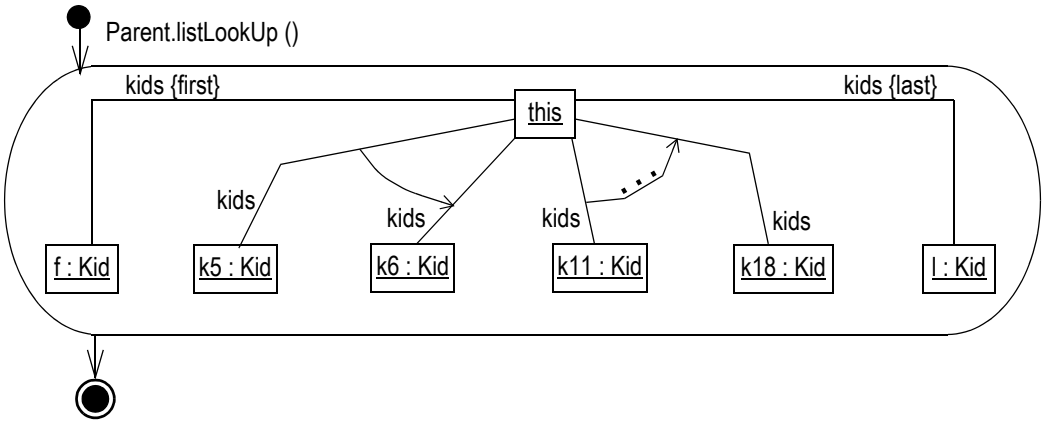
last (e) := let match (e) = (match (s), el, i', q', match (t)) $\in E_G$ be the match of e, then
$\forall$ e' = (match(s), el, j, r, u) $\in E_G$ must hold
i' >= j — if el is ordered or
match (t) >= u — if el is sorted according to >=

succ (e, e2) := let match (e) = (match (s), el, i', q', match (t)) $\in E_G$ be the match of e and
let match (e2) = (match (s), el, i2', q2', match (t2)) $\in E_G$ be the match of e2,
then it must hold:
not $\exists$ e' = (match(s), el, j, r, u) $\in E_G$ such that
i' <= j <= i2' — if el is ordered or
match (t) <= u <= match (t2) — if el is sorted according to <=

succ* (e, e2) := let match (e) = (match (s), el, i', q', match (t)) $\in E_G$ be the match of e and
let match (e2) = (match (s), el, i2', q2', match (t2)) $\in E_G$ be the match of e2,
then it must hold:
i' <= i2' — if el is ordered or
match (t) <= match (t2) — if el is sorted according to <=

Graphically, the constraints first and last are shown in curly braces attached to the corresponding edge. The succ constraint is shown as a small arrow from e to e2 and the succ* constraint is shown as a small arrow labeled with a ". . ." tag, cf. Example A.49.

Example A.49: Looking up the position of objects



In Example A.49 we look up several members of the kids association which might be ordered. In this case object f matches to the first member of the list of kids attached to the this object. Accordingly, l

matches the last list member. Nodes k5 and k6 must be direct successors within the kids list and objects k11 and k18 must be direct or indirect successors. Similar properties hold for sorted associations.

Note, above positional constraints are defined for elements of the left-hand side LG, only. In case of insertion into sorted associations the relative position of the new object is automatically defined by the order relations "<=" provided for the contained object. However, ordered associations model user defined orders of elements. Thus, we need some language elements allowing to indicate insertion positions for new edges in ordered associations:

Definition A.50: Providing positions for the insertion in ordered associations

Let grr = (LG, RG) be a graph rewrite rule and el be an ordered or sorted association and s, t ∈ N_{RG} and e = (s, el, i, q, t) ∈ $AddE_{grr}$ and e2 = (s, el, i2, q2, t2) ∈ $CoreE_{grr}$ and G be the current host graph, then

first (e) := let j be the minimal index i' of edges in
{ e' = (copyMatch (s), el, i', q', u) ∈ E_G } (or j = 2 if the set is empty),
then i = j - 1

last (e) := let j be the maximal index i' of edges in
{ e' = (copyMatch (s), el, i', q', u) ∈ E_G } (or j = 0 if the set is empty),
then i = j + 1

succ (e2, e) := let j be the minimal index i' of edges in
{ e' = (copyMatch (s), el, i', q', u) ∈ E_G | i' > i2 }
(or j = i2 + 2 if the set is empty),
then i = (i2 + j) / 2

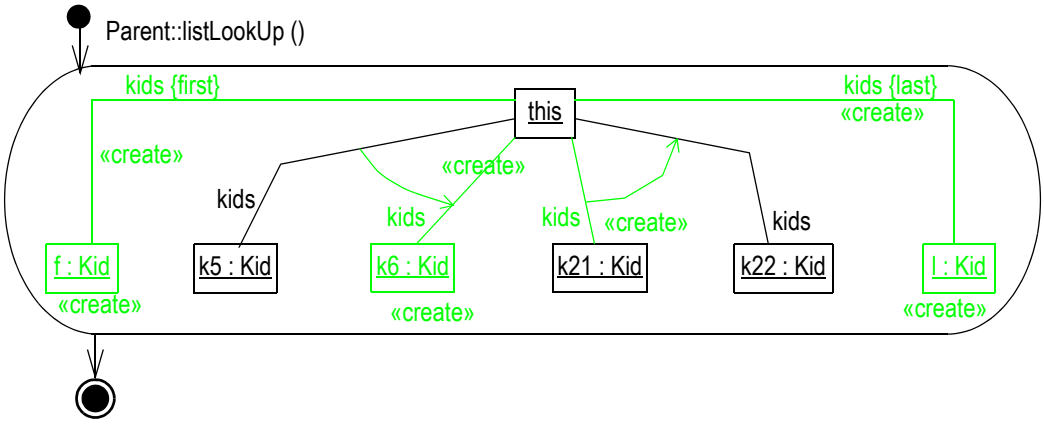
succ (e, e2) := let j be the maximal index i' of edges in
{ e' = (copyMatch (s), el, i', q', u) ∈ E_G | i' < i2 }
(or j = i2 - 2 if the set is empty),
then i = (i2 + j) / 2

Graphically, we use the same notation as in Definition A.48.

For the insertion at the first (last) list position, we compute the minimal (maximal) existing index and subtract (add) 1. In case of an empty list we use index 1. Inserting a new list element after a given element t' is done by computing the index j of the successor of t' in the list. The index of the new list element is then computed as the median of j and the index of t'. Note, this may create indices that are no natural numbers. However, we have defined the rational numbers as domain for edge indices. Using rational numbers as indices allows to insert a new index between to given indices, always. Thus, in our approach it is not necessary to shift the indices of subsequent list elements on the insertion of some new list member. This has the drawback, that the index number does not directly reflect the position of an element within the list. Formally, one may easily circumvent this drawback by defining an operator that looks up the element with the i-th lowest index. An usual doubly linked list implementation does not provide an efficient index based look-up operation, anyway. Finally, if one wants to access an element of an association via an index he or she may use a qualified association using e.g. consecutive natural numbers as keys for this purpose. In this case one would have to model the index shift operation required for insertion and deletion himself or herself. We leave this as an exercise for the interested reader.

Note, for insertion we do not provide a succ* position. Position succ* would mean something like insert somewhere behind the given position. This would create a nondeterministic insertion operation. This kind of nondeterminism is not supported by our implementation.

Example A.51: Insertion in ordered associations



Example A.51 shows the different possibilities to specify insertion positions for new links. Object f is going to be inserted as the new first element of the kids list. If there are already elements within the kids list, the index of the kids edge attached to f will be the lowest existing index minus 1. It will be index 1, otherwise. Object k6 is going to be inserted directly behind object k5. Assume object k5 has the index 5 and its direct successor has the index 6, then the kids link attached to k6 will become the value 11/2. Object k21 is inserted before object k22, similarly. Finally, object l will become the last list member.

A.7 Collaboration messages and sequence diagrams

So far, our semantics definition does not yet cover method calls and assignments as part of a story pattern. All method calls and assignments are contained in their own activity shapes, cf. Definition A.23 on page 123. Sequences of such operations may be combined using the sequential, if-, or while-composition. Although, this suffices from a theoretical point of view, in practice it turned out to be very handy to provide method invocations and assign statements within story patterns. Thus, we allow to use UML collaboration messages with sequence numbers in Dewey decimal numbering notation within story patterns:

Definition A.52: Collaboration messages

Let grr = (LG, RG) be a story pattern and sd be a story diagram

If sd consists of method calls and assignment statements, only, and if all method calls within sd are invoked on objects in N_{RG}, then the sequences, if-compositions, and while-compositions of sd may be shown as collaboration messages within grr.

Sequential composition uses consecutive numbers within the current nesting level e.g.: 1.1, 1.2, ...

If (cond1) then stat2 end is shown like 1.1 [cond1] and the numbers for the operations in stat2 get the prefix 1.1., e.g.: 1.1.1: stat2

While composition is shown as 1.1 [while cond1]

In addition we introduce the for composition shown as 1.1 [lvar = li..hi] which corresponds to 1.1: lvar = li; 1.2 [while lvar <= hi]; 1.2.1: ... ; 1.2.n: lvar++;

Note, our use of collaboration messages differs from UML in two ways. First, we do not provide thread identification prefixes. Story diagrams specify the bodies of certain methods and so far we support the sequential case, only. Second, there is no micro parallelism allowed as for example 1.1a; 1.1b. We are going to translate collaboration messages to nested Java statements, directly, and therefore we do not yet support this kind of micro parallelism.

The use of collaboration messages in story diagrams is quite different to the use of collaboration messages in scenario diagrams like usual UML sequence or collaboration diagrams. Let an UML collaboration diagram used within a scenario contain a collaboration message 1 [cond1]: m1() send to some object n. In this case, the collaboration message 1.1: m2 () would specify a substep of method m1 executed by object n. This is meaningful for scenarios where one models the cooperation of several objects. In contrast, story diagrams model the bodies of a certain methods. The current method may invoke a method on a neighbor object. However, the body of the invoked method is specified in the called story diagram. Thus messages send by the called method should not be part of the current story pattern. Consequently, in a story diagram all collaboration messages must originate from the this object. This allows us to use nested collaboration numbers to model control structure nesting. Thus, if a story pattern contains a collaboration message like 1 [cond1]: m1() send to some object n, then the nested collaboration message 1.1: m2 () belongs to the then part of the corresponding if-composition. This allows us to model conditional statements with complex bodies as nested collaboration diagrams, conveniently.

Example A.53 shows a simple story pattern employing different kinds of collaboration messages and their translation into explicit control structures. Note, collaboration messages send to the this object itself may omit the arrow that usually indicates the target. This allows to arrange such messages more flexible.

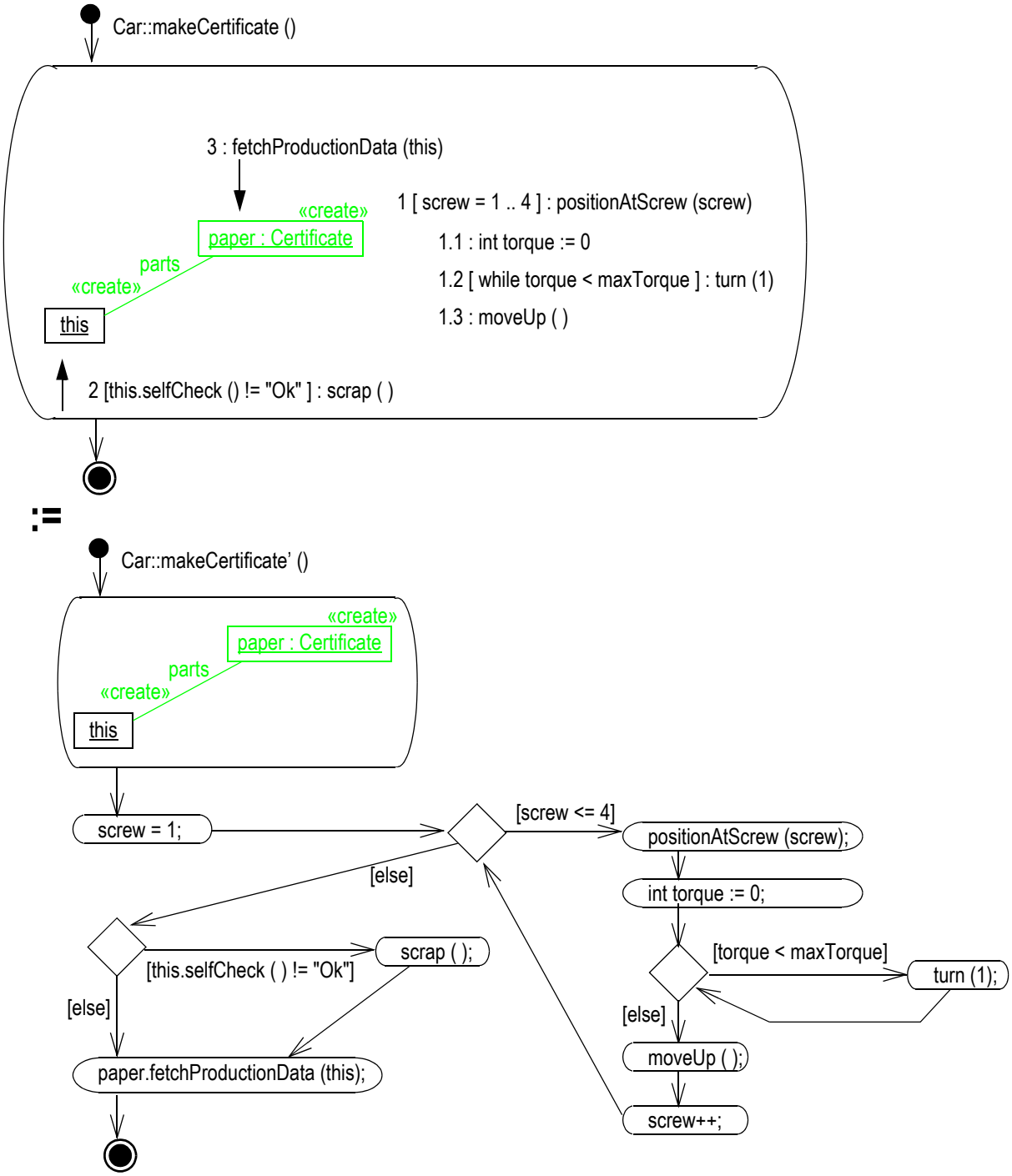
Note, semantically the collaboration messages are executed after the graph rewrite step. Thus collaboration messages must not be send to destroyed objects. In principle, collaboration messages may invoke methods that modify the object structure that is depicted in the calling story pattern. This may have confusing effects. For example, a story pattern may create some object and invoke some operation and pass the new object as parameter. The called operation could just destroy the new object. Thus, after the complete operation there is no new object although, the story pattern clearly shows the creation of an object. We consider this as a problem of poor modeling style. In general, a story pattern should have the depicted effects. Called methods may perform operations within the neighborhood of the story pattern application, however this should not obfuscate the depicted graphical operation.

UML collaboration diagrams and UML sequence diagrams have a common semantics, cf, [BRJ99]. Both diagram kinds rely on the same cut-out of the UML metamodel. The different diagram kinds may depict the same scenario just in different ways. The difference is, that collaboration diagrams emphasize the "structural organization" while sequence diagrams emphasize the "time ordering of messages". Therefore, we allow to depict story patterns as sequence diagrams, too, cf. Example A.54.

Example A.54 shows the collaboration messages of Example A.53 as a sequence diagram. Note, such a sequence diagram is shown below its story pattern within the so-called sequence diagram compartment of the corresponding activity shape. The sequence diagram compartment is separated from the story pattern via a solid vertical line. The sequence diagram employs no objects but it refers to the nodes within the corresponding story pattern. If possible, the "life lines" are shown vertically below the corresponding object. To avoid layout restrictions and to facilitate the identification of correspondences between nodes in the upper part of the story pattern and their life lines, a node may be connected to its life line via a dashed line.

If a story pattern employs only bound or created variables and if it has no links, one may arrange all objects just above their life lines and omit the separating solid line from the sequence diagram compartment. This turns the story pattern into a usual sequence diagram.

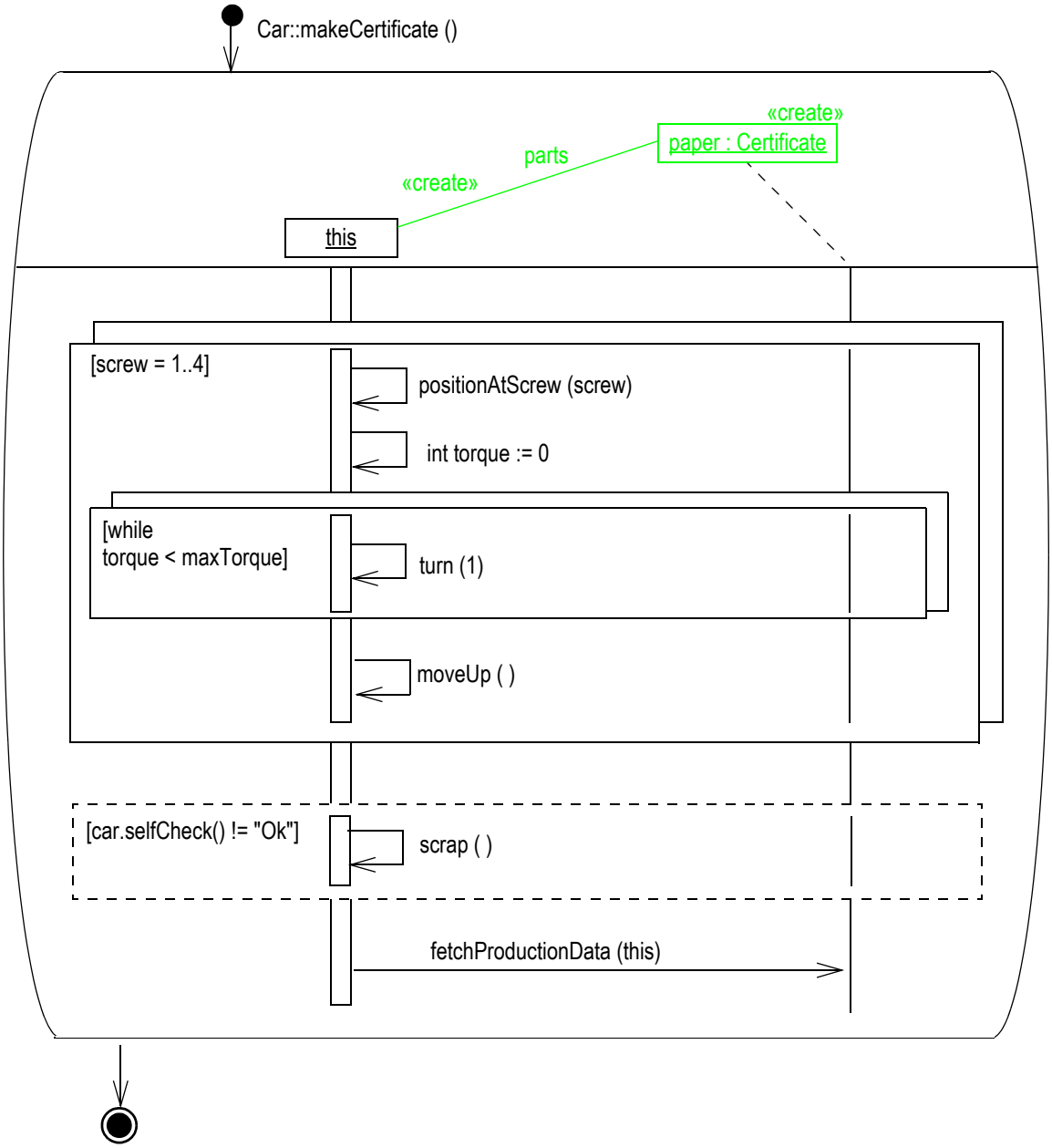
Example A.53: Using collaboration messages within story diagrams



Note, a story diagram may consists of several story patterns, some shown in collaboration diagram notation and some shown in sequence diagram notation. Frequently one uses first a story pattern in collaboration diagram notation to look-up a number of objects and second a story diagram in sequence diagram notation to show complex message flows.

Note, within collaboration diagrams we allow complex statements like if- and while-compositions. To preserve the equivalence of collaboration diagram notation and sequence diagram notation, we have extended the sequence diagram notation correspondingly. Our notation for complex statements has been inspired by [Sys97].

Example A.54: Showing a story board with collaboration messages as sequence diagram



Definition A.55: Sequence diagram notation for collaboration messages

Let grr = (LG, RG) be a story pattern and sd be a story diagram

If sd consists of method calls and assignment statements, only, and if all method calls within sd are invoked on objects in N_{RG}, then the sequences, if-compositions, while-compositions, and for-compositions of sd may be shown in the sequence diagram compartment of rule grr.

The sequence diagram compartment is shown in the lower part of the activity shape containing grr. It is separated from the story pattern via a solid vertical line. This separator may be omitted if grr contains no links and if the objects of grr are properly arranged above their life lines.

Each node x ∈ RG gets an additional "life line". The life line for the this node is shown as a vertical bar below the this node (note, this is the only active object). The life line of the other nodes is shown as a vertical line below the corresponding objects. All life lines start at the top of the sequence diagram compartment. The life line may be moved horizontally

e.g. for layout reasons. If appropriate, the life line may be connected to its object in the story pattern via a dashed line.

Collaboration messages are shown by labelled arrows in their temporal order from top to button. All arrows originate from the life line of the this node. Messages from this to other objects are shown using a vertical arrow to the life line of the target object. Messages from this to this are shown as a short arrow from the this life line back to itself. The arrow is labeled with the actual statement.

Sequential composition is shown by ordering the arrows in their temporal order from top to button.

An if-composition like if (cond1) then stat2 end is shown using a dashed box crossing all life lines. The upper left corner of the dashed box shows the condition [cond1]. All collaboration messages contained within the dashed box belong to the then branch stat2 of the if-composition.

A while-composition is shown using two stacked boxes crossing all life lines. The upper left corner of the two stacked boxes shows the iteration condition. All collaboration messages contained within the two stacked boxes belong to the body of the while composition. This body is repeated while the iteration condition evaluates to true.

A for-composition is shown like a while composition but with an iteration condition like [lvar = li..hi].

Our sequence diagrams provide complex control flow in order to preserve the equivalence to our collaboration messages in Dewey decimal numbering notation. However, in sequence diagrams these complex control structures are hard to read and should be used with care. But, for a long sequence of straight forward control flow, a sequence diagram compartment are an appropriate alternative to collaboration messages using Dewey decimal numbering notation.

Due to our experience, the sequence diagram notation is less appropriate for the specification of method bodies then for modeling example scenarios in the earlier OO analysis phases. Within an example scenario, a complex series of steps employing several objects may be exemplified. Within the specification of a method body the only active object is the this object and the contributions of the other objects, i.e. their internal reaction to an received message, is omitted due to information hiding principles. This cuts a lot from the expressiveness of sequence diagrams.

A.8 Story charts

Story charts are an adaption of statecharts to our approach. Statecharts have been introduced by [HG96]. Originally, they were intended for the specification of telecommunication devices and their communication. In UML, statecharts have been adapted for the specification of reactive object oriented systems. In UML, statecharts can be used for many different purposes. They can model the behavior of a whole application or just of a single method. Statecharts may model all possible sequences of method invocations on a certain object or they may model state dependent reaction of certain objects on the reception on certain events. Each of these uses leads to a different semantics of the given statecharts. In addition, the different usages employ different statechart language features. In order to formalize the semantics of statecharts, one must clearly identify which part of an application and which behavioral aspect of this part is modeled by the provided statechart.

In this work, we assume that statecharts are used to model the reaction of *active objects* on events sent to them. A given graph may contain an arbitrary number of active objects each running its own execution thread. Active objects may be created or deleted at runtime. These active objects share a com-

mon graph, that models the overall system state and may be used to exchange data and to send events to each other. Events are not broadcast but each event is explicitly targeted to a certain receiver object.

Thus, statecharts are attached to (active) classes. We turn events into so-called *event methods* of the corresponding classes in order to provide an uniform and convenient way of invoking some service on an object. The event methods build the public interface of the modeled class.

A statechart specifies the reaction of an active object to a received event in terms of state changes and by executing entry, exit, and do actions and actions attached to transitions. Usually, these additional actions are given as pseudo code, only. In our approach, we formalize these actions using story diagrams. Thus, the actions and guards employed within a statechart are modeled by story diagrams given as so-called *action methods* of the modeled class. Altogether, a statechart defines which action methods will be called as reaction on a(n event) method invocation and depending on the object's current state.[4]

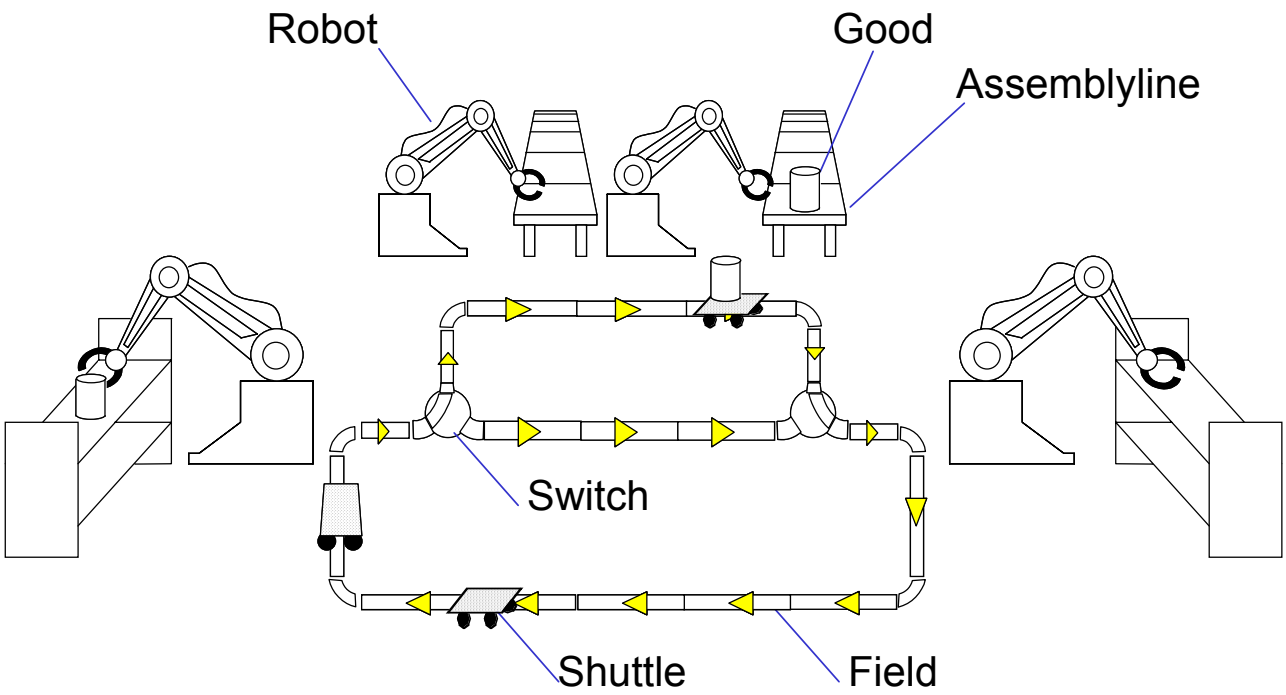
A.8.1 A reactive system example

Our approach especially targets complex, concurrent, object oriented systems. These object oriented systems employ multiple concurrent agents that perform complex tasks. Each agent may perform complex computations. In addition, complex object structures may be employed in order to provide a sophisticated information basis for complex behavior. As stated in chapter 1, object structures are able to represent arbitrary complex system states. Well known is the representation of the structure and current state of complex graphical user interfaces. The same holds in principle for all kinds of office applications like text processors, spread sheet calculators, (vector) graphics programs, and WWW browsers. This is also true for runtime data structures of operating systems, compilers, CAxx tools, database systems, WWW servers, etc. Other applications with complex state information and concurrent components are e.g. workflow management and groupware systems. Even embedded systems are now moving to a more decentralized organization that employs more intelligent components. A modern car employs several dozen controller nodes, e.g. for operating the windows, the seat, the control devices, the radio, the breaks, and last but not least the car engine. All these elements are connected to implement a cooperative behavior. For example, if a person starts the car using its personal car key, seat and mirrors may automatically move to the positions preferred by that person and the radio may recall the preferred radio station.

As another example for a complex, object oriented system with concurrent components, this chapter revisits the production process from chapter 1. This production process models a factory with various manufacturing places and with shuttles transporting goods from one manufacturing place to another via transportation tracks, cf. Example A.56. The example stems from the ISILEIT project funded by the German National Science Foundation (DFG). The goal of the project is the development of a formal and analyzable specification language for manufacturing processes. In addition, a code generator shall provide automatic code generation for driving the constituent parts of a manufacturing process, namely shuttles, robots, assembly lines, switches, etc. Note, in a modular factory all these constituents have local control. We plan to model the manufacturing process up-front and to simulate its functionality in order to validate if everything works correctly and then to generate the software that runs the constituents of the manufacturing process. Overall, more flexible processes adjusting to market demands more quickly will be achieved.

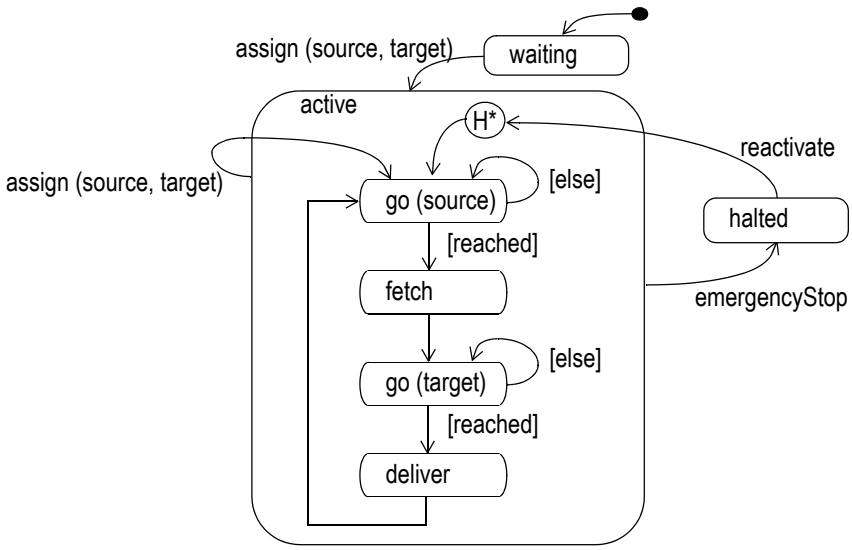
4. Action methods may call event methods on other active objects, thereby sending them events.

Example A.56: Simple factory example



The basic behavior of a shuttle is specified in the statechart shown in Example A.57. Initially, a shuttle is just waiting. Then the shuttle is instructed to transport goods from a source place to a target place by sending it an assign event. The shuttle reacts on such an event by switching from the waiting state to the active state and by storing its source and target place, internally. The active state is a complex state with state go(source) as its initial sub-state. In state go(source) the shuttle tries to go to the source place of its current order. It finds his way autonomously using its own routing algorithm and avoiding collisions with other shuttles. Once the shuttle reached its source place it fetches a good and switches to the go(target) state. When the shuttle reaches its target place it delivers the transported good, turns back to the go(source) state, fetches another good, goes to its target, and so on. The shuttle will execute this procedure until it "dies".

Example A.57: The statechart of a shuttle



But our shuttles are active objects. Thus, a shuttle may be reassigned with a different instruction at any time by just sending it another assign event. In case of an emergency stop, a shuttle is halted by

an emergencyStop event. In state halted the shuttle may be reactivated by sending it a reactivate event. When a shuttle is reactivated, it switches into the history state of the active state. The history state stores the last substate of the shuttle before it switches into the halted state and recalls that state when the shuttle is reactivated.

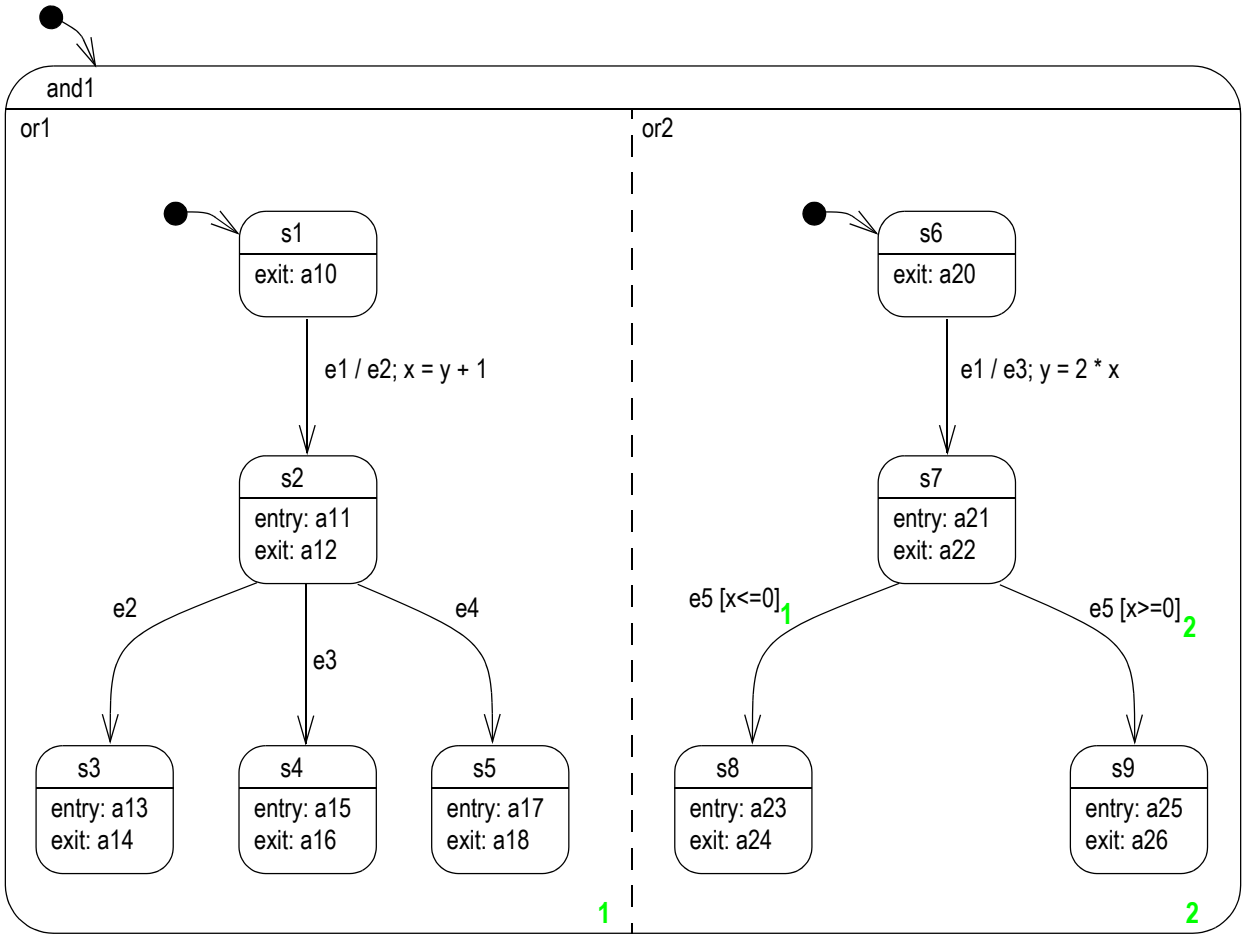
Actually, our example specification is much more complex. We have downstripped the example considerably to facilitate its understanding.

A.8.2 Key semantics concepts

For statecharts a number of different formalizations exist. Harrel has proposed the micro-step semantics [Har87, HPSS87] and the super-step semantics [H+90, Har96]. Our work proposes a sequential semantics. To illustrate the micro-step semantics consider Example A.58.

Let the statechart of Example A.58 be in the initial state and the event e1 is received. We first consider the sequence of actions that is triggered. According to the micro-step semantics, the transition from s1 to s2 and the transition from s6 to s7 are enabled and in a first micro-step these transitions fire, "simultaneously". This creates the events e2 and e3. In the second micro-step, these two events are considered. Since we are in state s2 now and events e2 and e3 are currently available, the transition from s2 to s3 and the transition from s2 to s4 are both enabled. According to Harrel, a nondeterministic choice is made and we switch either to s3 or to s4. The key point is, that the micro-step semantics deals with sets of events that are considered, simultaneously, and that may fire multiple different transitions. In case of conflicts, nondeterministic choices are made.

Example A.58: Micro-step vs. super-step vs. our sequential statechart semantics



Another key property of the micro step semantics is the immediate handling of additional external events received while handling the initial event. Assume that the e1 event is closely followed by an e4 event. The micro-step semantics is inspired by an implementation with hardware circuits. Events are values at some input wires and at a certain clock tick, the values of the input wires are read, the logical circuits calculate result values and these result values are written to output wires. If some outputs are connected to input wires of the same statechart, these output values are considered at the next clock tick. However, if some other input values have changed meanwhile, on the second clock tick these events are considered, too. Translated to our example, this means, that in the first clock cycle / micro-step event e1 is consumed and events e2 and e3 are created and event e4 may be received. Then, the second micro step considers all three events. Thus, in our example we would either switch to state s3 or s4 *or* s5.

The super-step semantics deals with additional external events, differently. The super-step semantics propagates *zero time execution speed* for micro-steps. Actually, this means that even multiple micro-steps are executed that fast, that no new external event can occur meanwhile. From the implementation point of view this means, additional external events are blocked as long as internal events trigger new micro-steps. Only if no new internal event has been created by the last micro-step, the next external events are considered.

In our example super-step semantics thus means, if e1 and e4 are received sequentially, in an arbitrary short period of time, the first micro-step consumes e1 and the internal events e2 and e3 are created. This is done in virtually zero time. This means, the second micro step considers e2 and e3, only. Event e4 is considered as not-yet-occurred. Accordingly, we switch either to state s3 or to state s4. This creates no new internal event and thus the super-step is completed, time starts running again and e4 may now occur / be considered.

Note, Harrel's semantics for statecharts is especially suited for single processing units that control a large number of system components in parallel. According to Harrel's statechart semantics, multiple events are considered in parallel in each micro step. Thus, multiple system components like shuttles and switches may send signals to a central control unit simultaneously and all these signals are processed simultaneously and all components receive simultaneous or immediate responds. The parallel handling of multiple system components is especially supported by and-states. The central processing unit may employ one and-substate for each system component. Each and-substate models the behavior of a single system component independently of all other components. Coordination between different and-substates may be achieved by exchanging events.

While Harrel's statecharts are appropriate for small and medium size systems, they do not scale to large numbers of complex system components. For large numbers of system components the central processing unit soon becomes a bottleneck. This problem is solved by splitting the central processing unit into multiple processing units and by distributing control responsibilities for different subsystems among such units. Communication between the different processing units may be organized via a bus-system that broadcasts events or by organizing the different processing units into some (hierarchical) topology. However, if such a system sticks with Harrel's statechart semantics and it still employs the idea of consuming all events throughout the system within synchronous micro steps. This implies the notion of some (virtual) global system clock or mechanism that provides common clock ticks synchronizing the micro steps of the different processing units. Such a common clock tick now becomes a serious bottleneck for more complex behavior of system components. In our example, a shuttle shall be able to compute its traveling routes, autonomously. This enables the shuttle to react e.g. on malfunctioning assembly robots by rerouting itself to an alternative production place. Such rerouting steps may require complex computations and data retrievals and therefore they may need some time. Within a system that sticks to Harrel's statechart semantics and that employs common clock ticks, the

next common clock tick can not occur until all system components have consumed their incoming events and until all system components have computed their results. Thus, all system components have to wait for the slowest component to complete its task. This has the consequence that the reaction of a single and-substate to an event must not be complex or time consuming. Complex computations have to be scattered into multiple steps where each step fits into a narrow clock cycle. Such a scattered computation may always be interfered by some other events that require direct handling. In practice, these problems prevent the realization of complex or intelligent component behavior.

To overcome these problems, our approach drops Harrel's statecharts semantics. We do not assume a common micro step in which all system components consume their events, simultaneously. Instead, we decouple the system components into autonomous processes. These autonomous processes execute on their own pace independently from each other. Complex operations may take their time to complete while simple operations may proceed without delay. Processes still communicate by sending events to each other. However, receiving processes are not forced to react on events instantaneously but the receiving process is allowed to queue incoming events and to complete complex computations, first. Complex computations do not need to be scattered into multiple time slots. This enables the realization of intelligent system components.

Harrel's statecharts semantics contains three potential sources of nondeterminism, consuming sets of events simultaneously, a single event firing multiple transitions in parallel and-substates simultaneously, and a single event firing multiple transitions with overlapping guards leaving the same state. Such nondeterminism may cause unexpected, unrepeatable, and unpredictable system behavior. Thus our approach tries to handle these cases, deterministically. This leads to our *sequential statechart semantics*

In Harrel's statecharts, multiple events may be created by the different system components within a single micro step. In Harrel's semantics all these events are considered to occur simultaneously. Accordingly, all these events are consumed in parallel within the subsequent micro step. By dropping the common clock tick and by decoupling the system into autonomous processes that execute on different pace, the notion of simultaneously created events does not longer exist in our approach. While one system component executes a complex task, a number of events may arrive at this component at different points in time. When the component has finished its complex task it could try to consume all waiting events in parallel. However, consuming sets of events in parallel is a potential source of nondeterminism. Recall Example A.58 where events e2 and e3 are consumed in parallel in the second micro step. Events e2 and e3 trigger two conflicting transitions one from s2 to s3 and one from s2 to s4. Our approach avoids this kind of nondeterminism. In case of multiple waiting events our system components consume them one after the other. Currently, our system components consume events in the order of arrival. (Note, we assume that the communication layer utilized for event transportation serializes incoming events for a given process automatically.) However, alternative event serialization schemes are possible. This will be discussed together with Definition A.64 and Definition A.66.

In Harrel's statecharts, parallel and-substates are executed, simultaneously. In Example A.58 this again creates the situation, that events e2 and e3 occur simultaneously. Since our approach consumes events sequentially, our semantics has to choose some order for the creation and transmission of such "simultaneous" events. We resolve this problem by defining a prioritization for and-substates. The prioritization of and-substates is defined by the user through small index numbers in the lower right corner of each partition of an and-state. Default index numbers are created by our tool, automatically. However, the user may change these numbers, later on. Accordingly, in Example A.58 the left partition of the and-state is considered first and the right partition is considered second. This means, event e2 occurs first and event e3 second.

This prioritization of and-substates also solves another semantic problem of Harrel's statecharts. In Example A.58, Harrel's simultaneous execution of the two and-substates creates a semantic problem for the two assignment statements attached to the e1 transitions. The e1 transition from s1 to s2 executes the statement x=y+1and the e1 transition from s6 to s7 executes the statement y=2*x. This raises the question, which values of x and y are used during expression evaluation. Let x and y initially be 1. If statement x=y+1 is executed, first, and statement y=2*x, second, x becomes 2 and y becomes 4. If the statements are executed in other order, y becomes 2 and x becomes 3. According to Harrel, the order of execution should not matter. Thus, Harrel defines that during expression evaluation always the old values of variables are used and the assignments become effective after all actions have been executed. Thus, according to Harrel, x becomes 2 and y becomes 2. In our approach, we prioritize the execution of and-substates. Accordingly, we first execute the transition from s1 to s2. This enqueues an e2 event to our event queue and assigns 2 to variable x. Second, the transition from s6 to s7 fires and enqueues an e3 event behind the e2 event to our event queue. Since x has already been changed to 2, in our approach variable y becomes 4. (Note, this is no nondeterministic choice but the definite deterministic result.)

Actually, Harrel's semantics relieves the user from worrying about execution sequences of statechart actions and statements. Actions from one transition are executed in isolation from concurrent actions of other transitions. Unfortunately, the implementation of this behavior is quite demanding. The standard approach is the double-buffering of all variable values. One copy of all variables holds the old values used for expression evaluation and one copy collects the new values resulting from assignments. This is affordable for systems that employ a limited number of boolean and integer variables, but an object-oriented system may employ complex object-structures consisting of thousands or millions of objects. Copying such large object structures would be very expensive. Sophisticated copy-on-write techniques may allow to copy only modified parts of the object structure and to share unmodified parts. However, a simple sort operation may effect a large fragment of the object structure and thus require to copy millions of objects. Alternatively, database transaction mechanisms may be employed. This solution implies locking mechanisms and one has to deal with various problems caused by conflicting locks. Our approach does not have such implementation problems since and-substates are considered one after the other and each and-state is executed in a conventional sequential way. Thus our sequential statecharts semantics facilitates the use of complex object structures within statecharts.

Note, our sequential handling of and-states cuts the central parallelism concept of statecharts. In our approach, a single active object executes all its operations within a single process or thread, sequentially. However, our approach provides multiple active objects that execute, concurrently. Thus, we replace the "common-global-clock-tick" parallelism provided by Harrel's and-states with independent, decoupled, concurrent processes that may be created and reorganized, dynamically. We still provide and-states due to their expressive power. Within our approach, and-states still allow to separate multiple concerns of one active object into multiple and-substates. Each and-substate may deal with one concern of the active object. Our approach only changes the scheduling of the handling of the different substates and the concurrency mechanisms. If our sequential handling of and-substates becomes a bottle neck for some and-state, the problem may be solved by splitting that and-state into multiple substatecharts and by introducing new active objects for these substatecharts. These new active objects then execute the substatecharts, concurrently.

Like and-states, a transition with multiple guards can be a source of nondeterminism. In Example A.58 such a transition with multiple targets is leaving state s7. In principle, the guards of such a transition must be mutually exclusive. However, this constraint may be violated by the user (without purpose). While in our example it is easily seen that the constraints are not mutually exclusive, generally the checking of this property is an undecidable problem. Thus, there exists no possibility to guarantee

mutual exclusiveness at compile-time. If in a certain situation multiple guards evaluate to true, Harrel proposes a nondeterministic choice. This might be implemented by a random choice or by an automatic choice performed by an execution tool. Finally, one could check all alternatives at runtime and raise an exception in case of a violation and stop the execution. Our approach, visits the guards in a fixed user defined order indicated by index numbers subscribed to the guards, cf. Example A.58. Again a default order is created automatically and this order may be changed by the user, later on. Thus, if we are in state s7 and an e5 event is consumed, we first consider the transition from s7 to s8. If the guard evaluates to true, we switch to state s8, otherwise the transition from s7 to s9 is considered. If that guard holds, we switch to s9, otherwise we stay in state s7. Thus, if both guards are fulfilled, we would deterministically switch to state s8. State s9 is considered only if the first guard evaluates to false.

Our sequential statecharts semantics faces a semantic definition problem similar to the distinction of Harrel's micro step and Harrel's super step semantics: when are additional external events considered that arrive while some component is still consuming secondary internal events created during the consumption of a previous external events. Revisit Example A.58: when an external event e1 is consumed, first the left and-substate creates an e2 event and sends this internal event to the executing component itself. Next, the right and-substate is considered and an internal e3 event is sent to the executing component itself. Note, the execution of the different and-substates may involve complex computations and thus it may need some time. In addition, the transportation of events may need time. This holds especially for events send to another system component that may be executed on some other processing unit but also for system components executed on the same processing units. Thus, while event e1 is consumed, new external events may arrive either before the internal event e2 or between the creation of e2 and e3. (Actually, when event e1 is consumed several other events may already be waiting in the event queue.) We could prioritize internal events to be consumed before the next waiting or meanwhile arrived external event is considered. However, we consider this just as a special case of a more sophisticated event prioritization scheme. Such event prioritization schemes are future work. See the discussion of Definition A.64 and Definition A.66 and Definition A.82. So far, our sequential statechart semantics does not block external events while internal events are consumed. This means, while events are consumed and internal events are created and enqueued, other external events may occur and become enqueued, too.

Note, our approach of decoupling the handling of different system components into autonomous, independent processes also allows to deal with the well known compositionality problem of Harrel's statecharts. Recall, Harrel's statecharts supervise all system components parallelly on a single processing unit (or on a common global clock tick). Usually this is modeled by combining the statecharts that deal with the individual components into a common and-state that models the whole system. In Harrel's approach, all statecharts in such a common, global and-state share a common name space for events and events are broadcast into this name space. If two developers have created two substatecharts independently, they may have chosen some similar event names by accident. If such substatecharts are combined into a common and-state, each of the substatecharts receives the events created by the other substatechart. This may cause unexpected state transitions and corrupt the whole system behavior. [5]

In our approach this cannot happen. The substatecharts for the different system components are not combined into a common global and-state but they are turned into their own independent processes. Events are not broadcast between these different processes but all events are explicitly targeted to a

5. Note, there exist approaches that introduce separated name spaces for events for Harrel's statecharts, cf. the Focus / Autofocus project at TU Munich, Broy.

certain target. Thus, each substatechart retains its own name space for events. Two independently developed statecharts do not influence each other unless they send events to each other, explicitly.

Note, the compositionality problem of Harrel's statecharts are not only a serious issue for large scale systems that are developed in a team, but the compositionality problem is also a serious issue with respect to maintenance. In Harrel's statecharts, a new system component or a change to an existing component's substatechart may easily create "new" events that trigger transitions in other components the developer is not even aware of. In this way the addition or change of a small substatechart within an existing and well working system may corrupt all established behavior. Again, in our approach this maintenance problem does not occur due to the separated name spaces and since events are not broadcast to all processes. A new component or a component change can interfere other existing components only if someone provides it with a handle to the corresponding process and if an event is send via this handle by purpose.

Note, the problems with Harrel's statecharts semantics have inspired numerous other authors to develop alternative semantics for statecharts. These alternative semantics differ in the details of the event handling within micro steps and these alternative semantics differ in solutions for the nondeterminism problems of original statecharts. A thorough and detailed discussion of such approaches may be found in [Bee94]. Our approach drops the micro step semantics at all in order to avoid the "common-global-clock-tick" bottle neck. Instead, we employ a set of concurrent active objects that execute on their own pace. Event transportation and event processing may need time and meanwhile arriving events are queued and processed, sequentially. This system view is shared by SDL [ITU96] and ROOM [SGW94].

SDL is well accepted within engineering disciplines. It employs hierarchically nested processes with explicit communication channels. Process behavior is specified with so-called process diagrams. SDL process diagrams are some kind of finite automata. As in our approach, within SDL all processes run on their own pace, event transportation and event consumption may take time, and meanwhile arriving events are queued and processed, sequentially. SDL has a formal execution semantics and industrial strength tool support. However, for complex object oriented applications SDL has two major limitations. First, SDL process diagrams lack the expressive power of statechart elements like e.g. and-states, or-states, and history states. Second, SDL provide no object oriented data modeling concepts. Thus, complex behavior and complex object structures are not appropriately supported.

ROOM targets the same kind of complex object oriented applications as our approach. Like SDL, ROOM provides hierarchically nested processes (called actors) with explicit communication channels. Process behavior is specified with so-called ROOMcharts, a variant of statecharts. Processes run on their own pace, event transportation and event consumption may take time, and meanwhile arriving events are queued and processed, sequentially. The syntax of ROOM is well defined and implementation concepts and tool support are provided. Compared to SDL, ROOM is a large step forward in the direction of complex (concurrent) object oriented applications. ROOMcharts provide almost all high level statechart features. However, ROOMcharts do not provide the very expressive and-states. ROOMcharts resolve the nondeterminism problems of original statecharts by handling the alternatives in a fixed order. In contrast to our approach, this order is not determined by the user but by some execution mechanism. Which order is chosen can not be predicted. ROOM only guarantees that the chosen order is repeatable. ROOM provides object oriented data modeling concepts that allow to model complex object structures. However, in ROOM the actors have no shared memory. This means each process/actor employs his own private object structure and no other process/actor can access this private object structure. If one process needs information that is owned by some other process, events must be used for information or service exchange. Contrarily, our approach employs a common object structure within a shared memory. This allows that multiple process maintain a common system view.

In our factory example, all shuttles and robots share a common model of the factory layout. This common model allows one shuttle to locate the other shuttles and to take this information into account e.g. within its routing algorithm. In ROOM each shuttle would have its own private model of the factory layout. To be informed on the position of other shuttles, each shuttle movement needs to be communicated to all other shuttles. In ROOM this communication problem could be solved by introducing a central factory layout and shuttle position information component. However, this central information component would become responsible for all routing tasks. Therefore, this central information component may soon become a bottle neck. On the other hand, the ROOM approach avoids all kinds of concurrent object structure access problems. In our approach multiple concurrent processes access a shared memory. Thus, we need concurrency control mechanisms like semaphores in order to coordinate the concurrent data accesses. In addition, ROOM processes may be easily distributed on multiple machines. Our approach assumes a shared memory, thus we are merely bound to a single machine. This may create a scaling problem for our approach. Therefore, we plan to incorporate a ROOM like capsule concept in our approach that provides the definition of separated work spaces for different groups of actors. To summarize, our approach extends ROOM by providing full statecharts including the very expressive and-states. In addition, the sharing of a common complex object structure allows multiple processes that maintain a common system view and that may exchange complex information within their shared memory.

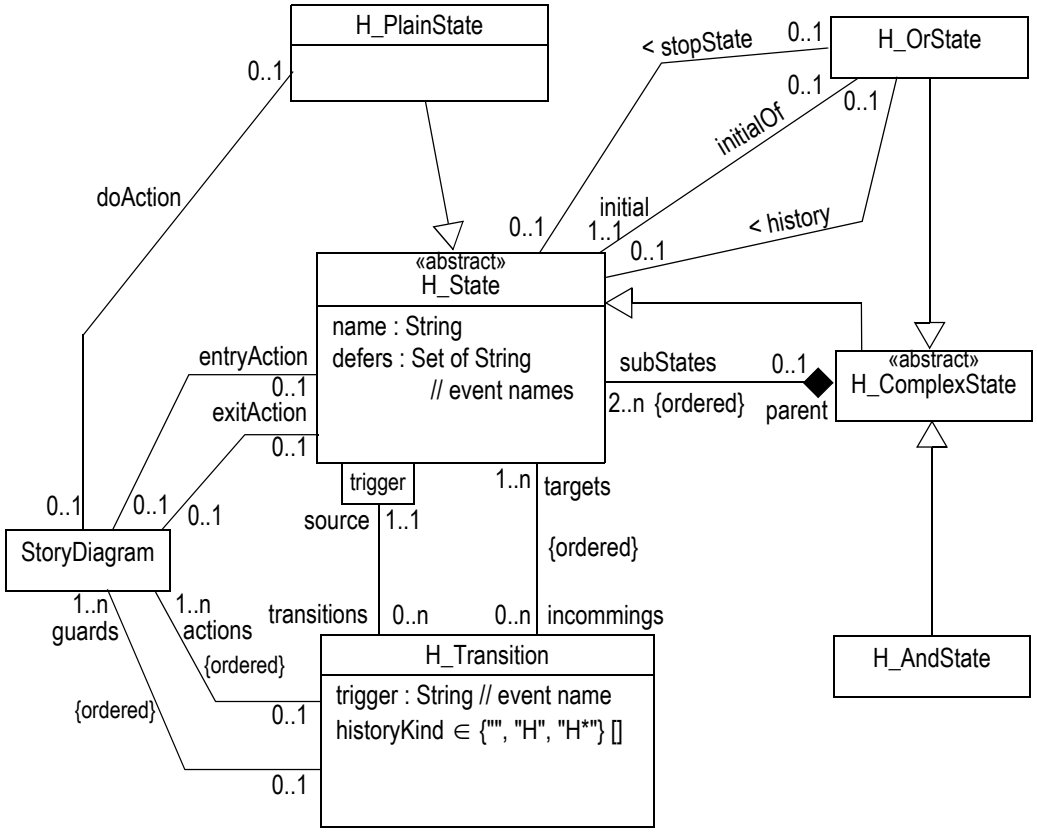
To summarize, our sequential statecharts semantics sequentializes the handling of events, deterministically. This solves several nondeterminism problems of Harrel's statecharts semantics. In addition it relieves us from executing data change operations in isolation to each other. This is achieved through user defined visiting orders for and-substates and guards. Note, the sequential handling of events by autonomous processes has the impact that multiple other things may occur between the creation of an event and its consumption. This is a significant difference compared to Harrel's micro step or super step semantics. In Harrel's semantics, all events are consumed directly within the next micro step or super step that follows the event creation. Combined with a common global clock tick, Harrel's semantics facilitate to guarantee certain system reaction times. Such reaction time guarantees are important for hard real time systems. Our approach focuses on large scale, distributed, object oriented systems that employ autonomous components with complex behavior (and object structures). This different application domain let to a different execution semantics for statecharts. Our sequential statecharts semantics decouples the system components and allows them to execute independently on their own pace. This enables the realization of complex behavior that may need some computation time. However, today's embedded and real time systems are becoming more complex, too. For such applications, the development of reaction time guarantees within our sequential statecharts semantics is current work, cf. chapter A.8.6.

A.8.3 Formalizing statecharts with story patterns

In order to integrate statecharts into our formalization of the UML, we first have to define what is a statechart:

Definition A.59: User defined Statecharts

A user defined statechart is a graph S that conforms to the following graph schema State-Schema (shown as a class diagram):



In addition, a statechart must fulfill the following constraints:

1) All states must have different names. (Obvious)
2) The substates relationship must form a tree of states covering all states in S. The root state must be an or-state without entry, exit, or do-action. No transition may enter or leave the root state.
   (Our meta-model would also allow disconnected groups of states. However, a meaningful statechart consists of a tree of states. We restrict the root state to have no actions itself, in order to facilitate the statechart initialization process.)
3) The depicted lower cardinality constraints for associations must hold.
   (As discussed in chapter A.2, in our approach class diagrams do not enforce lower cardinality constraints. Thus we have to require these properties, explicitly. This ensures that transitions have a source and a target and that complex states contain at least two substates.)
4) Transitions must provide exactly one guard and one action and one historyKind flag for each of their targets.
   (In our approach, multiple transitions with the same event trigger that leave the same state are modeled as a single transition with multiple targets. The required information allow to

handle all these targets uniquely. For transitions without an explicit guard, one has to employ the guard "true". The default history kind is "".)

5) A single state must not have multiple outgoing transitions with the same event trigger.
(As discussed above, we model such a situation using a single transition with multiple targets. This enables the derivation of the visiting order for guard expressions from the ordered targets association, cf. Definition A.74.)

6) A stop state must not have outgoing transitions.
(As soon as a stop state is reached, the corresponding substatechart has finished its execution. This may trigger some triggerless transition leaving the parent state. Since the current substatechart has terminated their should not exist a transition leaving the stop-state.)

7) Only complex states may serve as targets for transitions with history kind "H*" or "H". A single complex state may not serve as target for an "H*" and an "H" transition at the same time.
(For plain states a history mechanism makes no sense. We do not provide deep history and shallow history within the same complex state just because this would require two different history state markers within a complex state and this again could cause conflicting definitions of initial states for the first visit of that complex state.)

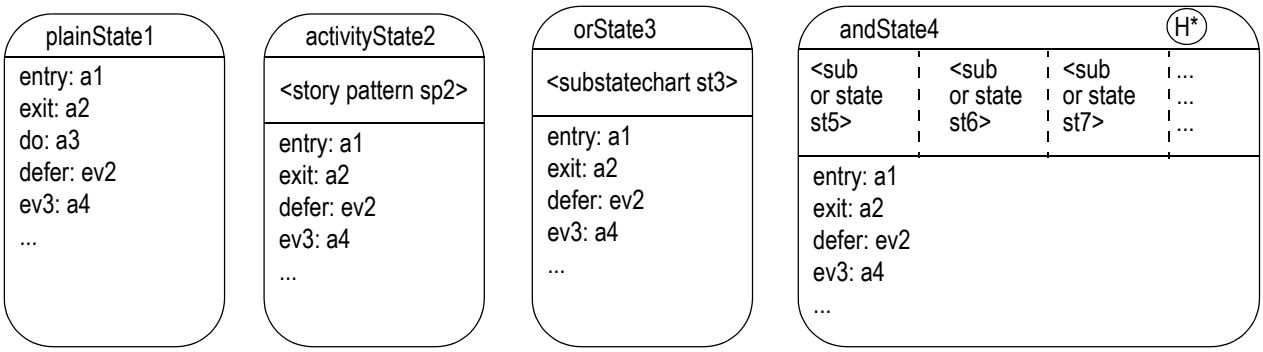
8) An H_AndState may contain H_OrStates, only. Each H_AndState must contain at least two H_OrStates. H_OrStates contained in an H_AndState must not contain entry, exit, nor do-actions nor defer clauses.
(For simplicity reasons, we model the and-substates by nested or-states and the and-state itself must not have actions attached to it.)

9) If a transition connects two (nested) substates x and y of an and-state z, then states x and y must belong to the same sub-or-states of that and-state z. A transition must not connect two (nested) substates that belong to two different sub-or-states of an and-state.
(Our execution mechanism requires that each and-substate is active, cf. chapter A.8.5. A transition from one and-substate to one of its sibling-substates would leave the source substate inactive. Note, such transitions are forbidden in other statecharts approaches, too. Note, a transition may leave a (nested) substate of an and-state and lead to some other state and vice versa transitions from other states may enter a (nested) substate within an and-state, cf. chapter A.8.5.)

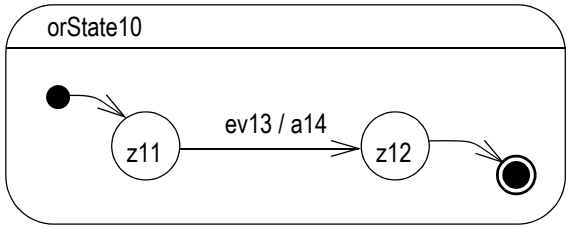
Note, our approach employs a relatively simple meta model for statecharts. Some of the context sensitive constraints could have been replaced by a more complex meta model. This design trade-off is similar to programming languages where certain constraints may be ensured either by the context free grammar or by context sensitive compiler checks. Our state chart model is relatively simple and more constraints are expressed in context sensitive rules. This complicates correctness checks, however, it facilitates the semantics specification.

Graphically, a statechart may be shown in the usual statechart notation, this means:

- The root state is not shown.
- States are shown using a rounded box with up to three compartments:



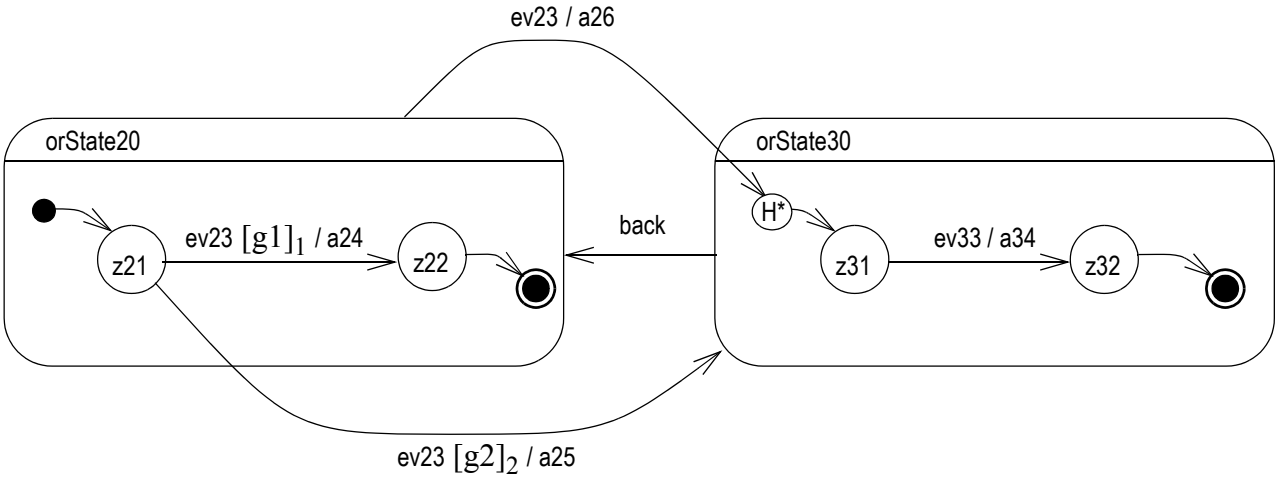
- The first / top compartments shows the state name.
- entry and exit and do-actions are shown in the optional bottom compartment using the corresponding key words. The entry and exit actions are shown as textual statements only.
- A do-action that calls a story diagram consisting of a single story pattern sp2, only, may be depicted using the middle compartment showing the story pattern sp2 directly.
- Defer clauses are shown one by one in the bottom compartment using the keyword defer.
- Or-states containing only a single substate with some self transitions like ev3 / a4 may be shown as plain states with so-called inner-transitions shown in the bottom compartment, e.g. ev3: a4.
- Within or-states, the initial state is marked by an additional filled circle with an arc leading to the initial state:



Note, the filled circle does not represent an additional state but the filled circle is regarded as a graphical marker for the state reached by the arc.

- Within or-states a stop-state is shown by marking it with a bulls eye and an arc leading from the stop-state to the bulls eye. Again, the bulls eye does not represent a state itself but it serves only as a graphical marker showing that the attached state is a stop-state.

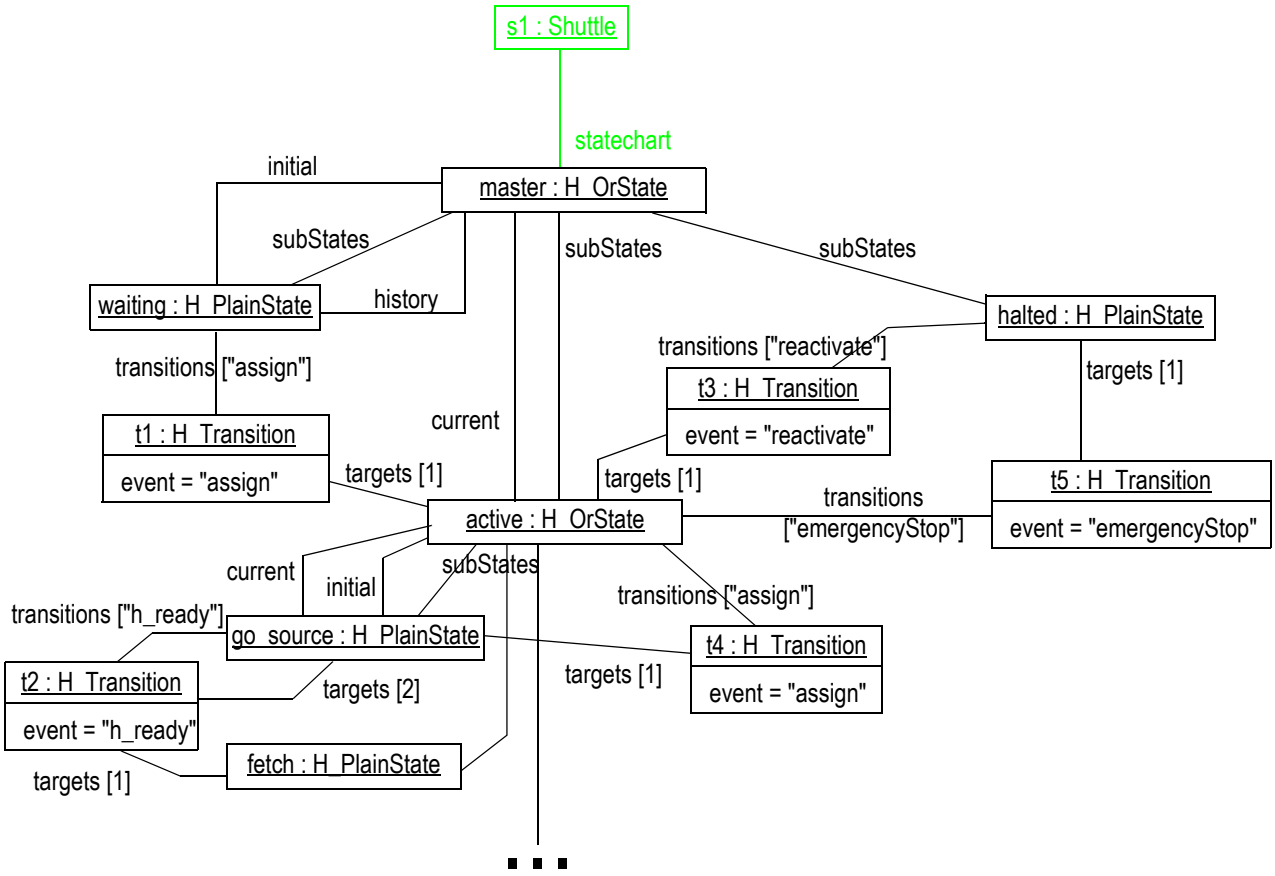
- Transitions are shown as arcs with stick arrow heads leading from their source state to their target state:



- The transition arc is labeled with the event name triggering the transition.
- If a transition has multiple targets, the transition is shown using multiple arcs, one per target. Each of these arcs is labeled with the event trigger followed by the guard condition shown in square brackets. Note, we employ an ordered association to store the sequence of targets for a transition. This ordering will be used to determine the order in which guards shall be evaluated and in which targets are considered. To show this ordering, the guard conditions may show subscribed index numbers. In case of only two targets, where the second target employs the guard $[true]_2$ we will omit the guard indices and show the second guard as [else].
  Note, the example above shows *two* transitions with the event trigger ev23. The first of these transitions leaves state z21 and has two targets: state z22 and state orState30, respectively. Therefore, we use two different arcs to depict this single transition. Each of these guards shows the triggering event name, the corresponding guard in square brackets with an index number indicating the ordering, and the corresponding action.
- The second ev23 transition leaves state orState20 and has state orState30 as its single target. However, this transition is a so-called *history* transition. Usually, if an or-state becomes the new current state, then the initial state contained within that or-State becomes active, too. Sometimes, one may want to re-activate the last active substate instead. In our graph representation of a statechart, each transition carries a historyKind flag for each of its targets. Transitions with an empty string as historyKind are drawn as an arc leading to the box of the targeted state, cf. transition ev23 $[g2]_2$ / a25 in the example above. If a transition with history kind "H*" or "H" targets an or-state then the initial substate of that or-state is not marked by an filled circle but by an hollow circle containing an H* or an H, respectively. In addition, the transition arc points to this marker instead of the border of the or-state. For and-states, such a history marker is provided within the name compartment of the and-state.
- And-states must contain at least two or-states. The middle compartment is divided into several areas by dashed lines, one area per contained or-state. Or-states contained in and-states have no graphical representation, themselves. Only the contained substates are shown within the subareas of the corresponding and-states.

Note, this graphical statechart representation may also be interpreted as usual UML object diagrams that employ special graphical stereotypes for the representation of statechart meta model instances.

Example 3.60: A snapshot of a statechart structure at runtime, cf. Example A.57



Having formalized the representation of statecharts, we are now ready to define their operational semantics. Therefore, we embed statecharts into the definitions of class diagrams and story diagrams:

Definition A.61: Reactive systems

An object oriented reactive system specification ROOSpec is defined by

ROOSpec := (OOSpec, Statecharts, EventNames, eventDecls, bind) where

OOSpec	is an object oriented specification, cf. Definition A.18
Statecharts	a set of statecharts, cf. Definition A.59
eventDecls	Func (NL) $\rightarrow$ P (EventNames)
bind	Func (NL) $\rightarrow$ Statecharts

In addition, ROOSpec must fulfill the following constraints:

1) eventDecls (C) != $\varnothing$ implies
   - bind (C) must be defined
   - the set of events employed in bind (C) must be equal to eventDecls (C)

Node labels / classes where eventDecls (C) != $\varnothing$ are called *active classes* or *thread classes*.

Nodes or objects which are instances of active classes are called *active objects* or *threads*.

2) For each node label C equipped with a statechart, OOSpec must provide a method h_start as described in Definition A.62 and a handleEvent method as described in Definition A.66.

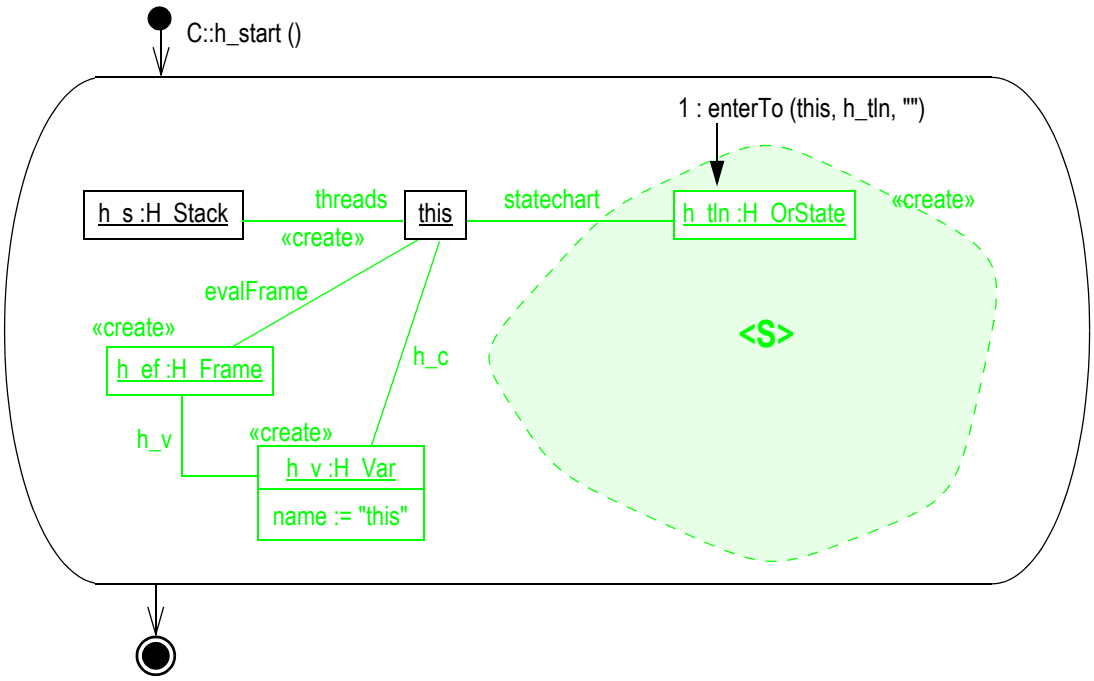
3) Any rule creating an active object x must send a collaboration message h_start () to the created object.

Sem [ROOSpec] := Sem [main; h_schedule*], where
h_schedule* is described in Definition A.65

Basically, a reactive system specification extends an object oriented specification by marking some classes as active classes. For each active class we provide one statechart specifying the reactive behavior of instances of that class. In addition, an active class provides a set of event declarations. The active class is able to receive exactly the event kinds it has declared within its interface. Accordingly, the statechart specifying the behavior of the active class may employ only events declared in the class interface. Conversely, the active class must not declare events in its interface that are ignored, i.e. not used, by its statechart.

Definition A.62: Initialization of active objects

The h_start method of an active class C described by statechart S is implemented by a story diagram:



where

h_tln is the root of the state hierarchy in S

the dashed shape represents the whole statechart S

We will define the behavior of a reactive system by providing an interpreter that schedules events and executes the event handling. As a prerequisite, Definition A.63 shows some help structures employed by our interpreter. Before we start with the definition of the statechart scheduler and interpreter we still have to explain the statechart initialization process in more detail.

Thus, the h_start method of an active class creates a copy of the statechart graph belonging to that class and a statechart edge attaching the active object this to the root of the state hierarchy. Note, as defined in constraint 2) of Definition A.61, the states must build a tree concerning the subStates asso-

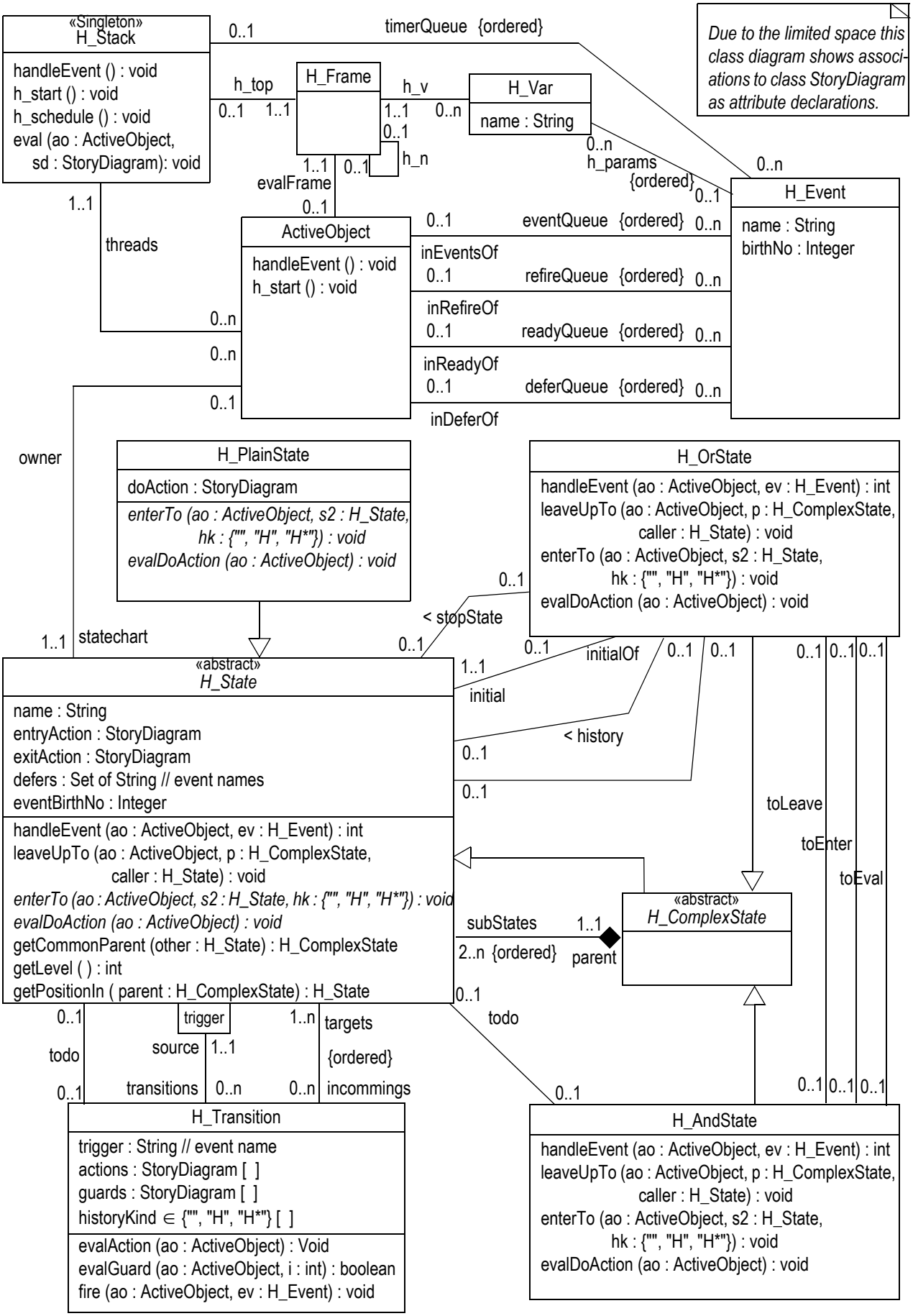
ciation. In addition, the active object this is added to the set of threads known to the unique H_Stack node. Finally, an H_Frame node is created for the active object that will serve as the root frame for the thread specific procedure call stack used to evaluate entry, exit, and do actions and actions attached to transitions.

Note, more formally, one could construct the story pattern employed by h_start by defining a pair of graphs (LG, RG). LG would contain the H_Stack node and the this node. RG would contain LG and the statechart graph S and the additional H_Frame and H_Var nodes and the edges depicted in Definition A.62. This rule would employ the this object as a bound variable. We leave this as an exercise for the interested reader.

Constraint 3) of Definition A.61 and the construction of the h_start method in Definition A.62 achieve that each active object employed in a reactive system is equipped with its own copy of the corresponding statechart graph. At run-time, the active object will store its current state(s) and the history states within this statechart graph using current and history edges, respectively. The active object will employ the methods declared in the class diagram of Definition A.63 to handle events and to change its states appropriately and to execute the corresponding actions. These methods are defined below.

Definition A.63: Help structures for the execution of statecharts

For the execution of statecharts we extend the graph schema / class diagram of Definition A.59 by some auxiliary methods and associations:

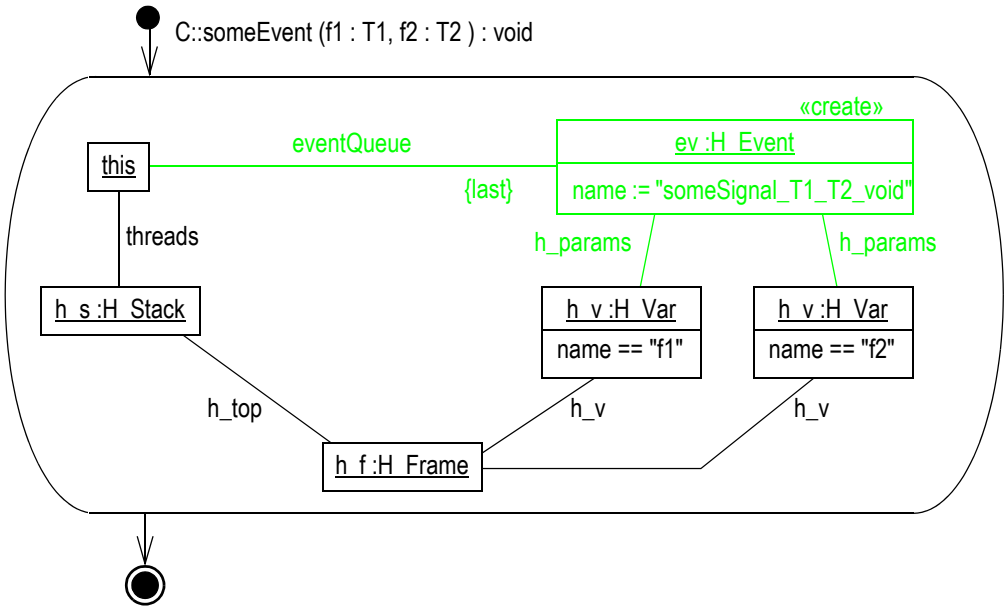


Before we define how an active object handles events, we first define how events are sent to active objects.

Definition A.64: Sending events to active objects via event methods

Let ROOSpec := (OOSpec, Statecharts, EventNames, eventDecls, bind) be a reactive system and let C be a node label in OOSpec.

For each event name ev ∈ eventDecls (C) where ev e.g. equals to someEvent_T1_T2_void the active class C implicitly provides an event method like follows:



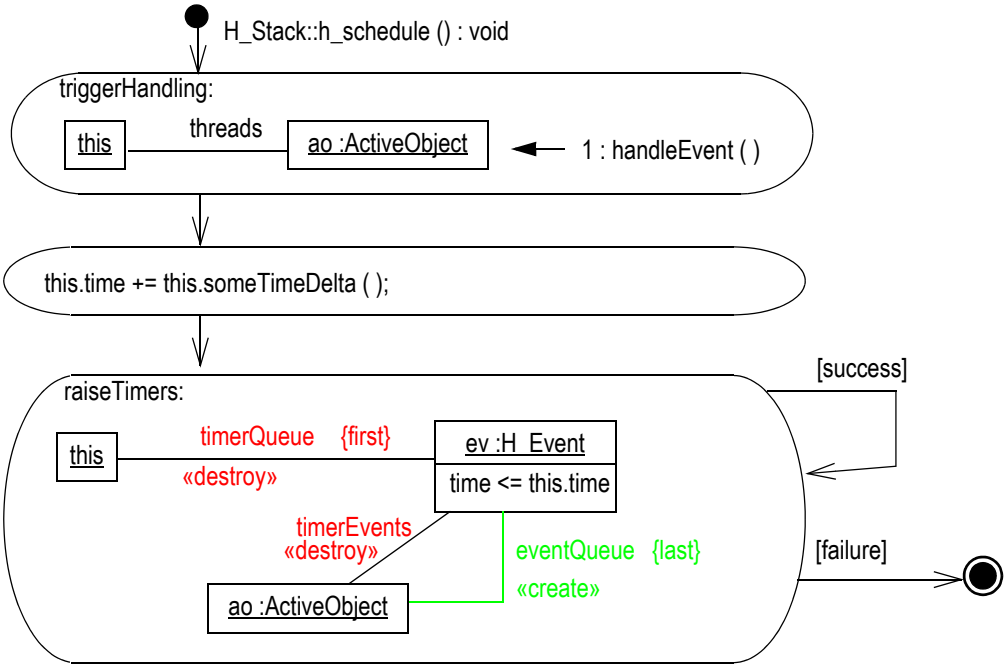
An event method just creates a new H_Event object and enqueues it to the event queue of the active object. If the event method has parameters, the corresponding H_Var objects are attached to the event object as parameters. After that, the event method returns. The event is waiting within the event queue of the active object until the scheduler triggers the active object to handle the event. Note, the eventQueue is organized as an ordered association, cf. Definition A.63. As indicated above, the event handling methods append new events at the end of the event queue. The sequence in which events are consumed will be discussed together with Definition A.66.

Next, we define the scheduler of the reactive system. The scheduler has the task to choose threads / active objects on a random basis and to trigger them to handle one event. Each such event handling step creates a new result graph which is part of the semantics of the reactive system:

Definition A.65: The scheduler for active objects

Sem [h_schedule*] is the set of all pairs of graphs that result from applying method h_schedule zero or more times.

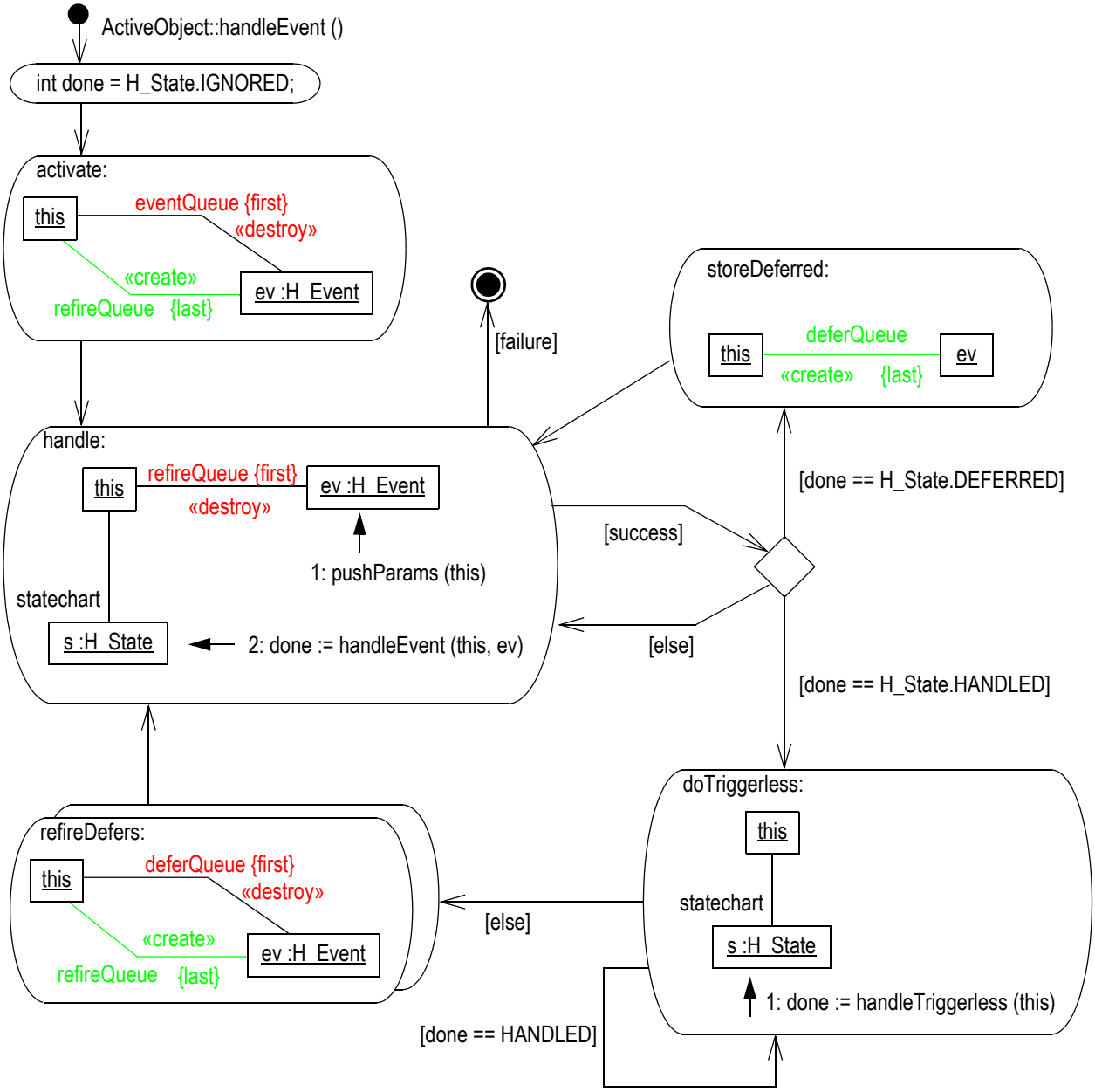
Method h_schedule is provided by our unique H_Stack object. It is defined as follows:



In addition we define that the choice of matches for story pattern triggerHandling shall be *fair*. This means, there must not exist an infinite series of applications of story pattern triggerHandling and an active object ao that is not matched by this series.

At the first shot we will discuss the first activity triggerHandling, only. The remaining of method h_schedule deals with after events and is discussed in chapter A.8.6. Activity triggerHandling looks-up an active object ao from the set of active objects attached to our unique H_Stack object and calls method handleEvent on that active object. Note, there may exist multiple active objects attached to the H_Stack and thus the story pattern of triggerHandling may have different match possibilities. According to our semantics definition for story patterns, each such match possibility may generate a different result graph. All these result graphs are part of the semantics of the triggerHandling step. The same active object may be chosen multiple times in a row. Some other active object may not be chosen for an arbitrary long time. The semantics definition of story patterns does not even guarantee that the scheduler is *fair*. The scheduler could choose the same active object each time. Using our semantics definition only, all possible series of applications would be part of the semantics of method h_schedule*, the fair ones and the unfair ones. However, to be able to model multiple active objects that collaborate by exchanging events we need a fair scheduler that does not ignore certain active objects infinitely long. Thus, we have restricted the series of allowed matches for story pattern triggerHandling, accordingly.

Definition A.66: handleEvent of active objects

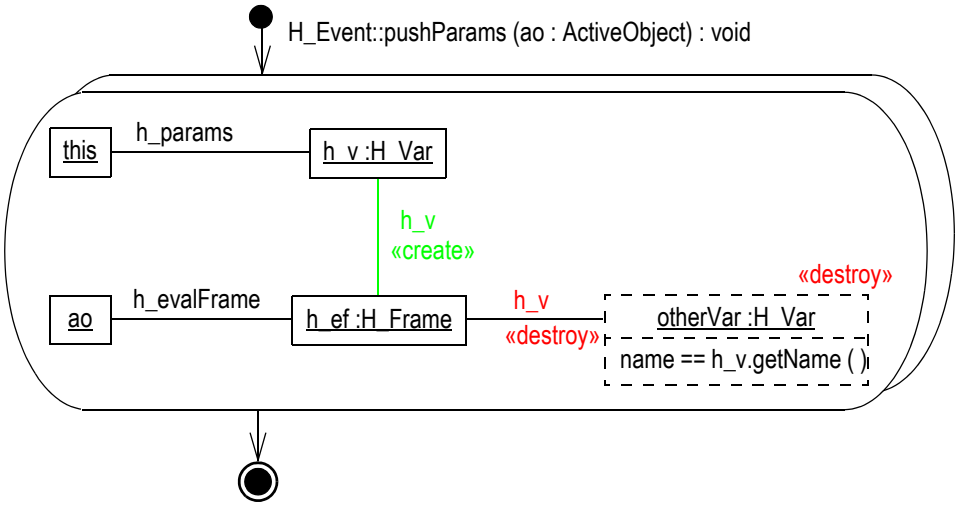


The scheduler chooses one active object and calls method handleEvent on it. The active object has now to look up its event queue, cf. Definition A.66, story pattern activate. The active object looks-up the *first* event ev within its event queue. Note, according to the class diagram of Definition A.63, the eventQueue association is ordered. As discussed for Definition A.64, in our approach new events are appended to the event queue. In addition, we consume events from the front of the event queue. This means events are handled in a first-in-first-out manner. This corresponds to the semantics proposed for SDL [ITU96]. Alternatively, we could choose events on a random basis. However, this would introduce an unnecessary source of nondeterminism that we want to avoid. Another, more interesting alternative is to equip events with priorities and to consume events on such a priority basis. One could also combine priorities and waiting time, i.e. the priority of waiting events could be increased, e.g. each time they are not chosen. Events with priorities could facilitate to guarantee "respond times" for certain critical functionalities e.g. for an emergency shutdown. However, we feel that event priorities create a lot of new properties for statecharts that need further investigation before we incorporate them into our semantics definition.

The main task of the handleEvent method of active objects is to deal with so-called *defer clauses*, cf. Definition A.71, and with *triggerless transitions*, cf. chapter A.8.6. We will discuss these topics later. So far, the main task of method handleEvent is to forward its call to the statechart attached to the active object. The statechart graph contains all informations about the current state(s), transitions that may fire, and actions that may need to be executed. Thus, we have assigned the main event handling functionality to the statechart graphs. To facilitate the handling of defer actions and triggerless transitions, story pattern activate of method ActiveObject::handleEvent does not directly handle the chosen first event but it just removes it from the event queue and stores it into the refire queue attached to the active object. Then method handleEvent proceeds to story pattern handle that actually handles the event. Story pattern handle takes the event from the refire queue and calls method pushParams on it. This method is described below. Next, story pattern handle looks-up the statechart s attached to the active object and just forwards the handleEvent call to the statechart s. The active object itself and the event ev are passed as parameter.

As already described in Definition A.64, events may be equipped with H_Var nodes, holding the values of parameters of the corresponding event method. Generally, a statechart may employ a set of local variables and parameters of different events. These local variables may be used within entry, exit, do-, and transition actions. In order to provide access to these local variables, the active object provides its own procedure call frame h_ef, cf. Definition A.62. This personal procedure call frame serves as the bottom frame for the execution of all kinds of actions attached to the statechart of the active object, cf. Definition A.75. The task of method pushParams of class H_Event is just to put the H_Var nodes attached to the event to the procedure call frame h_ef of the corresponding active object ao:

Definition A.67: Pushing the parameters of events to the event frame of active objects



Note, a procedure call frame must not contain two variables with the same name. If a certain object receives an event with parameters the second time then the procedure call frame will already contain H_Var objects for the corresponding parameters. Thus, method pushParams employs an optional node otherVar which tries to match an already existing H_Var node with the name of the H_Var node h_v, which is going to be inserted. If otherVar matches such an old variable, the old variable is just removed. In each case, after the execution of pushParams the procedure call frame holds exactly one H_Var node for each parameter of the event.

Note, different event methods could use formal parameters with the same names. In our definition these different formal parameters would result in the same H_Var node in the procedure call frame. This is not a bug but a feature. We will discuss this topic in more detail together with Definition A.75.

A statecharts handles an event through the steps described below. Note, we explain the event handling for statecharts with plain and or-states first, statecharts with and-states will be explained in chapter A.8.5. Thus, so far, our statechart may not contain an and-state. The general event handling steps of a statechart are:

1. Method handleEvent for states is called on the top level state. First it climbs down the hierarchy of nested complex states along the current state markers.
2. From the bottom current state we search upwards for a leaving transition with the appropriate event trigger.
3. The leaving transition may have multiple guards and targets. We evaluate the guards in order to determine the corresponding transition target.
4. Compute the closest common parent state of the source and target state of the firing transition
5. Leave the current states from the bottom upwards until the least common parent state is reached. Thereby, execute the corresponding exit actions and mark the states you are leaving with history edges.
6. Enter into the hierarchy of complex-states that contain the target state. Start at the least common parent state and proceed downwards towards the target state. If the target state itself contains nested complex states proceed downwards into these states employing initial and history state information, appropriately. Thereby execute the entry actions top-down.
7. Finally, execute the do-actions of the just entered states in top-down order. In addition, handle triggerless transitions.

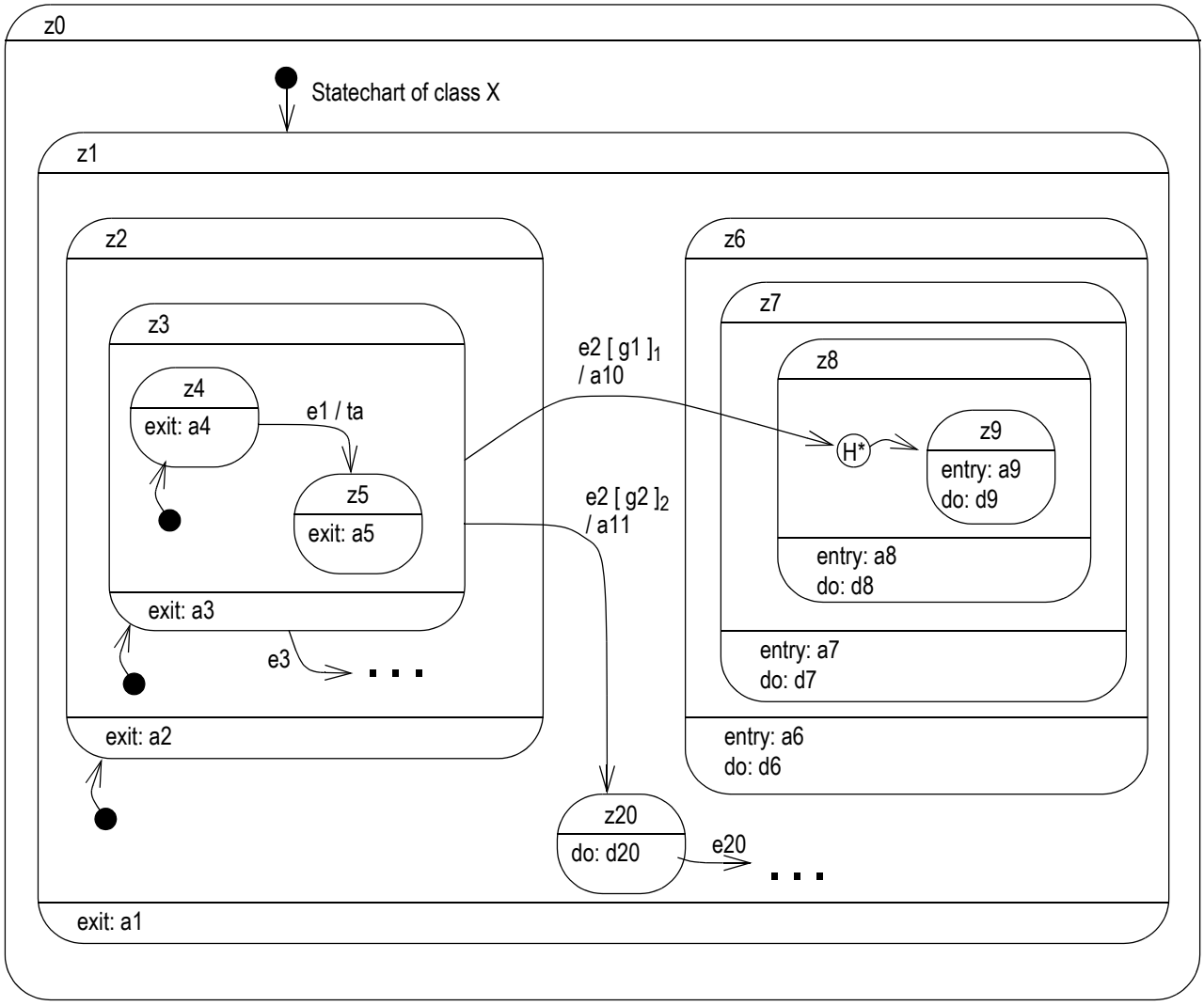
In the following we will explain these steps in detail and define the corresponding operations formally. Example A.68 will serve as the running example for these explanations.

As explained above, step 1 of handling an event is to climb down to the lowest current state. Therefore, story pattern trySubState of method handleEvent of class H_OrState just looks-up the current substate s1 and forwards the handleEvent call, recursively, cf. Definition A.69. Note, in our model each active H_OrState object provides exactly one edge of type current marking the currently active substate.

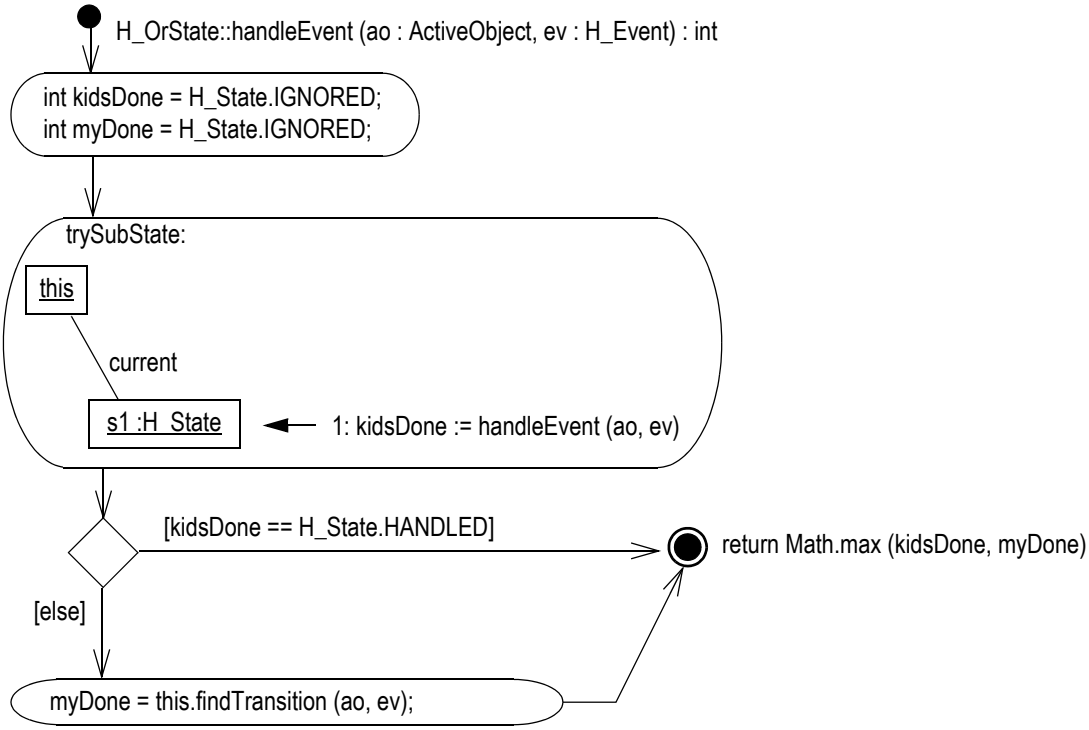
Once a plain state is reached method handleEvent of class H_State applies. Plain states just check whether they have an appropriate outgoing transition. This is done by calling method findTransition, cf. Definition A.73. In story pattern findTrans method findTransition just tries to find an outgoing transition t with an appropriate event trigger. If such a transition t exists, we call method fire on it. Method fire executes the remaining steps of the event handling. This will be described below. In addition, the flag variable done is set to value HANDLED indicating, that an appropriate transition has been found and that we do not need to examine parent states for appropriate transitions, anymore.

If story pattern findTrans fails, we proceed with story pattern findDefer. Story pattern findDefer checks whether an appropriate defer clause exists at the current state. In that case, the return value done is set to DEFERRED. The remaining handling of deferred events is described below, in Definition A.71.

Example A.68: Example statechart with complex nesting

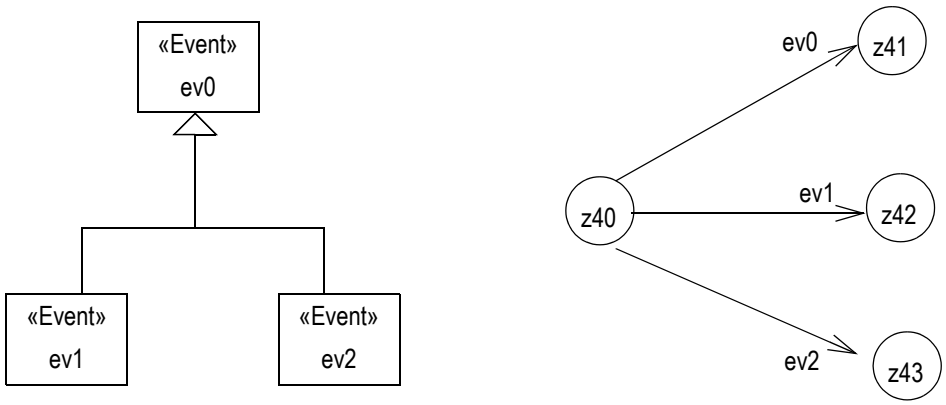


Definition A.69: handleEvents for or-states



Note, story pattern findTrans searches for transitions with exactly matching triggers, only. Our approach does not employ an hierarchy of event kinds as proposed e.g. in [BRJ99]. Hierarchically organized event kinds allow to declare some events ev1 and ev2 as subkinds of a parent event kind ev0. In such approaches, transitions labeled with trigger ev0 are fired by subkind events like ev1 and ev2, too. This allows to deal with groups of event kinds in a single transition. So far, our approach does not provide event kind hierarchies. In order to introduce event kind hierarchies we would have to provide language means for the definition of such hierarchies. In addition, story pattern findTrans would have to check wether event ev is a subkind of the event trigger attached to leaving transitions. However, introducing event kind hierarchies may cause semantic ambiguities that need to be defined with care. In Example A.70, it is not clear which transition fires if z40 is the current state and ev1 occurs. We could reach state z42 because ev1 matches the corresponding transition exactly or we could reach state z41 because ev1 is a subkind of event ev0. In this situation one could either define that the more specialized transition is preferred or that the user has to provide guards determining the resulting state. Another question that needs to be discussed is whether the event hierarchy should allow multiple inheritance. We think that these questions need further investigations and therefore this work does not yet support event kind hierarchies.

Example A.70: Ambiguous transitions caused by hierarchical transitions



In Definition A.73, story pattern findDefer of method findTransition checks whether the current state contains a defer clause for event ev. In this case, the done flag is set to DEFERRED. Note, the result of method findTransition is propagated upwards within the state hierarchy. Method H_OrState::handleEvent collects the results of the recursive call to handleEvent and of the call to findTransition on itself, cf. Definition A.69. If the event has been handled it is consumed and method H_OrState::handleEvent returns, directly. If the event has not yet been consumed, method findTransition is called on the or-state, itself. This will look for outgoing transitions or defer clauses on this higher nesting level. Altogether, method handleEvent of or-states returns the maximum value of the recursive call to handleEvent in story pattern trySubState and of the local call to findTransition. Thus, in method ActiveObject::handleEvent the call of handleEvent on the associated statechart may return either HANDLED or DEFERRED or IGNORED, cf. Definition A.66. If the event has been DEFERRED it is handled according to the following definition:

Definition A.71: Deferred events

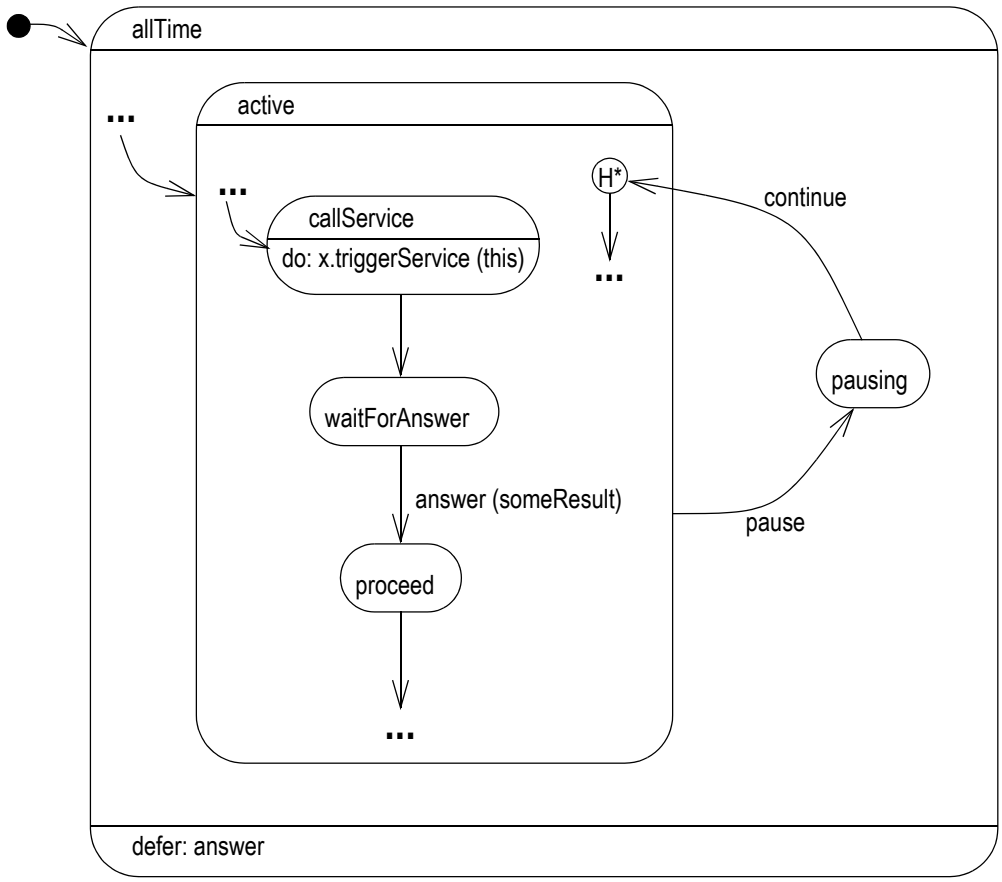
> A defer clause causes the corresponding event to be deferred, if no transition exists, that consumes the event. The event is deferred until a new state is reached and then it is raised again. Deferred events are considered before the next external event is consumed.
>
> More precisely, if an appropriate defer clause exists and if no transition has fired, then the call of handleEvent in story pattern handle of Definition A.66 returns value DEFERRED. This causes the execution of story pattern storeDeferred. Story pattern storeDeferred

stores the event within the deferQueue of the active object. Each time an event is actually handled by an active object, story pattern refireDefers of method handleEvent for active objects moves all events waiting in the deferQueue into the refire queue, cf. Definition A.66. Note, story pattern handle of method handleEvent for active objects is the head of a loop that consumes events from the refire queue until it drains.

Thus, each time an event is actually handled (i.e. done == HANDLED) all events waiting in the defer queue are reconsidered. The events are appended to the refire queue like new events. Consuming the events via story pattern handle may cause that the event is ignored at all, that it fires some transition, or that it is deferred again.

Deferred events may be used in order to guarantee that certain events are not ignored but that they are maintained until they fire a transition, cf. Example A.72. In Example A.72 the do-action of state callService calls some event method triggerService () on some active object x. The active object x runs its own statechart and may need some time to compute a result. Active object x is supposed to send an answer event back to the initiating object. Thus, our example statechart switches to state waitForAnswer where it waits for an answer event. As soon as the answer event arrives, the statechart switches to state proceed and proceeds with its computations. This kind of protocol where one statechart sends an event and then waits for an answer occurs quite frequently.

Example A.72: Using defer clauses to prevent event losses.

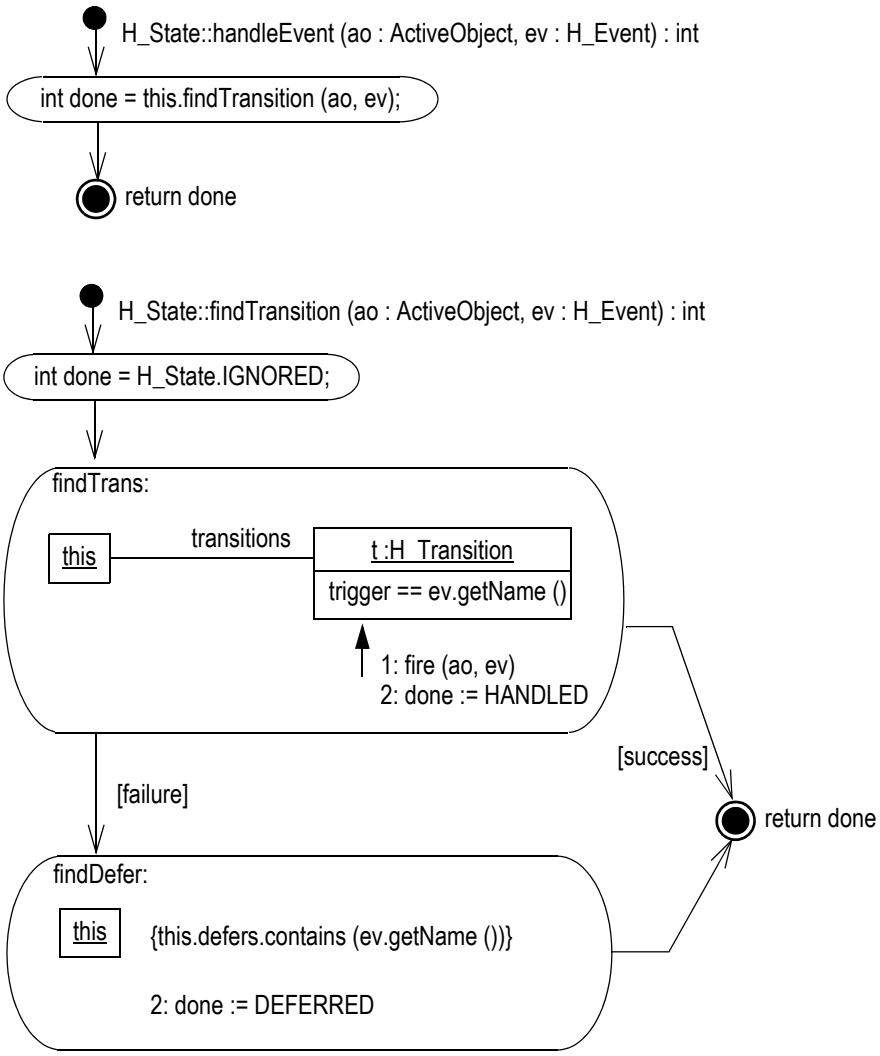


A problem arises if the initiating object needs to be able to handle certain interrupts while it is waiting for the result. Such an interrupt handling may either be modeled using a parallel and-state or like in our example using a transition leaving the surrounding or-state and another transition returning to the history state. If we leave the or-state and want to return later via a history state in order to continue the normal process, the following situation can occur. Our example object may initiate a triggerService call and switch to state waitForAnswer. Next, it could receive a pause event causing it to switch

to state pausing. While it is in state pausing the answer event may arrive. If the statechart now switches back to state active via a continue event, the answer event must not be lost. Otherwise, the statechart would probably stay within state waitForAnswer for ever, because it missed the answer event and no one is going to send a new answer event since answer events are send as response to triggerService calls, only.

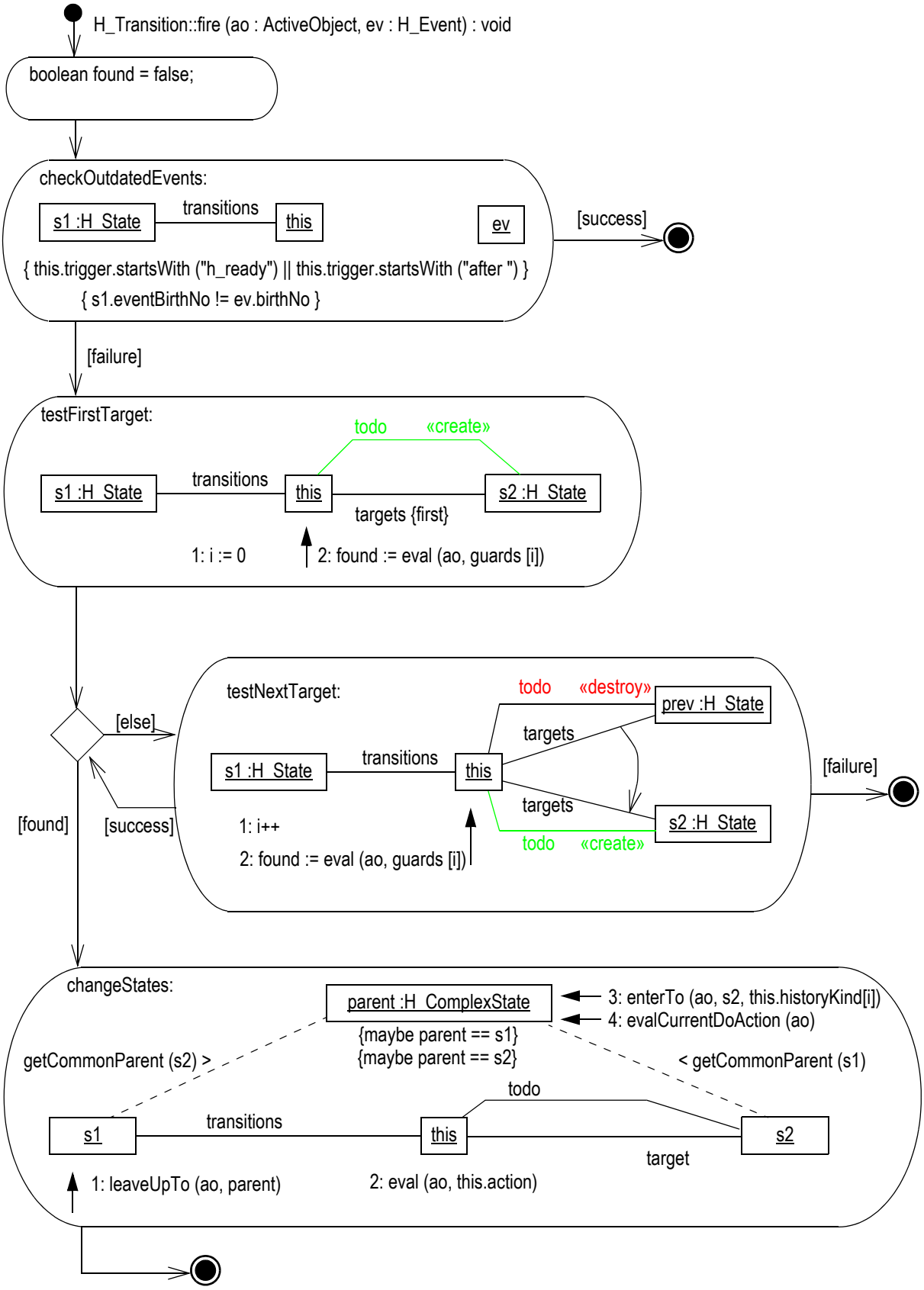
In order to guarantee, that the answer event is not missed while we are in state pausing, we employ a defer clause in state allTime. If the answer event arrives while the statechart is in state pausing, the answer event is stored in the defer queue. If a continue event arrives and the statechart switches back to state waitForAnswer, then the answer event is refired and state active proceeds with its computations as desired.

Definition A.73: handleEvent for elementary states



If the story pattern findTrans of method findTransition detects an appropriate transition, that transition is fired, cf. Definition A.73. This means, we call method fire on the corresponding transition. Definition A.74 shows the specification of method H_Transition::fire. The first story pattern checkOutdatedEvents deals with internal events and will be discussed in chapters A.8.4 and A.8.6. Story pattern testFirstTarget looks-up the first target s2 of the current transition and marks it with a todo edge. In addition, variable i is initialized and the first guard attached to the transition is evaluated by calling method eval passing the guard as parameter. Method eval will be discussed in Definition A.75. The result of the guard evaluation is stored in variable found.

Definition A.74: Firing a transition



Note, as already discussed we store the targets of a transition using an ordered association. If necessary, the guards show the order in which they are considered by subscribed indices. The user may change this order. This allows the user to define the sequence in which the targets and guards are to be considered.

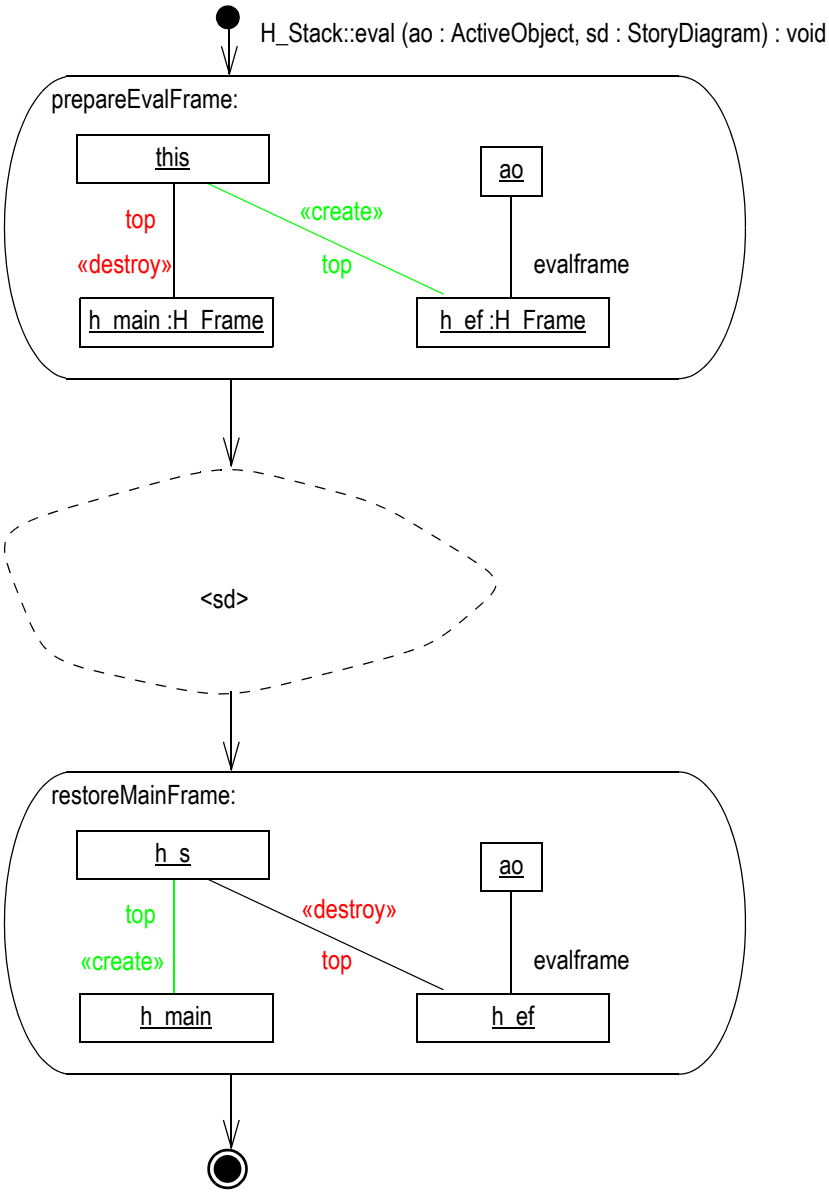
If found has become true, a proper target has been found and we switch to story pattern changeStates, discussed below. Otherwise, we switch to story pattern testNextTarget. Story pattern testNextTarget looks-up the subsequent target state s2 and redirects the todo edge, accordingly. In addition, variable i is incremented to point to the next guard and the next guard is evaluated. Story pattern testNextTarget might fail as soon as the transition runs out of new targets. In this case no guard has evaluated to true, the transition does not fire but the event is just ignored. Thus, in case of failure method fire just terminates. If a new potential target exists and a new guard has been evaluated, we switch back to the branch state checking whether the guard has returned true. Altogether, this considers each target in the given order until we run out of targets or until a guard evaluates to true. At this point we have executed step 3 of our event handling process.

Before we continue with the next event handling step, we discuss the execution of guards and of entry, exit, do-, and transition actions:

Definition A.75: Evaluating guards, entry, exit, do-, and transition actions

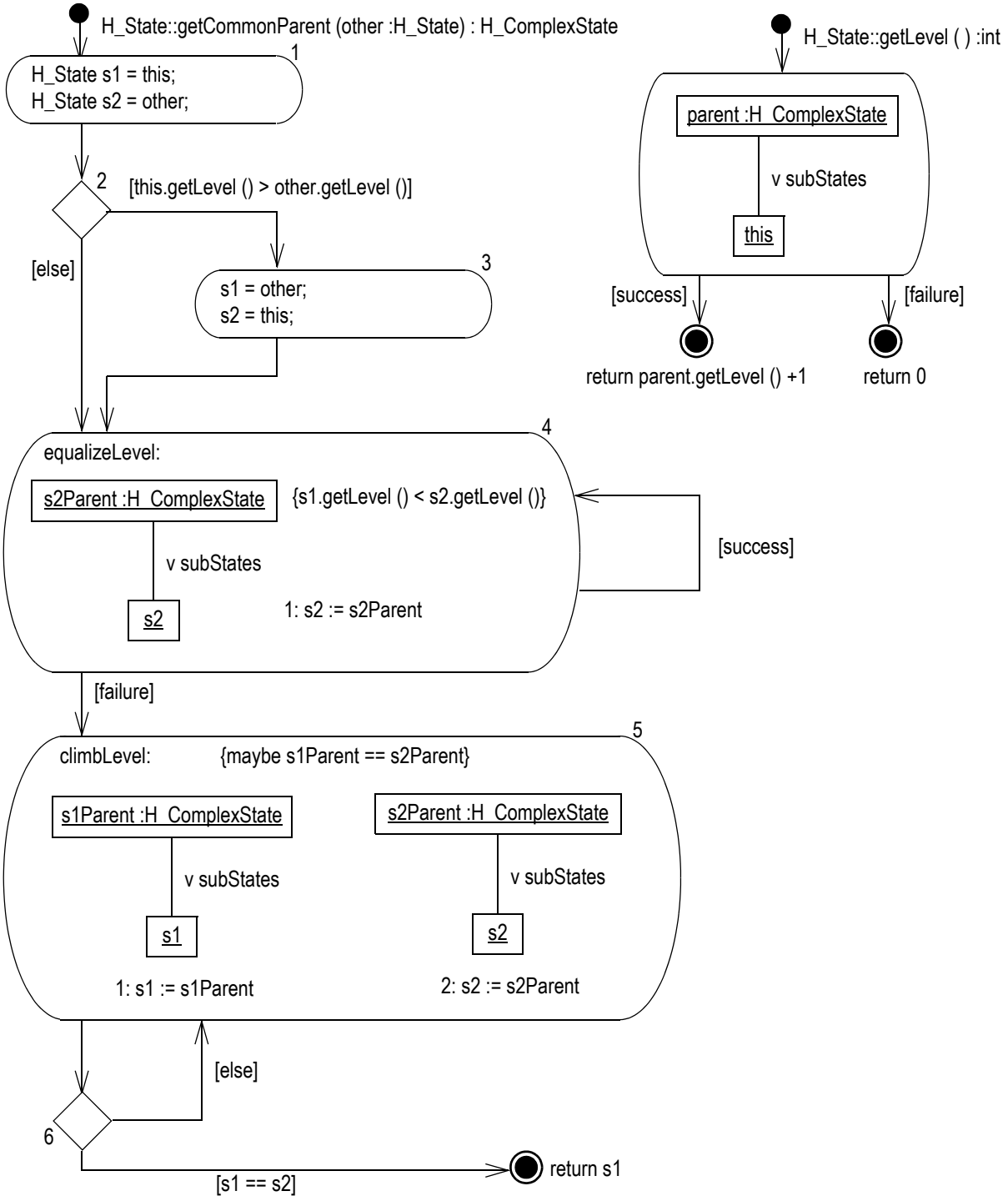
The evaluation method eval is a meta operation, that takes an active object ao and a story diagram sd as parameter. Method eval prepares the execution environment for the active object, i.e. its personal procedure call stack and then it executes the story diagram sd and then it restores the main procedure call stack. Thus,

Sem [eval (ao, sd)] := the semantics of the following story diagram



Now we proceed with the handling of events in story pattern changeStates of Definition A.74. Story pattern changeStates handles the state changes according to the determined transition. Therefore, story pattern changeStates first has to compute the least common parent state of state s1, which is to be left, and state s2, which is to be entered:

Definition A.76: Computing the least common parent of two states



Simple transitions may connect two states within the same or-state, cf. transition e1 leaving state z4 and entering state z5 in Example A.68. In this simple case only the exit action of z4 and the transition action ta and the entry and do actions of z5 need to be executed. However, in Example A.68 transition e2 leaves the or-state z3 and crosses the boundary of or-state z2 and enters either or-state z8 or the plain state z20. Or-state z8 is again nested into the or-states z6 and z7. Other transitions might leave a complex state and target one of its substates or a transition might leave a substate and target the sur-

rounding complex state. For all these cases the least common parent state containing both the source and the target state of a firing transition serves as an anchor point for the state changes.

The least common parent state is computed with the help of method getLevel, cf. upper right corner of Definition A.76. Method getLevel just counts the nesting depth of complex states, recursively. The nesting depth of a given state computes to the nesting depth of its parent state plus 1. If no parent state exists, the nesting depth is 0, cf. the failure return of the getLevel story diagram.

Activities 1, 2, and 3 of method getCommonParent of class H_State assign the state with the lower nesting depth to variable s1 and the state with the higher nesting depth to variable s2. Story pattern equalizeLevel looks-up the parent of state s2 and assigns the result to variable s2 as long as the nesting depth of s2 is higher than the nesting depth of s1, cf. the corresponding constraint.

Next, story pattern climbLevel is executed. Story pattern climbLevel looks-up the parents of state s1 and s2 and assigns the results to these variables, respectively. Note, at some stage, s1 and s2 will have the same parent. Thus, we use a maybe clause in order to allow that s1Parent and s2Parent may match the same object. Story pattern climbLevel is executed until the common parent is found. Note, we require that the states of a statechart graph form a tree. Thus, if variables s1 and s2 refer to objects within the same tree and at the same nesting depth, then climbing up from both variables with the same pace will finally find the common root of s1 and s2. Climbing up will terminate at the root of the whole statechart, at latest.

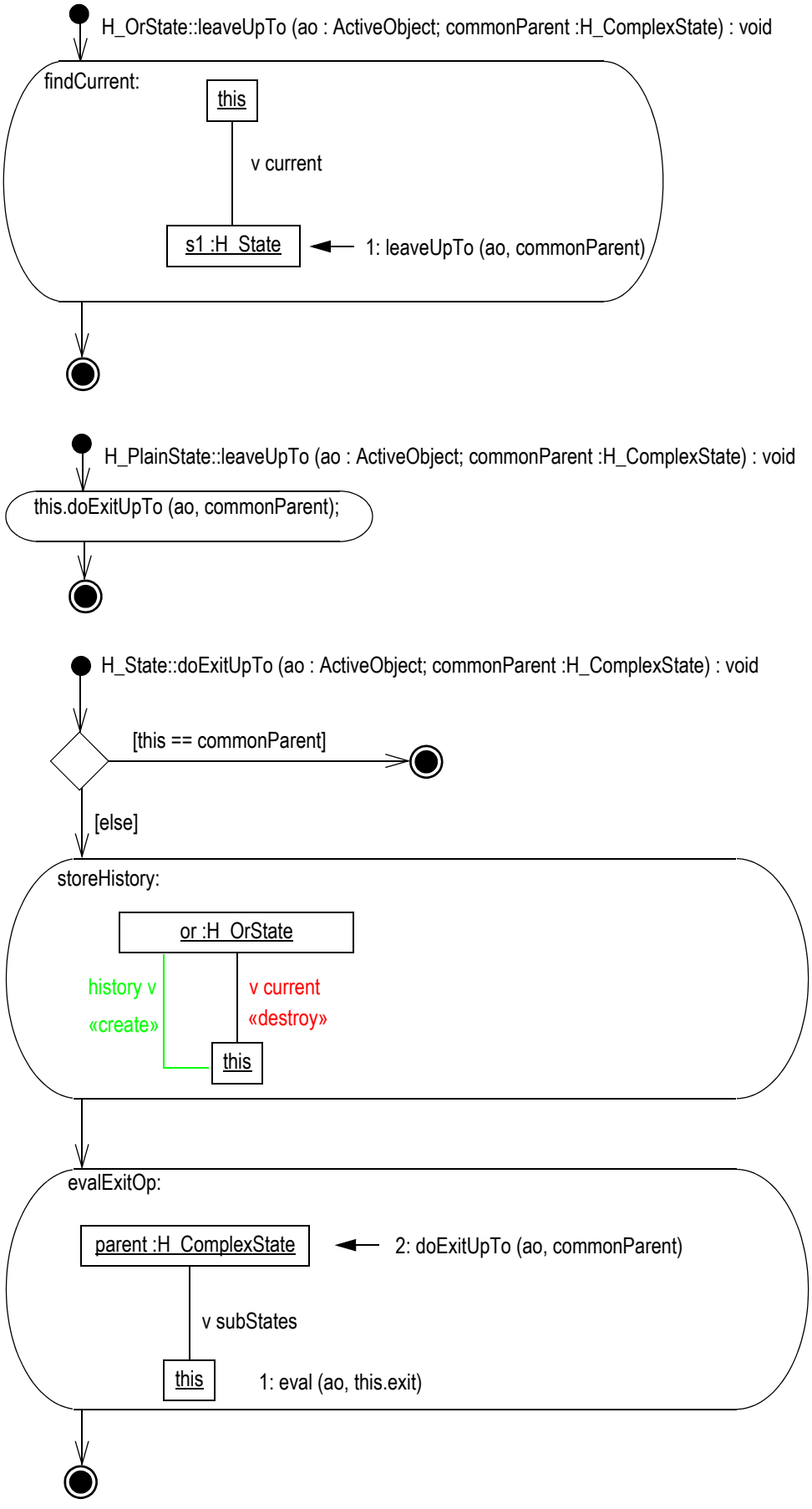
If s1 equals to s2, we have found the common parent. Note, if the source or target state of the firing transition is the parent of the other connected state or if we consider a self transition then s1 and s2 are already equal when we leave story pattern equalizeLevel. Formally, all these cases are handled like a self transition. This means, the corresponding state(s) are left and re-entered and thus, the corresponding exit and entry actions must be executed. To achieve this behavior, we execute story pattern climbLevel at least one time. Thus, in case of such a "self" transition, we compute the parent of the higher state as the common parent of s1 and s2.

Let us assume that story pattern testFirstTarget of Definition A.74 has detected that the e2 transition from state z3 of Example A.68 to the history marker of state z8 fires. In this case method getCommonParent will compute state z1 as the least common parent of states z3 and z8. State z1 is not affected by the state changes.

In story pattern changeStates of Definition A.74, method getCommonParent is employed within two navigation expressions that determine the match for variable parent. Note, one of the navigational expressions would have sufficed. The other has been added for symmetry reasons, only. In case of a transition between a complex state and one of its substates, the match for variable parent will be equal to variable s1 or s2. To allow such matches story pattern changeStates employs appropriate maybe clauses.

Now we are ready to leave the current states below the common parent state. In story pattern changeStates this is done by calling method leaveUpTo on state s1, passing the active object ao and the common parent state as parameters. Method leaveUpTo is shown in Definition A.77.

Definition A.77: Leaving current states



When leaving nested states, we have to execute the exit operations of the states starting at the leave state. In addition, we remove the old current links and replace them by history links. The latter are used to deal with history states which will be discussed later. However, while searching for a firing transition during phase 2 of our event execution, we may have climbed up several nesting levels. In our Example A.68 variable s1 would refer to state z3 if the e2 transition leading to the history marker fires. Thus, in story pattern changeStates variable s1 may refer to a complex state on a higher level. Therefore, the first task of method leaveUpTo is to climb down to the current leave state, again. To achieve this, story pattern findCurrent of method leaveUpTo of class H_OrState looks-up the current substate s1 and calls method leaveUpTo, recursively. As soon as we reach a plain state, method doExitUpTo is called, cf. the second story diagram of Definition A.77.

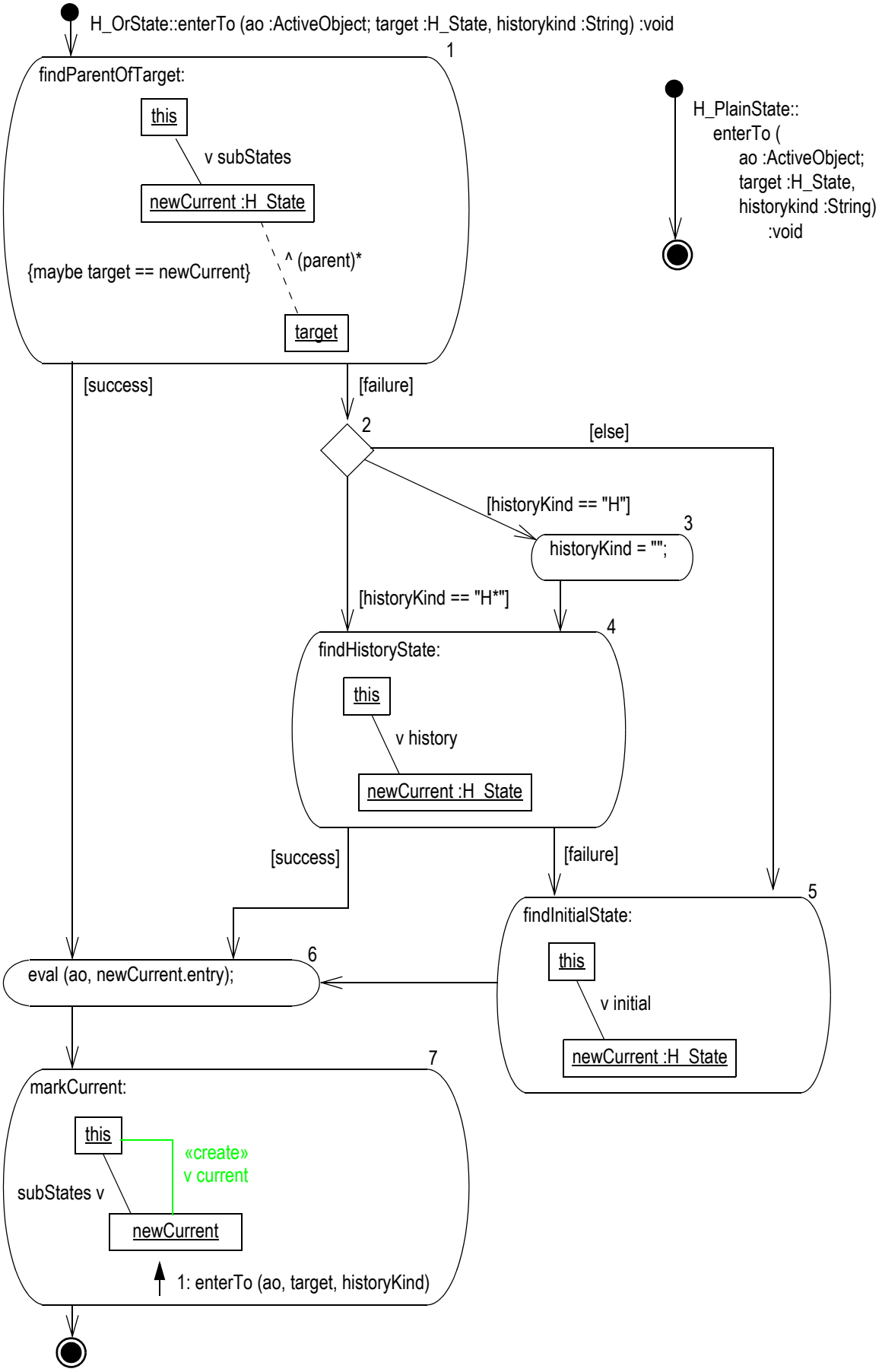
Method doExitUpTo now climbs up the nesting levels until the common parent state is reached. If the common parent state is not yet reached story pattern storeHistory is executed. Story pattern storeHistory looks-up the parent or-state[6]. It removes the old current link and replaces it by a history link, as required. Next, story pattern evalExitOp is executed. Within the first collaboration message, story pattern evalExitOp evaluates the exit action of the current substate and within the second collaboration message it calls method doExitUpTo on the parent state, recursively. Note, this recursive call will climb up the nested states until the common parent is reached, cf. the first branch. If the common parent is reached, method doExitUpTo terminates immediately. This means, the exit action of the common parent is not executed. This achieves the required behavior. Altogether, this completes step 5 of our event handling strategy.

Step 6 of our event handling strategy is to enter the target state hierarchy. We start at the common parent state and descent into the hierarchy of nested target states. The target states are derived from the target of the firing transition. In Example A.68, transition e2 enters the history marker of state z8. In our graph model this means, transition e2 targets state z8 and carries a "H*" marker identifying it as a deep history transition. Thus, we first have to enter the nested states z6, z7, and z8 in that order. If the target state of the firing transition is itself a complex state (as in our case), we enter either the history or the initial substates of the target state. This is repeated recursively until leave states are reached. In case of a history or deep history transition like in our example, we enter the state that is denoted by a history link, stemming from earlier state changes. If no history link exists, we enter the initial state, e.g. state z9. This behavior is specified in detail by method enterTo shown in Definition A.78.

Method enterTo is called in collaboration message 4 of story pattern changeStates of Definition A.74. There it is called on variable parent, the common parent of source and target state of the firing transition. In Definition A.78, the first story pattern findParentOfTarget is successful if the executing state this is a direct or indirect parent of state target, the target of the firing transition. Note, the match for variable newCurrent has to fulfill two constraints. First, there must exist a subStates link coming from this. Second, the navigation expression (parent)* computes the set of all direct or indirect parents of state target. The match of variable newCurrent must be contained in this parent set. Note, the content of variable target is included in the result of (parent)* since the transitive closure includes zero-time traversal of the parent link. To allow newCurrent to match the same node as variable target, we employ an appropriate maybe clause. To summarize, story pattern findParentOfTarget succeeds, if the process of entering states starting at the common parent state has not yet reached the target of the firing transition. In that case, we choose the substate of the current state that contains the target state. That substate is represented by variable newCurrent.

6. Only or-states employ current markers. For and-states, all substates are active and no current marker is needed.

Definition A.78: Entering target states



Thus, on success of story pattern findParentOfTarget we reach activity 6, which evaluates the entry-action of the new current state. Next, story pattern markCurrent creates a current link marking

newCurrent as the new current substate of state this. Finally, collaboration message 1 of story pattern markCurrent calls method enterTo, recursively.

If story pattern findParentOfTarget fails, the process of entering states has reached or passed the target of the firing transition. In this case we either enter the history substate or the initial substate. This is decided with the help of the historyKind attribute of the firing transition which is passed as parameter to method enterTo, cf. story pattern changeStates in Definition A.74. If the historyKind equals to "H" or "H*" then we want to enter the history state. This is achieved by story pattern findHistoryState. Story pattern findHistoryState looks-up a history link leaving this and tries to bind variable newCurrent to the target of this history link. Note, history links are created when we leave an or-state. If we enter the or-state the first time, then a history link will not yet exist and story pattern findHistoryState will fail. In that case we proceed with story pattern findInitialState.

Note, in case of a deep history transition the historyKind parameter is forwarded to the recursive call of enterTo within story pattern markCurrent. In case of a shallow history transition, activity 3 resets the historyKind parameter and the recursive call will execute story pattern findInitialState, directly.

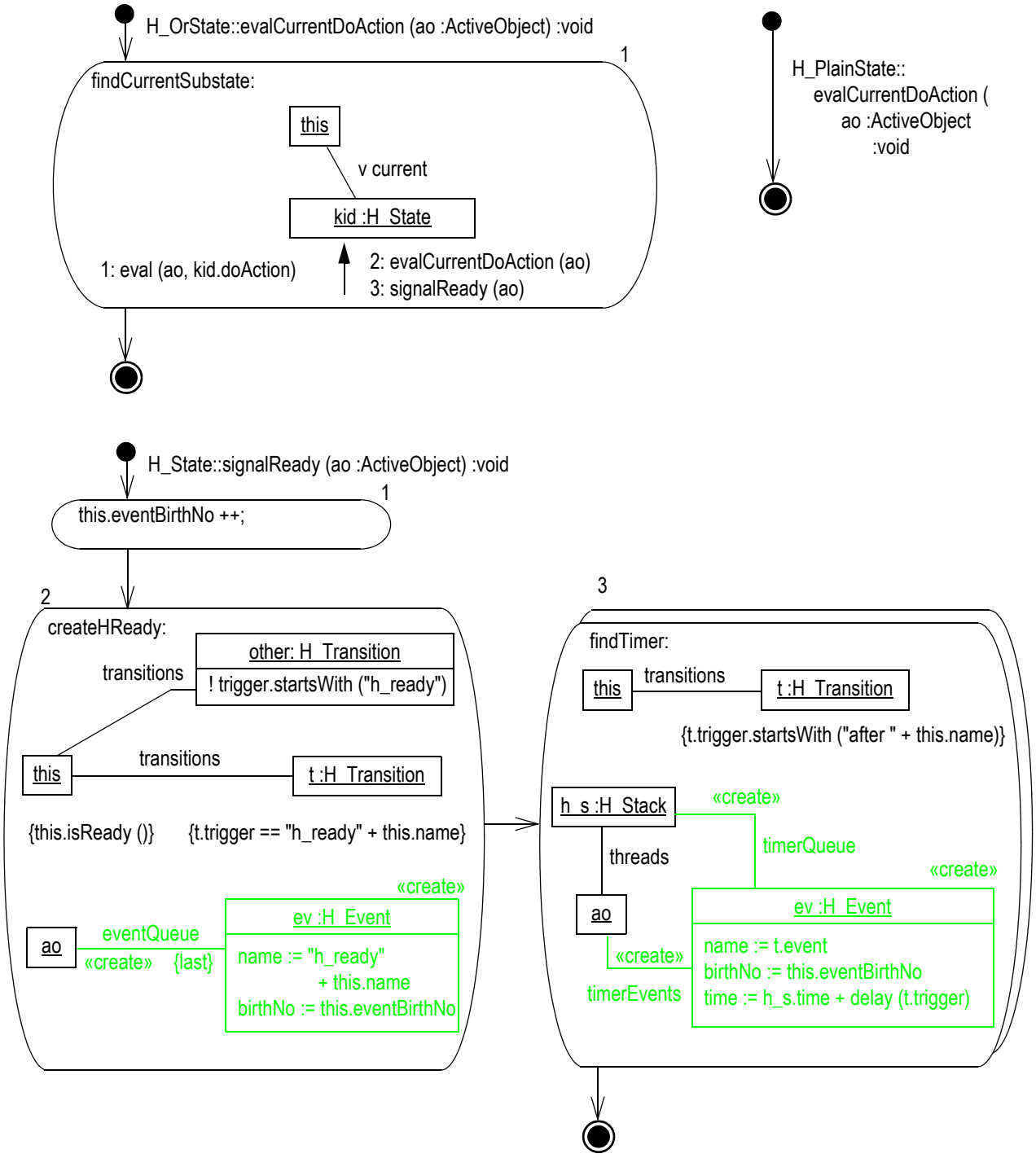
Story pattern findInitialState just looks-up the initial link in order to bind variable newCurrent. Note, according to the class diagram for statecharts, the initial association is mandatory for or-states, cf. Definition A.59. Thus, story pattern findInitialState never fails.

After binding variable newCurrent either by story pattern findHistoryState or by findInitialState we enter activities 6 and 7. Note, story pattern markCurrent uses variable newCurrent as a bound variable. This means, it uses the match for variable newCurrent that has been computed by story pattern findParentOfTarget or by findHistoryState or by findInitialState. Thus, activities 6 and 7 evaluate the entry-action and create the current marker and call method enterTo, recursively, as described above. The recursion terminates as soon as a plain state is reached, when nothing is done, cf. story diagram H_PlainState::enterTo at the top right of Definition A.78. This completes step 6 of our event handling process.

Finally, we have to execute step 7 of our event handling process, i.e. we have to execute the do-actions of all new current states in top-down order. This is achieved by calling method evalCurrentDoAction at the common parent state in story pattern changeStates, cf. Definition A.74. Method evalCurrentDoAction for class H_OrState is shown in Definition A.79. Story pattern findCurrentSubstate just looks-up the current link that has been created by method enterTo and binds variable kid to the corresponding substate. In addition, the do-action of the kid state is evaluated. Finally, story pattern findCurrentSubstate calls method evalCurrentDoAction on its current kid state, recursively. This recursion terminates as soon as a plain state is reached, cf. method H_PlainState::evalCurrentDoAction given on the top right of Definition A.79.

In addition, story pattern findCurrentSubstate calls method startTimers on the kid state. Method startTimers is given in Definition A.79, too. However, this method will be discussed together with the whole timer concept in chapter A.8.6. For normal transitions this completes the event handling process. However, our statechart allows so-called triggerless transitions. This is discussed in the following chapter.

Definition A.79: Executing the do-actions of new current states



A.8.4 Dealing with triggerless transitions.

Once the usual event handling process of a firing transition has been completed and new states have been reached we may have to handle so-called triggerless transitions:

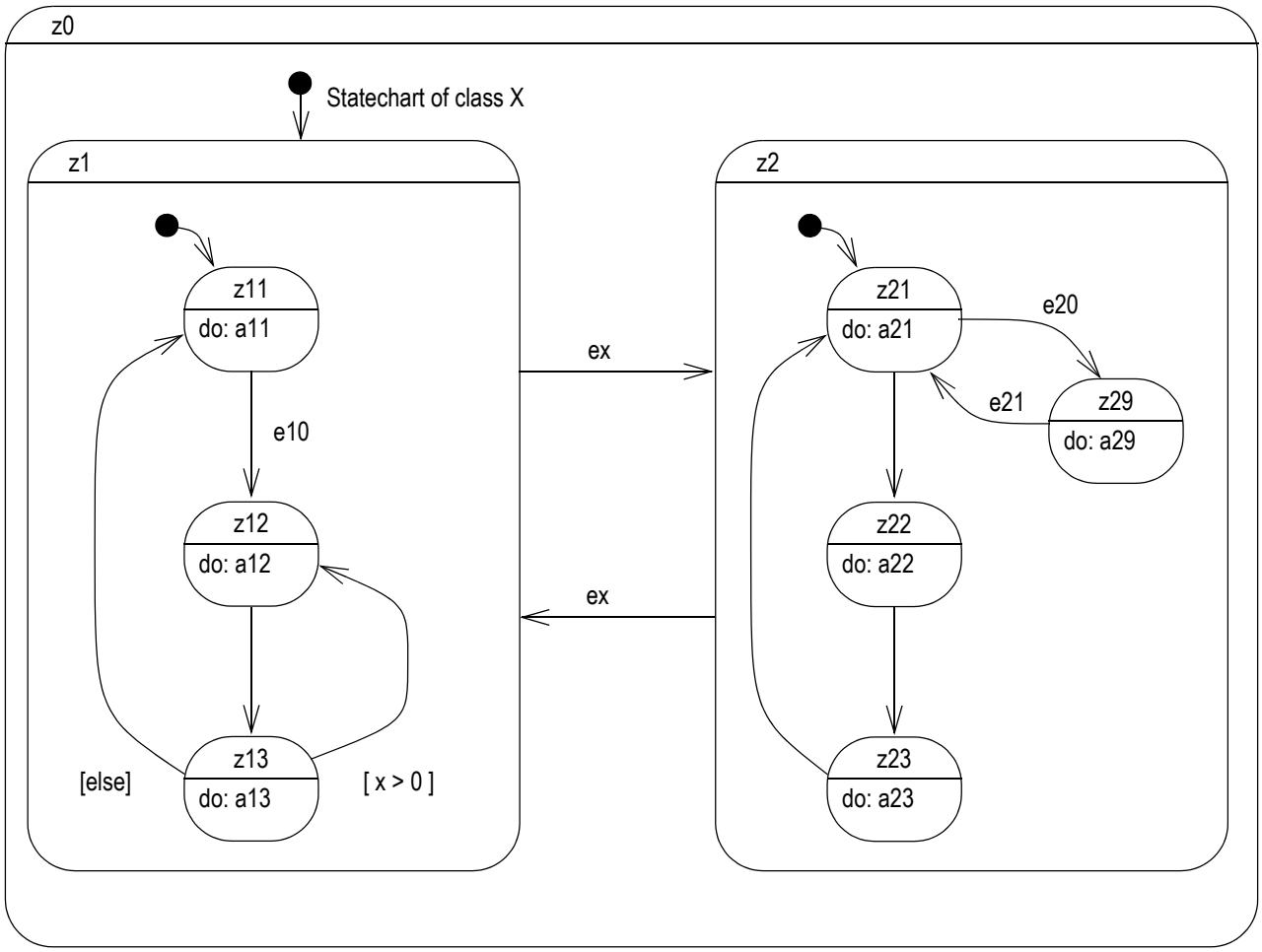
Definition A.80: Triggerless transitions

A transition without an event is called a *triggerless* transition. Triggerless transitions fire as soon as the body of their source state has been executed, completely. For reasons that are discussed below, triggerless transitions are handled immediately. This means, triggerless transitions are handled before the next event is consumed from the ready queue and even before deferred events are handled.

In case of a priority conflict between a triggerless transition and a transition with an explicit trigger leaving a state z at the same nesting level, we first consume events that may have arrived meanwhile. Only if no pending event is waiting, we proceed with the triggerless event.

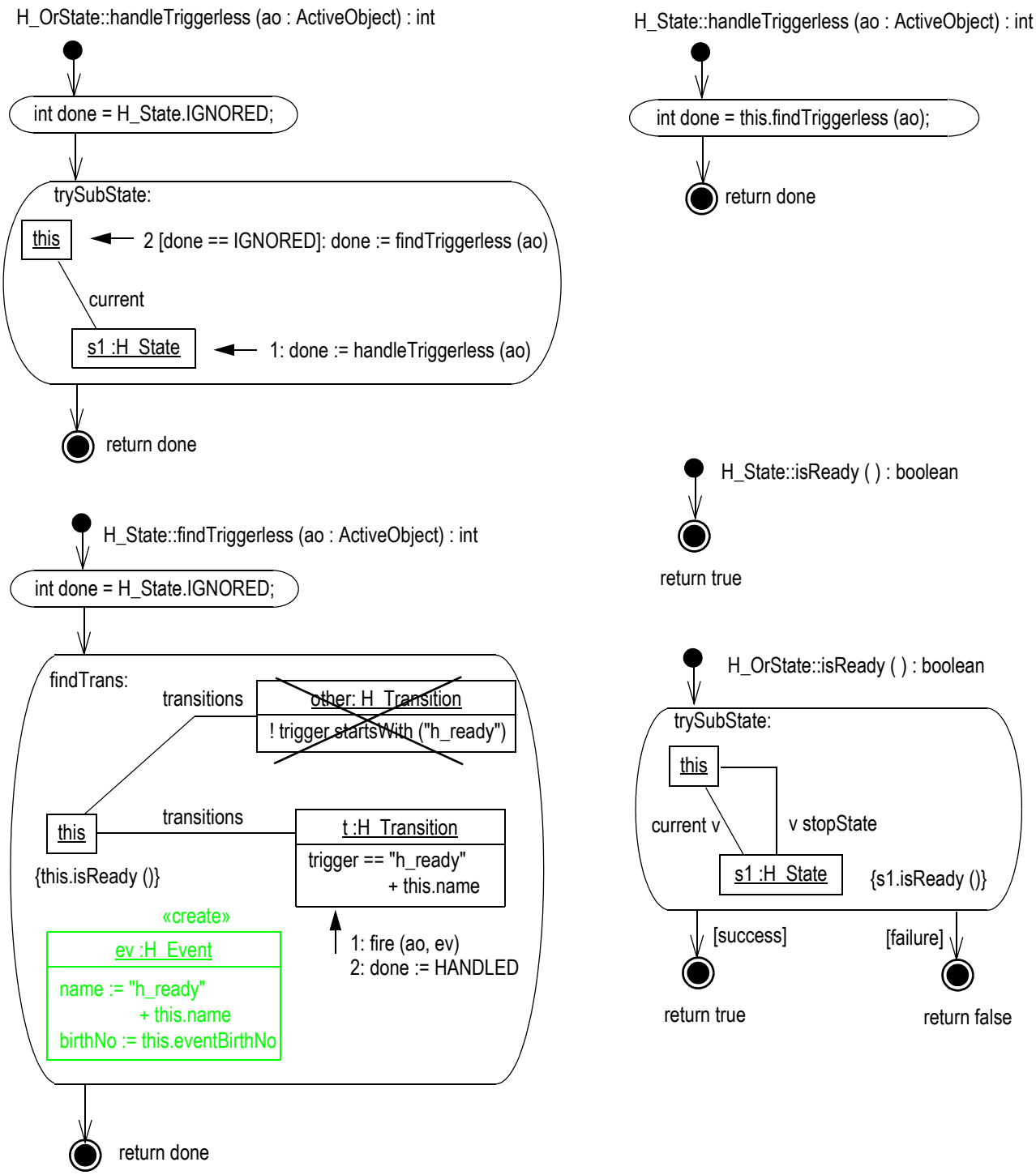
Triggerless transitions are frequently used to model complex computations that do not fit into a single activity or state. Typically, such a complex computation that has been split into several states connected by triggerless transitions needs to be executed completely in order to deliver its result and to reach a consistent overall state again. Example A.81 outlines such situations. In state z1, we usually wait within substate z11. If an e10 event occurs some complex computation is triggered that involves a loop through the states z12 and z13. Note, while the computation loops through states z12 and z13, an ex event may occur that triggers a transition from complex state z1 to complex state z2. However, we assume that the complex computation employing only triggerless transitions should not be interrupted by other events but it first needs to be completed in order to reach a consistent overall state, e.g. state z11, again. Thus the reaction to event ex should be postponed until the computation via triggerless transitions is completed and until the statechart has reached a stable state, e.g. state z11, again. As soon as a stable state is reached, we may react on the next external event.

Example A.81: Different priorities for triggerless transitions



We achieve this behavior by handling triggerless transitions with higher priority. Any time a regular event has been handled, method handleEvent of class ActiveObject reaches story pattern doTriggerless, cf. Definition A.66 on page 176. Story pattern doTriggerless just calls method handleTriggerless on the top level state. Definition A.82 shows method doTriggerless. On or-states, method doTriggerless just climbs down the nesting levels via the current links. If a plain state is reached, we climb up again. At each level, method findTriggerless is called, cf. Definition A.82.

Definition A.82: Handling triggerless transitions



Method findTriggerless checks whether the current state has an outgoing transition with the internal event trigger h_readyXY, where XY stands for the name of the source state. In addition, the negative node other ensures that no other event is leaving the corresponding state, directly. This will be discussed later. If a single outgoing triggerless transition is found, we create an appropriate internal event ev and send it to the current transition by calling method fire. This causes the corresponding state changes and action executions. In addition, the return state of method findTriggerless is set to HANDLED. If a triggerless transition has been fired and the return state has been set to HANDLED, story pattern trySubState of method handleTriggerless will no longer call method findTriggerless on the higher nesting levels but it will skip its collaboration message 2 and terminate, directly. Thus, we climb up the nesting levels again. Note, in class ActiveObject the story pattern doTriggerless is iter-

ated as long as method handleTriggerless returns state HANDLED. This iteration is not interrupted by the arrival of new events or in other terms this loop executes triggerless transitions as long as possible without looking for other events. This achieves the desired uninterruptable execution of complex computations that are modeled with triggerless transitions.

Note, a triggerless transition may only fire, if its source state has been executed, completely. Plain states are executed completely as soon as their do-action returns. or-states are executed completely as soon as a stop state is reached and that stop state is completely executed, too. This behavior is achieved by the constraint {this.isReady ()} employed in story pattern findTrans of method findTriggerless. The definition of method isReady is given in Definition A.82, too. For plain states method isReady always returns true. For or-states method isReady looks-up the current substate s1 and verifies that a stopState link exists between the current or-state and its substate s1. In addition, substate s1 must be ready, too.

State z2 of Example A.81 shows a slightly different situation. State z2 contains a loop of triggerless transitions involving states z21, z22, and state z23. At state z21, there exists an additional transition with an explicit trigger e20. If we stick to the prioritized handling of triggerless transitions, strictly, then transition e20 has no chance to fire, anytime. The triggerless transition from z21 to z22 would always have higher priority. We could deal with this situation by forbidding that a triggerless and an explicitly triggered transition leave the same state. However, this combination of triggered and triggerless transitions may also be interpreted as a "check for event" situation. An active object could have the task to perform certain control tasks, actively. In addition, it may have the task to check the arrival of certain external events, regularly. In order to model the active control task one would obviously use triggerless transitions arranged as a control loop. If this control loop shall check the arrival of external events at a certain point, one would like to do this using an explicitly triggered transition as shown in state z2 of Example A.81. If state z21 is executed completely, the statechart execution should first check the arrival of external events and only if there is no reason to interrupt the computation it should proceed with its triggerless computations. This means, in case of a conflict of a triggerless and an explicitly triggered transition, the explicitly triggered transition should be prioritized.

To achieve this behavior, story pattern findTrans of method findTriggerless shown in Definition A.82 checks if there exists another transition in addition to a potential triggerless transition. Note, according to our statechart model restrictions, no state may have two outgoing transitions with the same trigger. This includes the case that no state may have two outgoing triggerless transitions. If a statechart depicts several triggerless arcs leaving a single state, this represents a single logical transition with multiple targets. Thus, if story pattern findTrans detects a second transition, this second transition must have a different, explicit trigger. In this case we do not fire the triggerless transition but method findTriggerless will terminate without effects. At higher nesting levels this will return state IGNORED as a result of method handleTriggerless. Thus, story pattern trySubState of method handleTriggerless will call method findTriggerless on higher nesting levels, again. However, in our conflict situation there exists some substate s that has outgoing transitions (a triggerless and an explicitly triggered). Thus, that substate s cannot be a stop state. Recall, stop states must not have outgoing transitions at all, cf. the constraints 6) of Definition A.59 on page 166. Thus, in our conflict situation, the findTriggerless calls on higher nesting levels will never fulfill the {this.isReady ()} constraint of story pattern findTrans. This calls of findTriggerless will have no effect. When the top level is reached again and control is passed back to method handleEvent of class ActiveObject, the iteration of story pattern doTriggerless will terminate since state IGNORED is returned.

So far, the described behavior of method handleTriggerless and findTriggerless achieves the interruption of the execution of triggerless transitions, only. We now want to look-up the event queue and check for events triggering the explicit transition(s). If no such event exists, we would like to proceed

with the triggerless transition. We achieve this behavior by just creating an internal h_readyXY event and by enqueueing this h_readyXY event to the usual event queue of the corresponding active object ao. The usual event handling process will look-up and consume all events that have been arrived and are waiting within the event queue first and then it will consider the internal h_readyXY event. Thus, if an explicit event is waiting, that event will be checked and then the internal h_ready event is considered. If an waiting event fires an explicitly triggered transition, we will just ignore the h_ready event later on.

First we have to create and enqueue an internal h_readyXY event. This is done by default for all states during the evaluation of do-actions by method evalCurrentDoAction, cf. Definition A.79. Once story pattern findCurrentSubState has evaluated the do-action of the kid state, it calls method signalReady on that kid state. Method signalReady increments the eventBirthNo attribute of the current state in order to signal that the state is reached another time. Then, story pattern createHReady creates an h_readyXY event, where XY refers to the name of the source state. In addition, the eventBirthNo value of the current state is stored within the event. This event is added to the event queue of the corresponding active object ao.

Second, we have to ensure that h_ready events trigger only that triggerless transition that has created them. Note, meanwhile another event may have caused a change to another state that has a triggerless and an explicitly waiting transition, too. In that case the first h_ready event should not trigger that foreign triggerless transition but we should proceed with the event consumption until an appropriate event is considered. Similarly, some event may have fired some other transitions and we may have reached the old birth state of the h_ready event again. In that case the outdated h_ready event should be ignored and the event consumption should proceed. Finally, the appropriate h_ready event may be consumed and then it may trigger the corresponding triggerless transition (if nothing preventive has happened in between).

To achieve the described behavior, triggerless transitions have an implicit trigger h_readyXY where XY refers to the name of their start state. Thus, a triggerless transition may be fired by the usual event handling process, only, if an internal h_ready event with an appropriate name is consumed. If the state has changed e.g. due to an explicit transition, late arriving h_ready events are just ignored. In addition, all h_ready events carry an event birthNo attribute. If a state has been left via some transition and it is reached again then the eventBirthNo attribute of that state is incremented by method sendReady, cf. Definition A.79. If the birthNo attribute of an arriving internal event and the eventBirthNo attribute of the current state do not match, then that event is filtered out by story pattern checkOutdatedEvents of method H_Transition::fire, cf. Definition A.74 on page 183.

In state z2 of Example A.81 we exploit this "check for events" semantics of triggerless transitions. In state z21 we have a conflict between the triggerless transition targeting state z22 and the explicitly triggered transition e20. Thus, if we are in state z2, then we iterate through states z21, z22, and z23, continually. From state z22 to state z21, the triggerless transitions cannot be interrupted. However, each time we reach state z21, the events of the usual event queue are considered, first. If meanwhile an e20 or an ex event has arrived, then the corresponding transition fires and we reach state z29 or state z1, respectively. If no such event was waiting, we proceed with the triggerless transition targeting state z22. Note, the computation of state z2 may be interrupted in state z21, only. The other states may perform some complex computation. If an ex event arrives, the complex computation is not interrupted immediately, but it is completed first. The computation may be interrupted at state z21, only, where we check for other events, explicitly.

To summarize, in our approach triggerless transitions may model two different situations. If solely triggerless transitions are used, they model a complex computation that will not be interrupted. This is very important, since interrupting a complex computation at some undefined point in time could

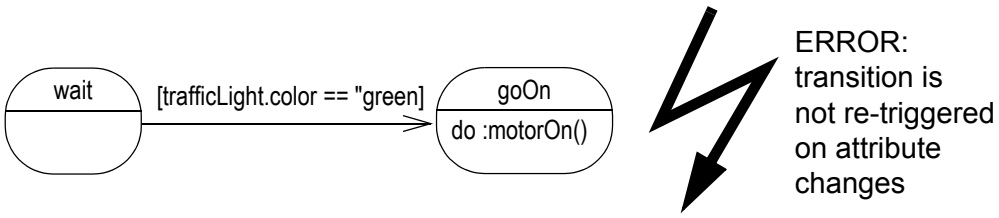
cause inconsistent internal states. However, this prioritized handling of triggerless transitions raises the risk of non-terminating loops that may cause that the active object is trapped and does no longer react on any events send to it. If the situation allows it to interrupt a complex computation at certain points, the user may use triggerless and explicitly triggered events at the same state. Such situations indicate states where we still would like to go on with a complex computations but in addition, we would like to be able to react on certain external events, e.g. an (emergency) stop signal.

Note, the prioritized handling of triggerless transitions raises the risk of non-terminating computations at the statechart level. While the statechart is in such a (non-terminating) loop, it is not able to consider any external event. Thus, it will ignore e.g. an emergency stop event arbitrary long. As in the body of methods called within the different actions attached to the statechart, the user is in charge to guarantee that he does not cause such a hazard. Conservatively, we could forbid loops of triggerless transitions that are not interrupted at some point. Currently, we perform some more case studies on this point.

Note, we are aware, that the different handling of triggerless transitions exhibits tricky semantic details and that the average user of our statecharts may not be aware of these differences. However, due to our experiences with several volunteers, the chosen handling of triggerless transitions closely reflects our users intuition. Interrupting a sequence of triggerless transitions e.g. via an explicitly triggered transition leaving a state at higher nesting level often surprises our users and leads to inconsistencies. In addition, most users have the intuition, that triggerless transitions are executed "quickly" or without loss of time. They probably will not think about a defer clause that would be necessary e.g. in order not to loose an e20 event in state z2 of Example A.81 which could arrive while we are in states z22 or z23. However, if an explicitly triggered event is used together with a triggerless transition, the user would not expect that the explicit transition is always ignored due to a higher priority of the triggerless transition. In this situation he wants to check for the external event and if its not there he want to proceed with the computation. This observations inspired our semantics definition and we have confirming feedback from our user trials.

In the literature, triggerless transitions with guards are sometimes used to model situations where a statecharts waits until some boolean expressions becomes true. In Example A.83, once state wait is reached, the (omitted) do action of state wait is executed and handleTriggerless considers the transition to state goOn. If the guard evaluates to false, the transition does not fire and the statechart stays in state wait. If the color attribute of the traffic light changes now to value "green" the boolean expression becomes true. However, the main event loop of our shuttle reacts on events delivered to its event queue, only. It does not check attribute values, automatically. Thus, the shuttle does not recognize that the attribute has changed and execution will not proceed.

Example A.83: Erroneous wait for boolean condition



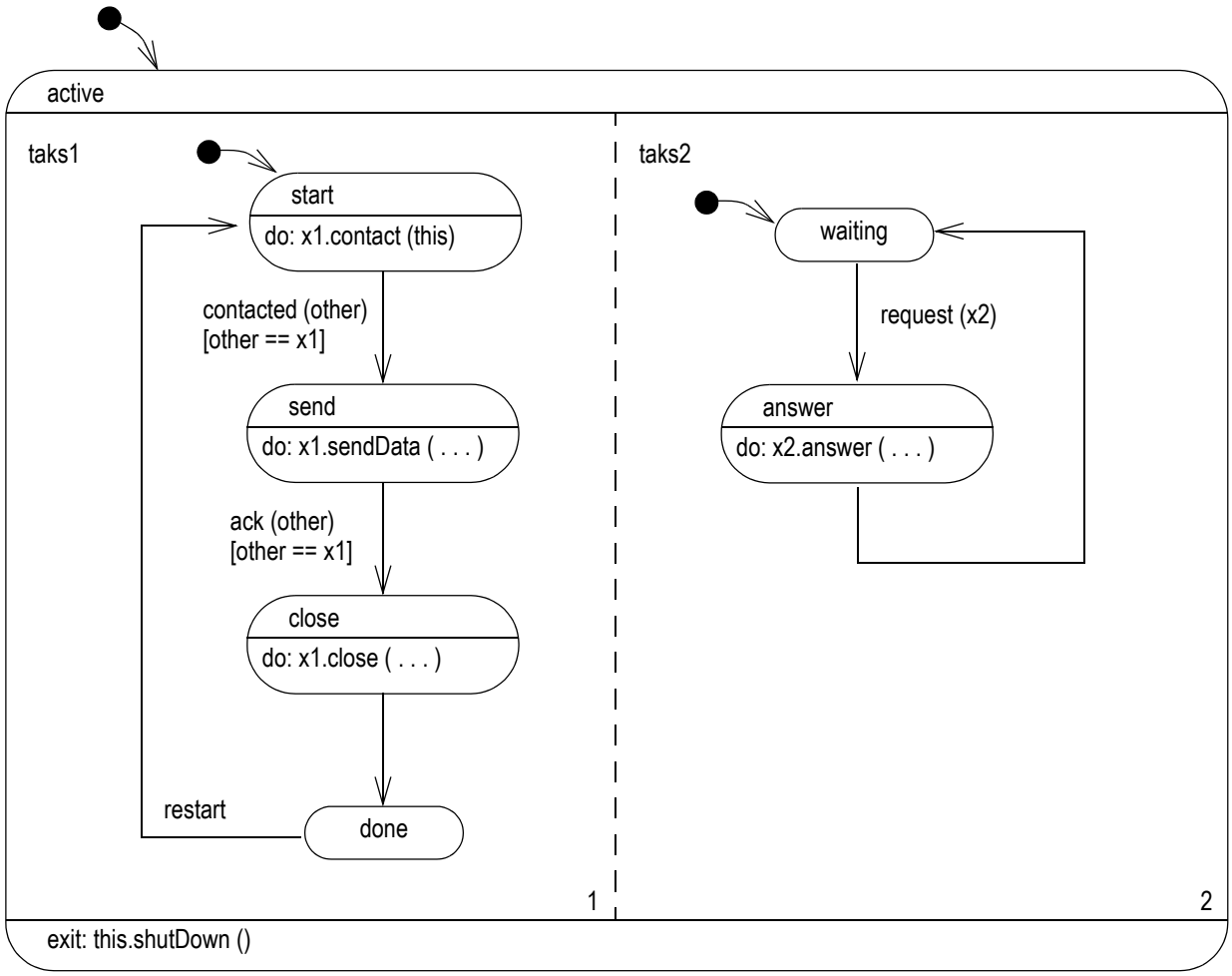
While it may be handy for the specification of certain waiting situations to wait until some data condition becomes true, the implementation of such a feature is complicated and thus our approach does not provide such a mechanism. The recommended solution for such a specification problem is to use an explicit notification mechanism like the Observer pattern, cf. [GHJV95]. The shuttle should subscribe itself as an observer for the color attribute of the traffic light and the setColor method of the traffic light should notify its observers by sending them an appropriate event. This event should be used as an explicit trigger for the corresponding transition.

A.8.5 Dealing with and-states

And-states allow to specify "parallel" processes within a single statechart. For example, an active object may have to supervise two independent sensors or actors or it may exchange certain events with two other active object where each of these communications employs a certain communication protocol, cf. Example A.84. Usually, parallel and-states work independent of each other, i.e. they recognize distinct sets of events and do not care in which substate the parallel sibling states are. However, in general it is allowed that two parallel and-states react on the same events and that one and-substate uses a guard checking the substate of its sibling. In addition, the parallel substates may employ guards and actions that look-up and modify the current object structure. Such effects may affect the behavior of sibling substates. As already discussed, our approach sequentializes parallel events and actions. Accordingly, we sequentialize the execution of parallel and-states. The substates of an and-state are stored using an ordered association, cf. Definition A.59. Our interpreter considers the substates in this order. The interpreter forwards the handleEvent request to each substate, one after the other, and executes the corresponding guards, actions, and state changes for one substate after the other.

Note, the event handling of and-states is introduced into our framework by just overwriting the event handling methods of class H_State. In this way, class H_AndState provides a specific behavior for the methods handleEvent, leaveUpTo, doExitUpTo, enterTo, evalCurrentDoAction, handleTriggerless, and isReady.

Example A.84: A simple and-state



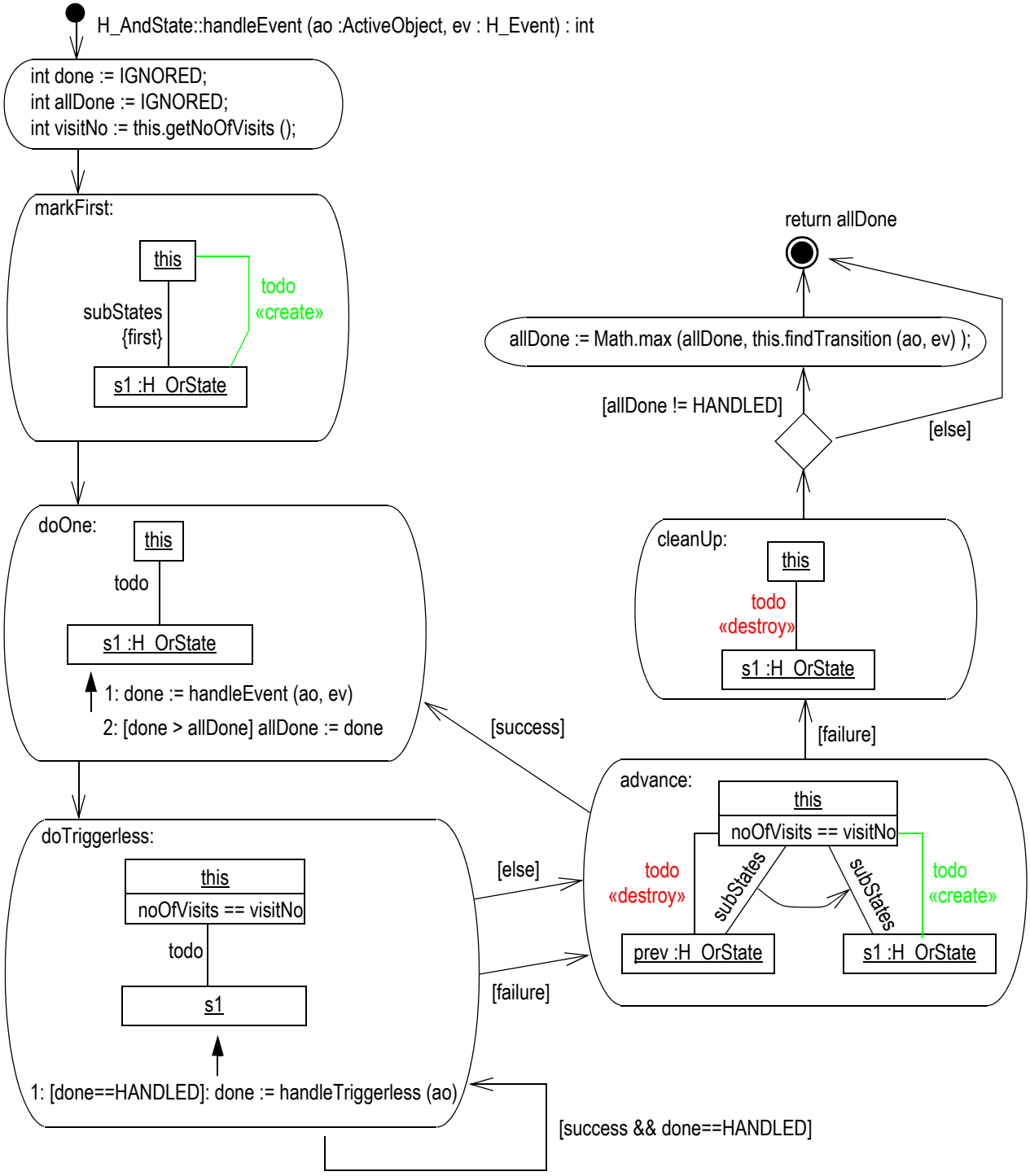
Method handleEvent is called on the top state of the whole statechart. Its task is to climb down the nested states via current edges, recursively. From the leave state(s) upwards, method handleEvent calls method findTransition on each nesting level. Method findTransition looks for outgoing transitions

with matching event names. On success, method findTransition forwards the event to the corresponding transition.

Within an and-state all direct substates are active "in parallel". One may consider each of the parallel states as its own active subobject with its own substatechart. In this view, method handleEvent serves as the scheduler that assigns a time slot to each of the substatecharts, one after the other. This shall be done in the order prescribed by the subStates association. To achieve this, story pattern markFirst of method handleEvent of class H_AndState creates a todo link marking the first element of the subStates association of the current and-state, cf. Definition A.85. Story pattern doOne forwards the handleEvent call to the currently marked substate. Note, method handleEvent will return whether the event has been IGNORED or DEFERRED or HANDLED by the corresponding substatechart. For the whole and-state, we consider an event as handled if one of its substates does so and we consider it as deferred if one of the substates defers it. To achieve this, story pattern doOne accumulates the highest done value in variable allDone. Note, the values are related IGNORED < DEFERRED < HANDLED. The value of variable allDone will be returned, finally.

Each time story pattern doOne has forwarded the received event to the current substate, we reach story pattern doTriggerless which handles triggerless transitions. Note, each substate of an and-state is handled like a substatechart. As discussed in chapter A.8.4, in our approach sole triggerless transitions model non-interruptable computations. For and-states this means, if one substate fires a transition, then subsequent triggerless transitions fire, too, before the next sibling substate is considered. Story pattern doTriggerless checks two conditions. First, the event forwarded by story pattern doOne may have fired a transition which leaves the whole and state. In this case, attribute noOfVisits of the and-state is incremented, which will be discussed in Definition A.89. The attribute condition of the this object ensures that the noOfVisits attribute has not changed. If in addition the handleEvent operation called within story pattern doOne has returned state HANDLED, story pattern doTriggerless calls method handleTriggerless. This is repeated by the self transition with guard [done==HANDLED] as long as triggerless transitions fire. Method handleTriggerless has been discussed in Definition A.82.

Definition A.85: Handle event for and-states



After the execution of story patterns doOne and doTriggerless, story pattern advance is reached. Again, the attribute condition noOfVisits==visitNo ensures that no transition has left the whole and-state. If the and-state has not been left, story pattern advance tries to advance the todo marker to the next element within the list of subStates links. On success, story pattern doOne is executed, again. This is repeated until the last substate has been considered.

Note, even if some substate has already handled the current event, we forward the event to all other sibling substates, too. This corresponds to the usual semantics of and-states where a single event may be recognized and handled by each substate, individually. However, if one substate fires a transition that leaves the whole and-state, then the subsequent substates are not considered any more since they

are no longer active. In Example A.86, transition eout leaves state z3bb toward state z6. If this transition fires, we leave not only state z3bb (and all its substates) but also the parent state p2bb and its siblings p2ba and p2bc and the parent and-state z2b and the two parallel substates p1a and p1b and their parent z1. Thus, it makes no sense to try to send an handleEvent message to state p2bc since state p2bc is not active, any more. Note, in some cases early parallel states may have reacted on some event e when some parallel state leaves the whole and-state and all later siblings do not even "see" the event, although they might have reacted on it. This order dependent behavior results from dropping the double-buffering semantics and is unavoidable within our approach. At least, our approach provides a clear order in which parallel states are considered. However, this complex semantic details may easily be overseen and thus they indicate poor modeling style and should be avoided.

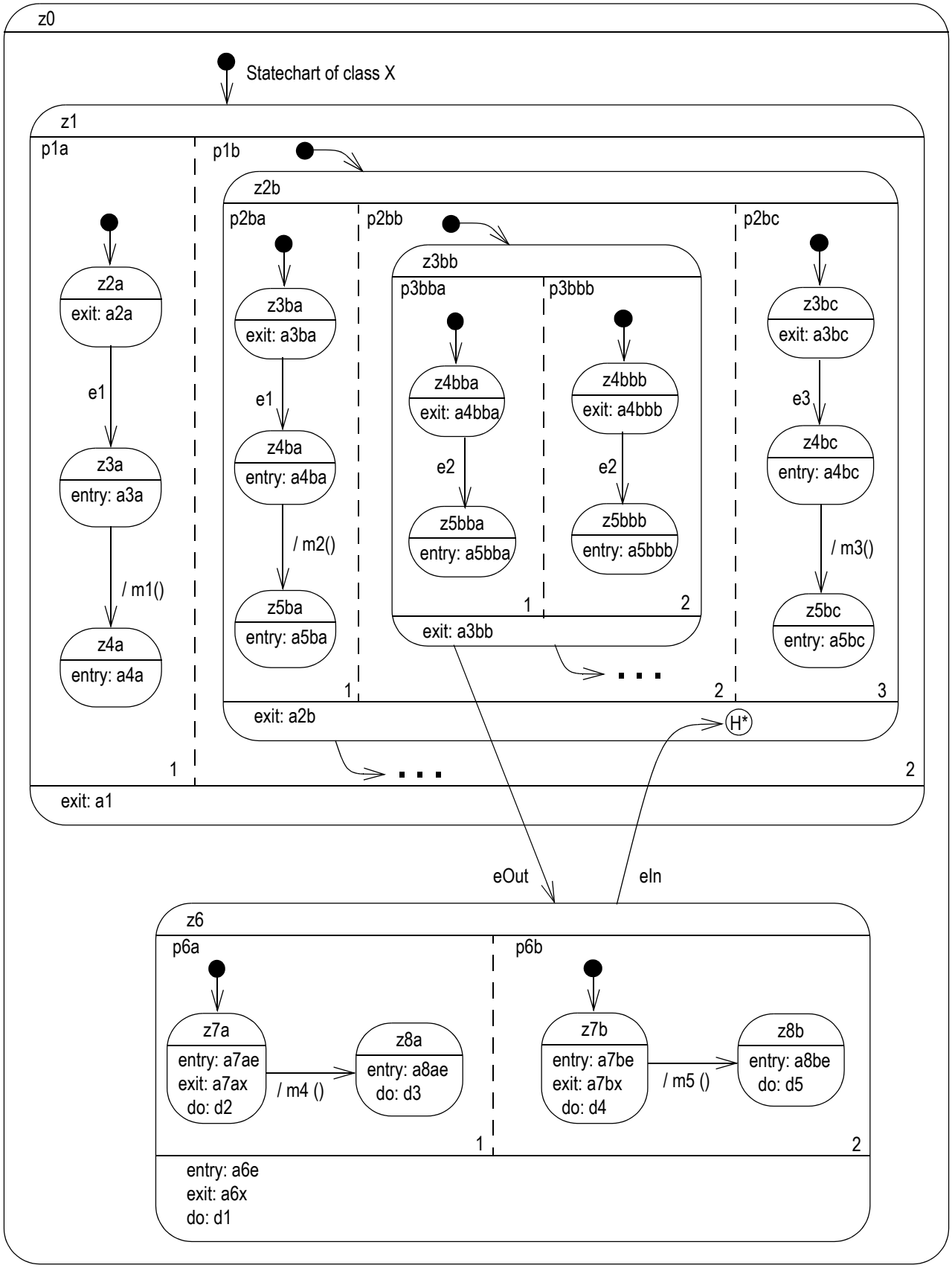
Once story pattern advance fails, the execution reaches story pattern cleanUp. Story pattern cleanUp just destroys the todo link. Finally, we check the status of variable allDone. In state doOne variable allDone accumulates the highest return value of method handleEvent from all substates. If one or more of the substates has handled the event, then the whole and-state considers the event as handled and we just return the value of allDone. If none of the substates was able to deal with the event, then we call method findTransition on the and-state, in order to check for transitions or defer clauses attached to the and-state itself.

To illustrate the behavior of method handleEvent for and-states, let us assume that the statechart in Example A.86 has just been initialized, i.e. the statechart is in all the initial states. If event e1 would occur in this situation, method handleEvent would descent from state z0 to z1. The parallel state z1 would first consider substate p1a. Note the box for or-state p1a is not shown. The ordering of substates is indicated by the ordering numbers in the lower right corner of the substate compartment. The or-state p1a further descents to state z2a. State z2a actually detects an appropriate transition and forwards the call to it. This will cause the corresponding state change from state z2a to state z3a including the execution of the attached exit-, transition-, entry-, and do-actions, as far as present.

Method handleEvent of and-state z1 recognizes that substate p1a has handled the event and stores this in its status variable allDone. Next, story pattern doTriggerless fires the triggerless transition from z3a to z4a. This causes the corresponding state change from z3a to z4a and the execution of method m1 and entry-action a4a. Once sub-state p1a has finished, method handleEvent of state z1 proceeds with the next parallel substate, i.e. it calls handleEvent on state p1b. State p1b forwards the call to state z2b which forwards it to state z3ba. State z3ba has an appropriate transition and fires it. This causes the corresponding state change to state z4ba. State z4ba has an triggerless transition which fires. We switch to state z5ba and execute method m2 and entry-action a5ba. On return of method handleEvent from state p2ba, state z2b continues with state p2bb and p2bc, which are not able to handle the event.

Let us now assume that an eout event occurs in Example A.86. Method handleEvent will search through the nested and-states as described and after considering the substates of state z3bb, method findTransition will detect the eout transition leaving state z3bb. Accordingly, findTransition calls method fire on that transition. The common parent of state z3bb and state z6 is state z0. Thus, story pattern changeStates of Definition A.74 calls method leaveUpTo on state z3bb passing state z0 as parent. We now have to leave the substates of state z3bb and of z2b and of z1. Nested or-states are left from bottom to top executing the exit operations in that order. For parallel states we leave all substates in the user-defined order.

Example A.86: Example statechart with complex nested and-states



<<Caution: The following description of exiting and-states is going to be changed in the next version of this work. In future, we will just determine the sub-tree of states that needs to be left and then we will execute the corresponding exit-actions in a standard post-order traversal of that subtree. >>

To achieve this behavior, we again use method leaveUpTo to descent into the nesting levels until leaf states are reached, cf. Definition A.77. Then, we use method doExitUpTo in order to execute the exit-actions and to climb up the nesting levels. In case of an and-state method leaveUpTo just looks-up the first parallel substate and forwards the leaveUpTo call, cf. Definition A.87. In addition, method leaveUpTo of and-states creates a toLeave link to mark the substate under consideration. In case of our example this marks state p3bba with an toLeave link and calls leaveUpTo on it. In turn method leaveUpTo is called on state z4bba. The plain state z4bba switches to method doExitUpTo. Method doExitUpTo will leave state z4bba and state p3bba as described in Definition A.77. This will call method doExitUpTo on state z3bb.

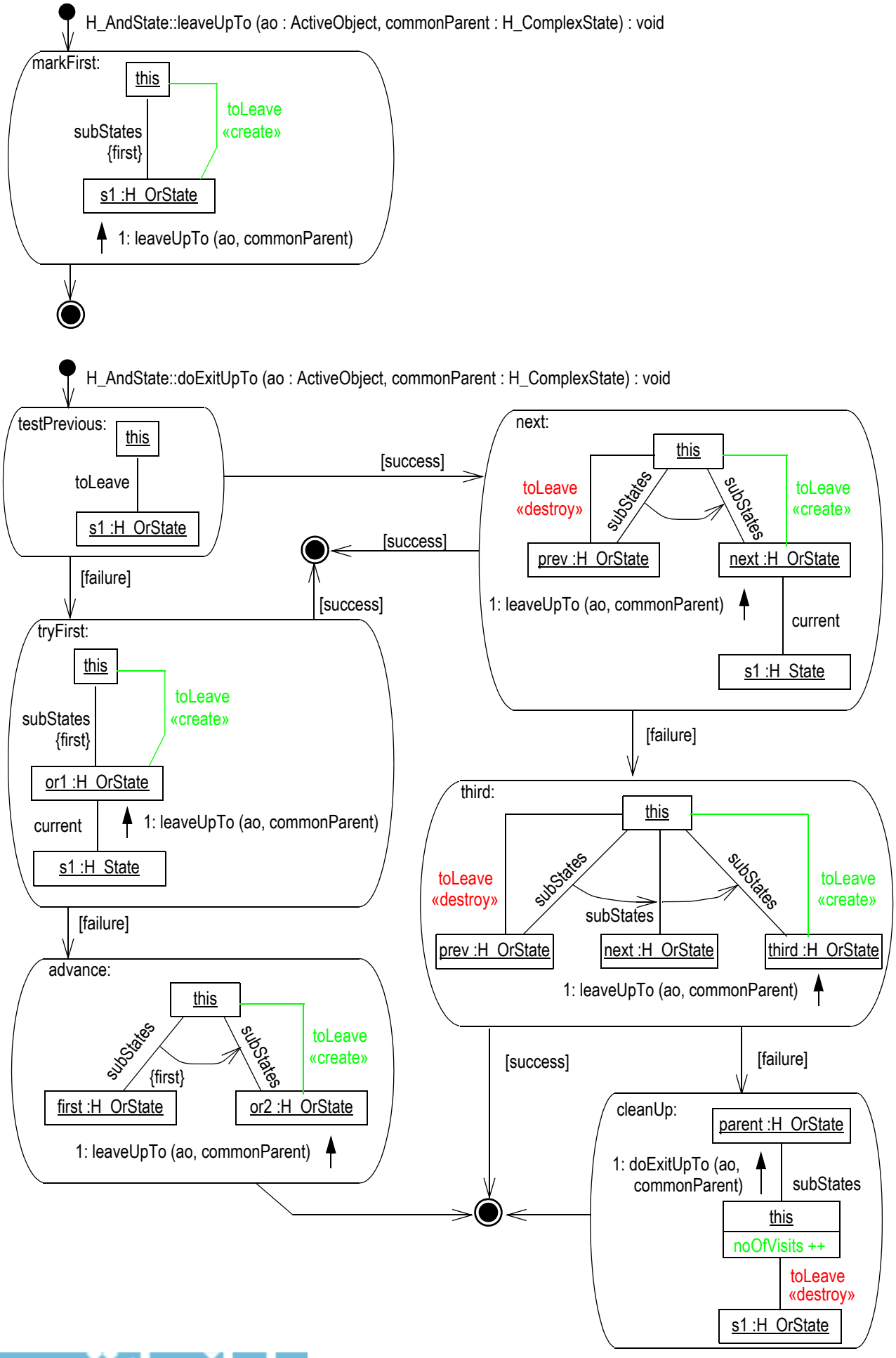
For and-states method doExitUpTo has to check whether a toLeave link exists already, cf. story pattern testPrevious in Definition A.87. In our example this holds for state z3bb and thus we continue with story pattern next. Story pattern next tries to advance the toLeave link to the next sibling substate of the current and-state. On success, it calls method leaveUpTo on the next substate and method doExitUpTo terminates. Note, method leaveUpTo will descent into the corresponding substate until it reaches a leaf state. On the leaf, state method doExitUpTo is called, again, which will climb up to the current and-state and thus the execution will reach story pattern next, again. In this way all sibling states are left in the pre-defined order. Story pattern next may fail due to two reasons. First, the toLeave link may already have reached the last substate and second, the next substate may not have a current link. In these cases we reach story pattern third.

The next substate may miss a current link because it has already been left. In Example A.86, the eout transition leaves and-state z3bb. Once this has been completed, we leave or-state p2bb. In that way method doExitUpTo reaches state z2b the very first time. On state z2b it will begin and leave the first substate p2ba. When that recursive call of method doExitUpTo reaches z2b, again, it will try to advance the toLeave link from substate p2ba to substate p2bb. However, since we started to leave the current states at state z3bb, we already have left substate p2bb and we have to skip it. We have to continue with the second next substate, i.e. p2bc. In method doExitUpTo this is achieved by story pattern third. Story pattern third looks-up the second next substate and calls leaveUpTo, recursively. However, story pattern third may fail if we have reached the end of the substate list. In that case, all substates are left and story pattern cleanUp removes the toLeave link and calls doExitUpTo on the parent of the current and-state.

If method doExitUpTo reaches an and-state the very first time, then story pattern testPrevious fails. In that case we proceed with story pattern tryFirst. Story pattern tryFirst looks-up the first substate and tries to call method leaveUpTo on it. However, as discussed above, we might already have left the first substate. In that case, story pattern advance looks-up the second substate and calls leaveUpTo on that one.

If the statechart of Example A.86 is in its initial state and an eout event occurs, our implementation of method leaveUpTo and doExitUpTo execute the exit-actions of left states in the following order: a4bba, a4bbb, a3bb, a3ba, a3bc, a2b, a2a, a1. We call this order the local-exit-first order. Choosing this order was motivated by several interviews with different Fujaba users. Most of these users expected that if the eout transition fires, one first leaves state z3bb (and it substates) and then one continues by climbing up the state hierarchy. However, the local-exit-first-order has the disadvantage that the order in which states are left depends on the transition that fires. If another transition exists (and fires) that leaves e.g. state z3bc towards state z2 a different execution order of exit actions would result. To avoid this we could have defined a so-called exit-in-fixed-order semantics. This semantics computes that actually state z1 is to be left and starts method leaveUpTo on state z1, always. This would achieve the same execution order independent on the transition that fires.

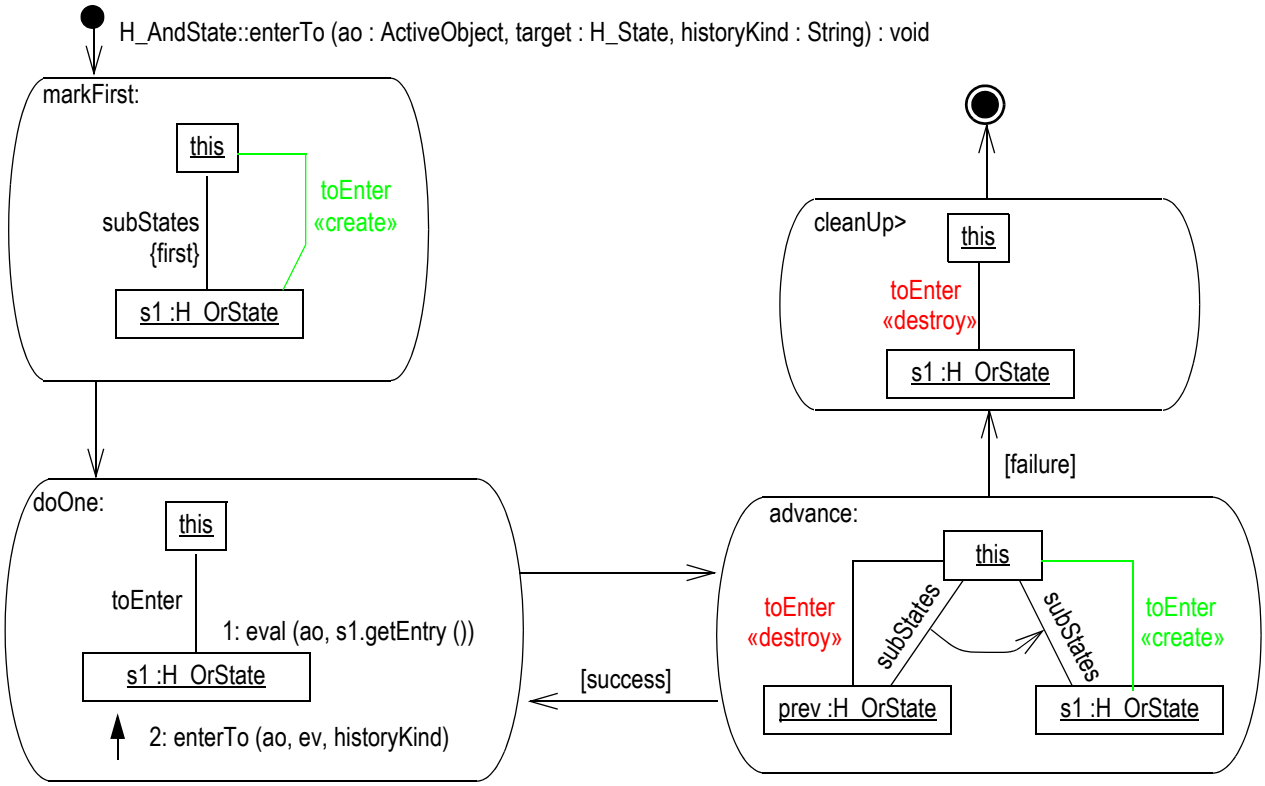
Definition A.87: Leaving and-states



However, in our example the exit-in-fixed-order semantics would execute action a2a, first, which was a total surprise for all interviewed users and contradicted to their intuitive understanding. Therefore, we chose our local-exit-first semantics. However, this implies that the specifier have to be aware, that the substates of an and-state are left in different orders, depending on the transition that fires.

Once we have left state z1 and all its substates, story pattern changeStates of method Transition::fire calls method enterTo on the common parent state, passing the target state of the firing transition as parameter. Entering an and-state is quite simple. We just have to enter all substates in the pre-defined order. This is achieved by method H_AndState::enterTo, cf. Definition A.88.

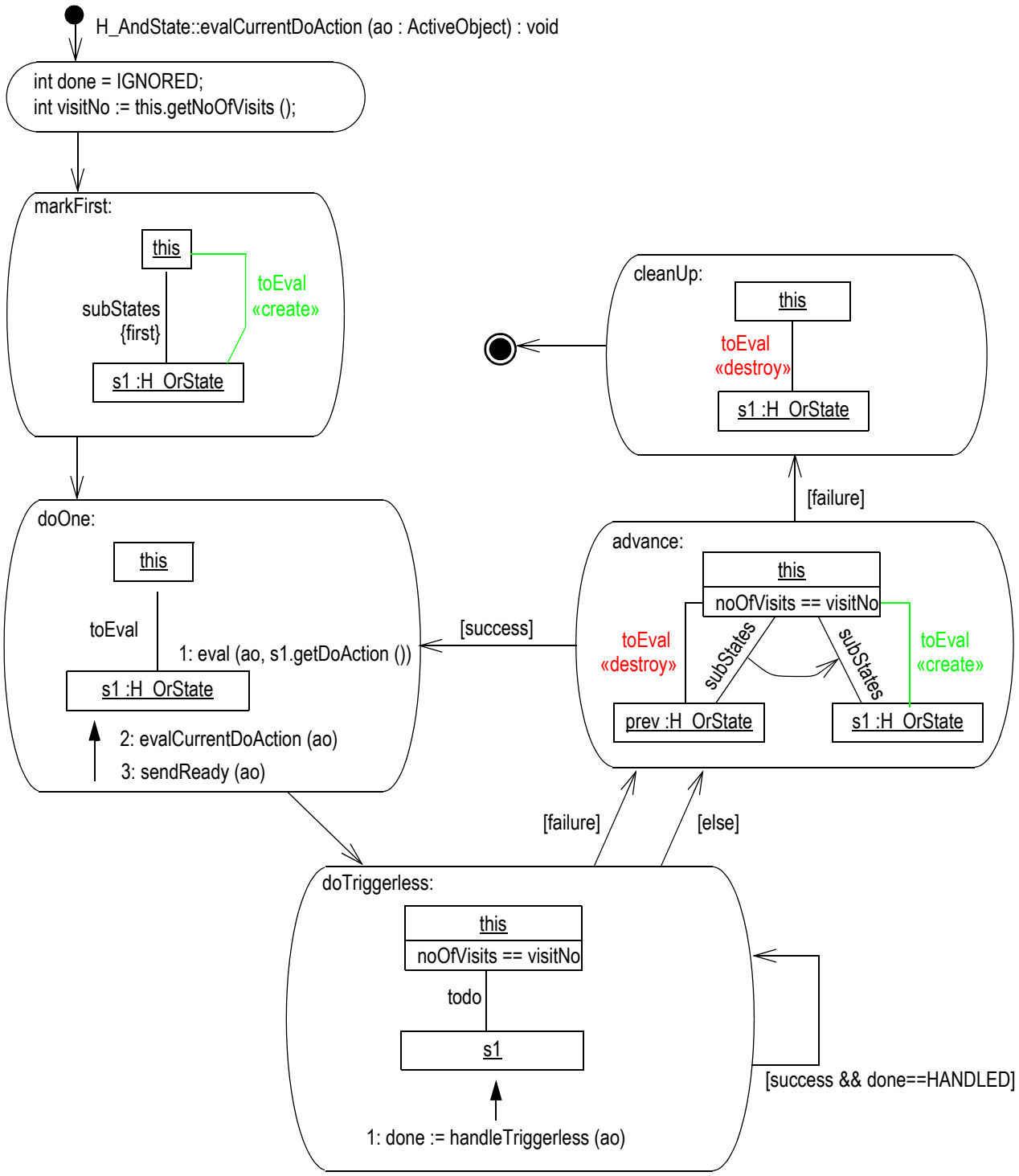
Definition A.88: Entering and-states



After entering the and-state, we perform the do-actions of the new states. Again, this is done in the pre-defined ordering of the substates, cf. Definition A.89. On each substate story pattern doOne evaluates the corresponding do-action, calls evalCurrentDoAction, recursively, and calls method signalReady, cf. Definition A.79. Directly after executing the do-actions of one substate we handle sole triggerless transitions in that substate, too. This is achieved by story pattern doTriggerless. Story pattern doTriggerless just calls method handleTriggerless on the current substate s1 as long as triggerless transitions are detected, cf. the self transition with guard [done==HANDLED]. As in Definition A.85, the handling of triggerless events is aborted if the whole and-state has been left, condition [noOfVisits==visitNo]. Once all do-actions are executed and all sole triggerless transitions have fired, we proceed with the next sibling substate of the current and-state until all substates are visited (or the and-state has been left).

Note, in our semantics triggerless transitions fire directly after the do-actions of the corresponding substate of an and-state. We have already discussed which exit-actions are executed in Example A.86 if the statechart is in its initial state and an eout event is handled. Once the exit-actions are executed method enterTo establishes the new states z6, z7a, and z7b and executes the corresponding entry-actions a6e, a7ae, and a7be in this order.

Definition A.89: Evaluating do-actions for and-states



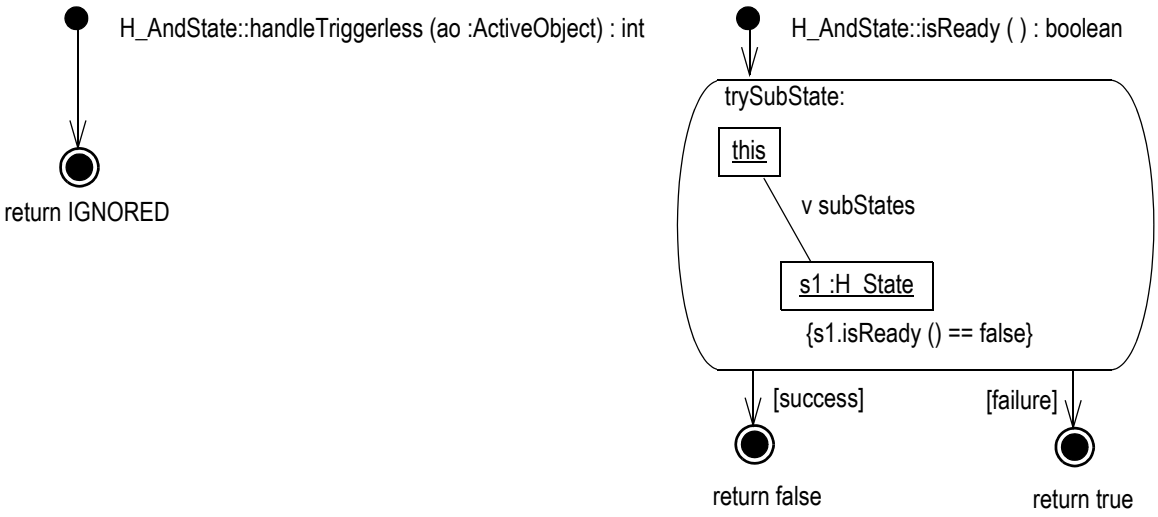
Next, evalCurrentDoAction is called on state z0. Method evalCurrentDoAction of state z0 detects that z6 is the current substate and it evaluates do-action d1 and forward its call to and-state z6, cf. Definition A.79. State z6 is an and-state. Thus, story pattern doOne of Definition A.89 first visits substate p6a, which has no do-action, and calls method evalCurrentDoAction on it. On substate p6a method evalCurrentDoAction executes do-action d2 and then it terminates. However, method evalCurrentDoAction of state z6 reaches story pattern doTriggerless now. Story pattern doTriggerless fires the triggerless transition from z7a to z8a which causes the execution of actions a7ax, m4, a8ae, and d3 in that order. Now we are done with state p6a and proceed with state p6b. We execute do-action d4 and

fire the triggerless transition attached to z7b. This causes the execution of actions a7bx, m5, a8be, and d5 in that order.

Note, the important point is that the triggerless transition of substate p6a is handled before the do-actions of substate p6b are considered. This corresponds to our point of view, that each substate of an and-state is considered as its own substatechart with its own event handler. Accordingly, we consider sole triggerless transitions as a kind of complex computation that somehow belongs to the do-action of their source state. Thus, this complex computation should be executed without interruption. If we would execute all do-actions of all substates, first, and handle the triggerless transitions of all substates in a second run, then substate p6b would face an interrupt between the execution of its do-action d4 and the execution of its triggerless transition. Our semantic avoids that by handling triggerless transitions directly after the do-actions of an substate. However, a triggerless transition may also leave the whole and-state. In this case, the execution of do-actions would stop after the corresponding substates and the do-actions of the subsequent substates would not be considered. This creates the strange situation that all substates of a certain and-state are properly entered (including the execution of their entry-actions) but some of the corresponding do-actions may not be reached. We consider such surprising situations as bad modeling style and, therefore, we recommend that triggerless transitions should not cross the boundaries of and-states.

Note, and-states handle their triggerless transitions themselves. Either, an explicit transition is triggered within some substate and then the triggerless transitions are directly handled by method H_AndState::handleEvent. Or, triggerless transitions are attached to the initial or history states of and-states that are entered. In that case method H_AndState::evalCurrentDoAction handles them, directly. However, once the call of handleEvent terminates on the top-level state, method ActiveObject::handleEvent may call handleTriggerless on the top-level state, too, cf. Definition A.66. If such a call reaches an and-state, there is no more work to do. Accordingly, method H_AndState::handleTriggerless has just an empty body and returns state IGNORED, cf. Definition A.90.

Definition A.90: And-states and triggerless transitions



Finally, we need to be able to determine if an and-state has finished its execution such that a triggerless transition leaving it may fire. An and-state has finished its execution if all its substates are ready or in other terms if none of its substate is not ready. Therefore, story pattern trySubState of method H_AndState::isReady tries to look-up an substate s1 which is *not* ready, cf. Definition A.90. If that fails, i.e. if all substates are ready, we proceed along the failure transition and return true. Otherwise we return false.

A.8.6 A simple handling of time and after events in statecharts

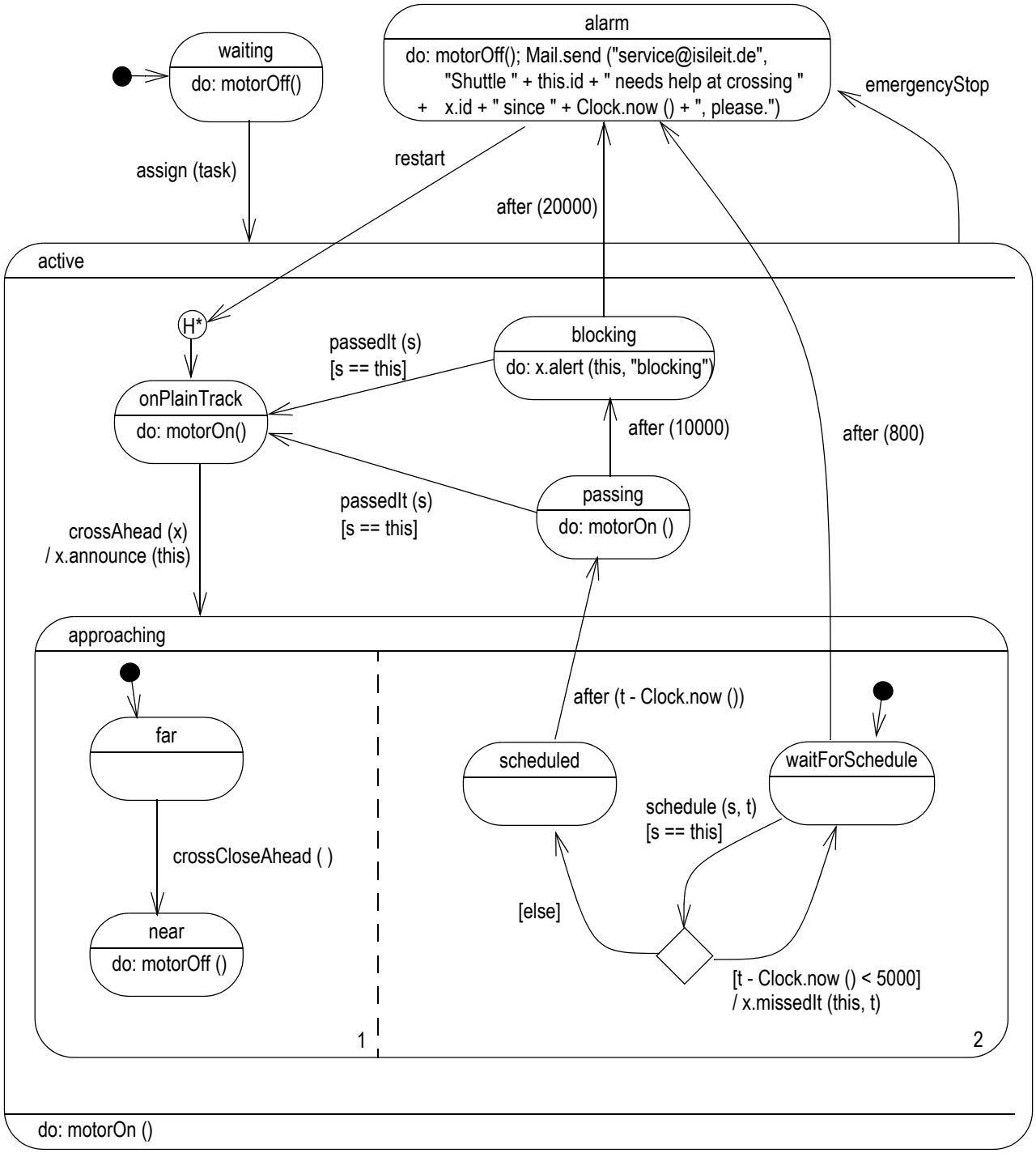
Disclaimer: our approach is *not* feasible for hard real time requirements.

Our focus lies on complex, distributed, object oriented systems where several agents communicate e.g. via a local area network. The behavior of such agents may be modeled using statecharts. Within these statecharts one may occasionally need time constraints, e.g. in order to model time-outs. Example A.91 shows a statechart employing certain time aspects that are already at the edge of our approach. The Statechart models the behavior of a shuttle traveling around in a hall and approaching a crossing. The crossing is controlled by another active object. The crossing object acts like a traffic light. Each shuttle announces its arrival and the crossing reserves a time slot when the shuttle may pass the crossing. Once the shuttle has passed the crossing, the crossing sends a passedIt event signaling that the shuttle is no longer in the crossing area.

The statechart of Example A.91 employs time at several stages. If the shuttle receives a crossAhead event in state onPlainTrack, it switches into state approaching. State approaching has two parallel substates. The first (left) substate just controls that the shuttle does not enter the crossing area without getting the OK for passing it. If the shuttle is still in state approaching and a crossCloseAhead event is received, then the shuttle switches from state far to state near and the shuttle turns its motor off, i.e. the shuttle stops. The second substate of state approaching handles the negotiation with the crossing object. The crossAhead transition from onPlainTrack to approaching already sends a signal to the crossing object x announcing the new shuttle. In state waitForSchedule the shuttle waits for the assignment of a time slot granting traversal. This may take some time due to an overload of the crossing object or due to communication problems. Note, if the shuttle approaches the crossing area while it is still waiting for the traversal grant, the shuttle is automatically stopped by the first substate of state approaching. Thus, in principle the shuttle has enough time to wait for the traversal grant. However, if the crossing does not react on an approaching shuttle within some reasonable time, then the shuttle must assume a serious communication problem or some other technical problem. Therefore, the statechart specifies a time-out of 800 milliseconds / time units after which the shuttle switches to state alarm, turns its motor off and sends a message to the service personal. This is achieved by an after transition from state waitForSchedule to state alarm. An after transition models a special kind of event that occurs if the source state of the corresponding transition has not been left for the specified amount of time. Note, if the state is left and entered again, e.g. via a self transition, the time-out of the after event is reset. However, in case of a complex state, some transitions between contained substates may fire without resetting the after event.

The crossing sends schedule events with two parameters to the shuttle(s). The first parameter s denotes the target shuttle that gets a time slot to pass the crossing and the second parameter t denotes the time, when the shuttle shall pass the crossing. In state waitForSchedule, the shuttle first ensures that itself is the target of the schedule event, cf. guard [s == this]. Second, the shuttle may need about 5000 time units to reach the crossing because it is still far away from the crossing or because it has stopped due to a crossCloseAhead event and it needs some time to restart its engine. Thus, if the scheduled passing time t is closer than 5000 time units from now, then the shuttle will not be able to pass the crossing in time. This is checked by the condition [t - Clock.now () < 5000]. Note, Clock.now is a library function that accesses the current system time. This function will be discussed below. If the scheduled time t is too close (or already passed) then the shuttle sends a missedIt event to the crossing x and switches back to state waitForSchedule. In this case the crossing shall send a new (later) schedule event. If the time t - Clock.now() suffices, the shuttle switches to state scheduled. In state scheduled the shuttle waits until the schedule time t is reached. This is modeled using an after transition with waiting time t - Clock.now (). If nothing interrupts this waiting and the time-out event occurs the shuttle switches to state passing and ensures that the motor is on.

Example A.91: A statechart with time for class Shuttle



The shuttle should be able to pass the crossing within 10000 time units. If the shuttle has passed the crossing within this time it receives a passedIt event and switches to state onPlainTrack, again. However, if the passedIt event does not occur within 10000 time units a safety critical error has occurred. There might be an unrecognized problem with the shuttle's drive, which would mean that the shuttle is still in the crossing area. This is dangerous since the crossing may have scheduled another shuttle for the next time slot. Therefore, after 10000 time units the shuttle switches to state blocking and sends an alert event to the crossing. This enables the crossing to send emergencyStop events to all shuttles that are already scheduled to pass the crossing. In case of an emergencyStop event a shuttle switches from any substate of state active to the state alarm, directly. In state blocking a shuttle waits another 20000 time units for the passedIt event. If the passedIt event occurs in time, the shuttle switches back

to state onPlainTrack and proceeds. Otherwise something went wrong and the shuttle switches to state alarm which stops the shuttle motor and sends an emergency message to the service personal.

Our statechart for the shuttle behavior seems to model the timing aspects for passing the crossing, reasonably. However, this model makes certain assumptions about the computational speed of the shuttle controller software itself and about the event transportation mechanisms and about the accuracy of the shuttle's clock that may not hold in practice. Before we discuss these problems, we first explain the notion of time employed in our formal model.

In order to be able to deal with time aspects, actively, we adapted a common simulation approach. Our scheduler employs an extra queue for timer events. The timer events stored in this queue are sorted according to the time when they will occur. In addition, our unique H_Stack object employs an attribute storing the current simulation time. This time is returned by method Clock.now(). When method h_schedule of our event scheduler has scheduled a time slot to one of the active objects in story pattern triggerhandling, then it advances the simulation time by some delta, cf. the second activity of Definition A.65 on page 175. This models that the handling of an event by an active object may consume some time for computation. Next, method h_schedule looks-up the timer event queue and raises all waiting events for which the time of occurrence has been passed. This is discussed in more detail, below.

First we discuss how timer events are created. Timer events are created during the execution of do-actions of new current states. Each time a transition has fired and the statechart has switched to some new state(s), method H_Transition::fire executes the corresponding do-actions by calling method evalCurrentDoAction on the new state, cf. Definition A.74 on page 183. As a side-effect, method evalCurrentDoAction calls method signalReady on the new state, cf. Definition A.79 on page 191 and Definition A.89 on page 205. Method signalReady is shown at the bottom of Definition A.79. As already discussed in chapter A.8.4, method signalReady first creates h_ready events for certain triggerless transitions. Second, the iterated story pattern findTimer looks-up all transitions t that leave the current state and that have a trigger starting with the string "after" followed by the name of the source state. Internally, after transitions are labeled with the keyword after followed by the name of the source state followed by an expression denoting the waiting time for this transition, e.g. "after waitForSchedule 800". At the user interface the source state name is omitted. The source state name within the transition label is used to guarantee that an after transition is fired by its own timer event, only, and not by a timer event from some other state.

For each such after transition, the iterated story pattern findTimer creates a new timer event ev. The event name is just copied from the transition trigger. In addition, the birthNo attribute of the current state is copied into the eventBirthNo attribute of ev. As discussed for triggerless transitions, this birthNo attribute is used to guarantee that the source state has not been left between the creation of the event and its consumption. Recall that method signalReady increases the birthNo attribute of the current state each time the state is visited again. Next, story pattern findTimer computes the time when the timer event shall be raised. This time is derived from the time attribute of our unique H_Stack object plus the delay time which is the third part of the trigger of the after transition, e.g. 800. The new timer event is added to the special timerQueue provided by the unique H_Stack object. In order to be able to deliver the timer event to the correct active object we create a timerEvents link attaching it to the current active object ao, which is passed as parameter to method signalReady.

Timer events are raised by method h_schedule of our unique H_Stack object, cf. Definition A.65 on page 175. First, method h_schedule chooses one active object and asks it to handle one event. In the second activity, method h_schedule advances the simulation time stored in the time attribute of the unique H_Stack object by the result of method someTimeDelta. This models that in the previous step the active object may have used some time to handle an event and to perform the corresponding com-

putations. After method h_schedule has increased the current simulation time, the raising time of some timer events may have been passed and this events should now occur. This is handled in story pattern raiseTimers of method h_schedule. Story pattern raiseTimers looks up the first element of the timerQueue and checks whether the simulation time has passed the event schedule time. In this case story pattern raiseTimers looks-up the active object ao that has raised the timer event via the timerEvents link. Story pattern raiseTimers deletes the timerQueue and the timerEvents link and adds the event to the eventQueue of the active object ao. Story pattern raiseTimers is repeated via its success transition as long as the first element of the timerQueue has passed the simulation time. Recall, the timerQueue association is ordered by the time when the contained events shall be raised.

Once the timer event is added to the eventQueue of the corresponding active object it is consumed like an ordinary event with only one exception. In method fire of class H_Transition the second story pattern checkOutdatedEvents looks for h_ready and for after events that are outdated, cf. Definition A.74 on page 183. An h_ready or after event is outdated if its source state has been left (and re-entered) between the creation of the event and its consumption. In such a case the h_ready of after event must no longer fire a transition but it must be ignored. Story pattern checkOutdatedEvents achieves this by comparing the birthNo attribute of the current state and the eventBirthNo attribute of the event and in case of a difference method fire terminates. Note, the name of the source state that created the h_ready or the after event is encoded in the event name. No other state has a transition with a similar trigger. Thus, if the original source state has been left, meanwhile, then the h_ready or after event will just not find a matching transition and it is ignored, too.

Now we are ready to discuss the time related problems of our example. First of all, our formal model does not allow to make any assumptions on how much time an active object needs to receive and to handle an event. In our formal model time proceeds due to method someTimeDelta. Method someTimeDelta is employed by method h_schedule of our unique H_Stack object and is discussed below. We have introduced method someTimeDelta in order to be able to simulate different computational speeds and different computational complexities for the handling of events.

Formally, for method someTimeDelta there is no upper limit, i.e. handling a single event may need arbitrary time. This means, the user may not assume, a minimal computational speed, not even for simple operations. This reflects the problem, that even a simple computation may take an unexpected amount of time, e.g. because the corresponding process has been swapped out of memory by the operating system or because some required services are blocked. Thus, in our model one may not rely on guaranteed computational speed but one has to model protocols that are able to deal with unexpected delays.

There is also no lower limit on the consumption of time by an event handling step, i.e. the handling of an event may even need no time at all. This reflects the problem, that the user must not assume, that only a limited amount of other things may happen within a certain amount of time. If one sets a time-out of e.g. one second during which the shuttle proceeds unattended, then at some day someone will increase the speed of that shuttle so much, that one second without attention suffices to cause a crash. Or someone will increase the computational speed of the shuttle controller such that it is able to execute some other part of the statechart that fast, that one second suffices to create enough events in order to cause a serious buffer overflow. Thus, in our formal model an arbitrary small amount of time shall suffice to execute an arbitrary number of event handling steps.

Method someTimeDelta allows us to parameterize our formal model with different time consumption models by just exchanging method someTimeDelta. For a certain application area one may easily introduce e.g. a fixed amount of time per event consumption or a random time within certain ranges or an amount of time depending on the complexity of the performed task. However, there is a basic assumption that one may rely on for any time model: finally time must go on, i.e. if one models a time-

out event using an after transition and if the source state of that after transition is never left, then finally that after event will be raised.

There is one time effect that is not yet covered by our formal model, this is the time required for the transportation of events from their creator to their receiver. In our formal model events are created by calling the corresponding event method of the target object. This creates the event and enqueues it to the event queue of the target object, immediately. In addition, our formal model employs an explicit scheduler and only one active object is actually active, at a time. This has the effect that e.g. if one active object sends two events to a second active object within a single event handling step then these two events will be enqueued at the target object in their invocation order and no third active object has a chance to send an event in between. These assumptions may not hold in practice since event transportation may need time and since active objects may be executed in parallel. However, we could incorporate such aspects by just dropping the assumption that the event queue is order preserving and by fetching events from the event queue in a random order. This relates to the aspect of prioritizing events and is future work. Alternatively, we could introduce event transportation objects simulating communication channels. All events are send to some event transportation object and this object just forwards these events to their actual destination. The event transportation objects would employ their own statecharts modeling transportation times using after events. This introduces an explicit model for event transportation times, that may be adapted for the simulation of different physical event transportation mechanisms.

Corresponding to our formal model of time our example exhibits serious design flaws. For example, the first parallel substate of state approaching in Example A.91 has the responsibility to stop the shuttle motor before the shuttle enters the crossing area if the crossing did not yet send the time slot. In state far the shuttle waits for a crossCloseAhead event and then the shuttle switches to state near and stops its motor. The problem is, that in our formal model the simulation time may proceed by an arbitrary amount between the moment when some sensor enqueues the crossCloseAhead event and the moment when our scheduler assigns the next time slot to the shuttle object and until all previous events waiting in the event queue have been consumed. In a physical setting the shuttle may have crashed into the crossing area, meanwhile. Due to the same reasons, one may not rely on the fact that an after 800 events actually occurs after 800 time units. Much more time may have been consumed.

In order to avoid such problems, one has to guarantee a certain minimal reaction time for the shuttle control software. To be able to guarantee certain reaction times one has to address several aspects. First of all, the event handling process needs to be fast enough. This means one has to compute which elementary steps are executed during the event handling and how much real time is consumed by these steps with respect to the given hardware. In simple cases such measurements can be computed or one can provide conservative upper bounds for the required time. In the general case, it may be complicated to predict the number of iterations for certain loops and the time consumption of other complex computations.

In case of a multi-process environment one also has to take care of scheduling effects of the operating system or time consumed by some hardware interrupts. The same holds for other exclusive system services like access to certain IO devices or to certain buffers or to certain memory regions. These aspects may be addressed using a single process environment or a real time operating system. Real time operating systems are able to reserve certain services and certain time slots for certain processes, cf. [Kope97]. This allows to guarantee that a process is able to perform a certain amount of operations within a given time frame.

Our formal model employs an event queue storing not yet consumed events. This creates the problem that a large number of old events may block a new event from being handled. If one active objects sends events to a second active object much faster than the second object is able to consume these

events then the event queue may grow, arbitrarily. Urgent events like emergencyStop and crossCloseAhead in our example may be recognized too late. In conventional statechart approaches this problem is attacked by handling all waiting events in parallel, cf. [HG96]. This avoids the problem of events waiting in an event queue but it creates the problem of conflicts between "simultaneous" events, as discussed in chapter A.8.2. In addition, the parallel handling of events implies the implementation requirement of handling all actions in isolation, cf. chapter A.8.2. As discussed, this is not feasible for object oriented settings. In our model one may easily avoid the problem of jammed event queues by modeling appropriate event exchange protocols. For example all senders could be restricted to send only one event per receiver at a time and to wait for an acknowledge from the receiver before it sends the next event. At the side of the receiver this would restrict the maximal number of events waiting within the event queue to the maximal number of senders. If this number is limited, too, and if one is able to guarantee a maximal time for the consumption of an event then one is able to guarantee an upper bound for the reaction to certain (urgent) events. This protection against event queue overflows requires a lot of extra modeling efforts. To reduce these extra modeling efforts one could introduce prioritized events. Prioritized events would just pass all waiting events with low priorities and would be recognized immediately. Above modeling restrictions would be needed for other prioritized events, only. This is future work.

If events have to be transported between different processors or computers via a communication system, another bunch of problems may occur. Usually, communication systems employ error correcting protocols in order to guarantee the delivery of events. This means, in case of faulty transmissions, the transmission is repeated. This may occur several times. Therefore in general, it is not possible to guarantee the delivery of events within a given time frame. However, real time communication protocols are able to detect such problems and to send failure events in case of communication problems or if they are not able to guarantee a given transportation time, cf. [Kope97]. Note, if an event transports not just a simple signal but a varying amount of complex data this may have an effect on the event transportation time, too.

Another time specific aspect of our example is the assumption of a common global time. In our example the crossing object sends a schedule event to the shuttle object that contains a time parameter t. The time parameter t tells when the shuttle shall pass the crossing. If everything works fine, the shuttle waits in state scheduled for an after (t-Clock.now ()) time-out event. Even if the event handling within the shuttle is fast enough and causes no serious delay, this operation could cause a hazard in practice. The expression t-Clock.now() computes the correct waiting time only if the crossing object and the shuttle object use the same global time. If the crossing software and the shuttle software run on different computers these computers provide their own system time based on their own clocks. If these system times differ by some seconds or minutes then the shuttle may enter the crossing too early or too late by this amount of time. This problem of clock synchronization is well known and their exists several technical solutions from using very exact clocks to a common broadcast based clock or to clock synchronizing communication protocols, cf. [Kope97].

To summarize, there are still a lot of open problems concerning a correct handling of time in the specification of a distributed concurrent system. To deal with these problems one has to consider aspects of all areas of computing from communication protocols to hardware and operating system aspects. Traditional statecharts approaches address concurrency problems through their inherent parallel handling of events. As discussed in chapter A.8.2, this parallel event handling is not feasible for object oriented applications. Therefore our approach employs sequential event handling based on event queues. However, these event queues create new problems for the guarantee of real time properties. These problems are subject of current research.

References

[ASU86] A. Aho, R. Sethi, J. Ullman: Compilers: Principles, Techniques, and Tools; Addison-Wesley, 1986.

[AT98] J. Ali, J. Tanaka: Implementation of the Dynamic Behavior of Object Oriented System; IDPT Vol. 4, 1998, Proc. of third biennial world conference on integrated design and process technology, pp. 281-288, ISSN No. 1090-9389, Society for Design and Process Science, 1998.

[Beck99] K. Beck: Extreme Programming Explained: Embrace Change; Addison-Wesley, ISBN 0-201-61641-6, 1999.

[Bee94] M. v. d. Beeck: A comparison of statecharts variants; in Langmaak, de Roever, Vytopil (eds): Formal Techniques in Real-Time and FaultTolerant Systems, LNCS 863, pp. 128-148, 1994.

[BFG95] D. Blostein, H. Fahmy, A. Grbavec: Practical Use of Graph Rewriting; in [EER95], 1995.

[Boe88] B. Boehm: A spiral model of software development and enhancement; Computer, May 1988, pp. 61-72, Reprinted in B. Boehm, Tutorial: Software Risk Management, IEEE Computer Society Press, Los Alamitos, CA, 1989.

[Boo94] G. Booch: Object-Oriented Analysis and Design; Benjamin/Cummings, Redwood City, CA., 1994.

[BRJ99] G. Booch, J. Rumbaugh, I. Jacobson: The Unified Modeling Language User Guide; Addison-Wesley, ISBN 0-201-57168-4, 1999.

[CCM92] M. Consens, I. Cruz, A. Mendelzon: Visualizing Queries and Querying Visualizations; ACM SIGMOD Record 21, No. 1, pp. 39-46, 1992.

[CD94] S. Cook, J. Daniels: Designing Object Systems (Object-Oriented modeling with Synthropy). ISBN 0-13-203860-9, Prentice Hall, 1994.

[CER78] V. Claus, H. Ehrig, G. Rozenberg (eds.): Proc. International Workshop on Graph Grammars and Their Application to Computer Science and Biology; LNCS 73, Springer, Berlin, 1978.

[CJJÖ93] M. Christerson, I. Jacobson, P. Jonsson, G. Övergaard: Object-Oriented Software Engineering - A use-case Driven Approach; Addison-Wesley, New York, 1993.

[Cre00] K. Cremer: Graphbasierte Werkzeuge zum Reverse Engineering und Re-engineering; PhD RWTH-Aachen, 1999, Wiesbaden, Deutscher Universitätsverlag, 2000.

[Doug98] B. P. Douglass: Real Time UML; Addison-Wesley, ISBN 0-201-32579-9, 1998.

[EER95] H. Ehrig, G. Engels, G. Rozenberg: Proc. 5th International Workshop on Graph Grammars and Their Application to Computer Science; LNCS 1017, Springer, Berlin, 1995.

[EK91] H. Ehrig, H.-J. Kreowski (eds.): Proc. 4th International Workshop on Graph Grammars and Their Application to Computer Science; LNCS 532, Springer, Berlin, 1991.

[ELNSS92] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, A. Schürr: Building Integrated Software Development Environments Part I: Tool Specification; ACM Trans. on Software Engineering and Methodology 1, no. 2, pp. 135-167, 1992.

[ELS86] G. Engels, C. Lewerentz, W. Schäfer: Graph Grammar Engineering: A Software Specification Method; in [ENRR86], pp. 186-201, 1986.

[Emm00] W. Emmerich: Engineering Distributed Objects; Wiley, 2000.

[ENR82] H. Ehrig, M. Nagl, G. Rozenberg (eds.): Proc. 2nd International Workshop on Graph Grammars and Their Application to Computer Science; LNCS 153, Springer, Berlin, 1982.

[ENRR86] H. Ehrig, M. Nagl, G. Rozenberg, A. Rosenfeld (eds.): Proc. 3rd International Workshop on Graph Grammars and Their Application to Computer Science; LNCS 291, Springer, Berlin, 1986.

[FFFL97] M. Feather, S. Fickas, A. Finkelstein, A. Lamswerde: Requirements and specification exemplars; Automated Software Engineering, 1997.

[FNT98] T. Fischer, J. Niere, L. Torunski: Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling; Master Thesis (in German), University of Paderborn, Germany, 1998.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, A. Zündorf: Story Diagrams: A new Graph Grammar Language based on the Unified Modeling Language; Proc. 6th International Workshop on Theory and Application of Graph Transformation (TAGT '98), LNCS 1764, Springer, 1998.

[Fujaba] The Fujaba home page: http://www.fujaba.de/

[GGN91] H. Göttler, J. Günther, G. Nieskens: Use Graph Grammars to Design CAD-Systems; in [EK91], pp. 396-410, 1990.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Pattern (Elements of Reusable Object-Oriented Software); Addison-Wesley, ISBN 0-201-63361-2, 1995.

[GK97] U. Glässer, R. Karges: Abstract State Machine Semantics of SDL; Journal of Universal Computer Science, Vol. 3, No. 12, 1997.

[Hal90] M. Hallmann: Prototyping of complex Software Systems; (in German) Teubner-Verlag, Stuttgart, 1990.

[Har87] D. Harrel: Statecharts: A visual formalism for complex systems; Science of Programming, Vol. 8, pp. 231-274, 1987.

[Har96] D. Harrel: The Statemate Semantics of Statecharts; ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 4, pp. 293-333, 1996.

[HG96] D. Harrel, E. Gery: Executable Object Modeling with Statecharts; Proc. 18th International Conference on Software Engineering (ICSE '96), Berlin, Germany, pp. 246-257, IEEE, SIGSOFT, ISBN 0-8186-7246-3, acm press, 1996.

[HJKW96] P. Heimann, G. Joeris, C. Krapp, B. Westfechtel: DYNAMITE: Dynamic Task Nets for Software Process Management; Proc. 18th International Conference on Software Engineering (ICSE '96), Berlin, Germany, pp. 331-341, acm press, March 1996.

[HPSS87] D. Harrel, A. Pnueli, J. Schmidt, R. Sherman: On the formal semantics of statecharts; Proc. 2nd LICS, pp. 54-64, Springer, 1987.

[HW95] R. Heckel, A. Wagner: Ensuring consistency of conditional graph grammars - a constructive approach; Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Vol. 2 of Electronic Notes in TCS, http://www.elsevier.nl/gej-ng/31/29/23/27/23/show/Products/notes/index.htt, 1995.

[H+90] D. Harrel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Tauring, M. Trakhtenbrot: STATEMATE: A Working Environment for the Development of Complex Reactive Systems; IEEE Trans. Soft. Eng., Vol. 16, pp. 403-414, 1990.

[ITU96] ITU-T Recommendation Z.100, Specification and Description Language (SDL); International Telecommunication Union (ITU), Geneva, 1994 + Addendum 1996.

[Java] http://java.sun.com/

[JBR99] I. Jacobson, G. Booch, J. Rumbaugh: The Unified Software Development Process; Addison-Wesley, ISBN 0-201-57169-2, 1999.

[Jeckel] UML tool comparison page http://www.jeckel.de/umltools.htm

[JFC99] Technical reference of the Java Foundation Classes, contained in the Java Development Kit v1.2 (Java2) http://www.sun.java.com/

[JSWZ01] J.-H. Jahnke, W. Schäfer, J. Wadsack, A. Zündorf: Managing inconsistency in evolutionary database reengineering processes; accepted for Science of Computer Programming, Elsevier. (to appear)

[JSZ96] J.-H. Jahnke, W. Schäfer, A. Zündorf: A Design Environment for Migrating Relational to Object Oriented Database Systems; Proc. of the 1996 International Conference on Software Maintenance (ICSM '96), IEEE Computer Society Press, 1996.

[JSZ97] J.-H. Jahnke, W. Schäfer, A. Zündorf: An experiment in building a light-weight process-centered environment supporting team software processes; Software Process Improvement and Practice, Vol. 3, issue 3, John Wiley & Sons, 1997.

[JSZ97b] J.-H. Jahnke, W. Schäfer, A. Zündorf: Generic Fuzzy Reasoning Nets as a basis for reverse engineering relational database applications; Proc. of European Software Engineering Conference (ESEC/FSE '97), September 1997, Zürich, Springer, LNCS 1301, 1997.

[JSZ97c] J.-H. Jahnke, W. Schäfer, A. Zündorf: A Design Environment for Migrating Relational to Object Oriented Database Systems (Abstract); in: N. Barghouti, K. Dittrich, D. Maier, W. Schäfer (eds.): Software Engineering and Database Technology, Dagstuhl-Seminar-Report 173, March 17th - 21st, 1997.

[JZ97] J.-H. Jahnke, A. Zündorf: Rewriting poor Design Patterns by good Design Patterns; in: S. Demeyer, H. Gall (eds.): Proc. ESEC/FSE '97 Workshop on Object-Oriented Reengineering, Technical Report TUV-1841-97-10, Technical University of Vienna, Information Systems Institute, Argentinierstrasse 8/184-1, A-1040 Wien, Austria, 1997.

[JZ98] J.-H. Jahnke, A. Zündorf: Specification and Implementation of a Distributed Planning and Information System for Courses based on Story Driven Modeling; Proc. of the Ninth International Workshop on Software Specification and Design, April 16th-18th, Ise-Shima, Japan, IEEE Computer Society Press, pp. 77-86, ISBN 0-8186-8439-9, 1998.

[JZ98b] J.-H. Jahnke, A. Zündorf: Using Graph Grammars for Building the Varlet Database Reverse Engineering Environment; Proc. of TAGT '98 - 6th International Workshop on Theory and Application of Graph Transformations, Paderborn, Nov. 16th - 20th, Technical Report, tr-ri-98-201, University of Paderborn, Germany, 1998.

[JZ99] J.-H. Jahnke, A. Zündorf: Applying Graph Transformations To Database Re-Engineering; Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2 - Applications, World Scientific, Singapore, 1999.

[Klein99] T. Klein: Reconstruction of UML activity and collaboration diagrams from Java Code; Master Thesis (in German), University of Paderborn, Germany, 1999.

[KMST94] K. Koskimies, T. Männistö, T. Systä, J. Tuomi: SCED - An environment for dynamic modeling in object-oriented software construction; Proc. Nordic Workshop on Programming Environment Research '94

(NWPER '94), Lund, Department of Computer Science, Lund Institute of Technology, Lund University, pp. 217-230, June 1994.

[KNNZ99] H.-J. Köhler, U. Nickel, J. Niere, A. Zündorf: Using UML as visual programming language; Technical Report tr-ri-99-205, University of Paderborn, Germany, 1999.

[KNNZ00] H.-J. Köhler, U. Nickel, J. Niere, A. Zündorf: Integrating UML Diagrams for Production Control Systems; Proc. 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, pp. 241-251, acm press, 2000.

[KNNZ00b] T. Klein, U. Nickel, J. Niere, A. Zündorf: From UML to Java and back again; Technical Report tr-ri-00-216, University of Paderborn, Germany, 2000.

[Köhl99] H.-J. Köhler: Code Generation for UML Collaboration, Sequence, and Statechart Diagrams; Master Thesis (in German), University of Paderborn, Germany, 1999.

[Kope97] H. Kopetz: Real-Time Systems - Design Principles for Distributed Embedded Applications; Kluwer Academic Publishers, Boston, ISBN 0-7923-9894-7, 1997.

[KSZ96] P. Klein, A. Schürr, A. Zündorf: Generating Single Document Processing Tools; Chapter 4.4 in: [Nagl96], 1996.

[LV00] J. Larrosa, G. Valiente: Graph Pattern Matching using Constraint Satisfaction; Proc. Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems, pp. 189-196,Berlin, Germany, 2000.

[Meye97] B. Meyer: Object-Oriented Software Construction; Prentice Hall Professional Technical Reference, ISBN 0-13-629155-4, 1997.

[MS00] E. Mäkinen, T. Systä: Implementing Minimally Adequate Synthesizer, University of Tampere, Dept. of Computer and Information Sciences, Report A-2000-9, 2000.

[MS01] E. Mäkinen, T. Systä: MAS – An Interactive Synthesizer to Support Behavioral Modeling in UML; Proc. 23rd International Conference on Software Engineering (ICSE 2001), Toronto, Canada, acm press, 2001.

[Nagl96] M. Nagl (ed.): Building Thightly-Integrated (Software) Development Environments: The IPSEN Approach; LNCS 1170, Springer, 1996.

[NNSZ99] U. Nickel, J. Niere, W. Schäfer, A. Zündorf: Combining Statecharts and Collaboration Diagrams for the Development of Production Control Systems; Proc. of OMER Workshop, Technical Report 1999-01, University of Armed Forces, München, 1999.

[NNZ00] U. Nickel, J. Niere, A. Zündorf: The Fujaba Environment; Proc. 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, pp. 742-745, acm press, 2000.

[NWZ01] J. Niere, J. Wadsack, A. Zündorf: Recovering UML Diagrams from Java Code using Patterns; Proc. 2nd Workshop on Soft Computing Applied to Software Engineering, Enschede, The Netherlands, LNCS, Springer. (to appear)

[NZ00] J. Niere, A. Zündorf: Using Fujaba for the Development of Production Control Systems; Proc. AGTIVE '99, LNCS 1779, pp. 181-191, Springer, 2000.

[NZ00b] J. Niere, A. Zündorf: Tool demonstration: Testing and Simulating Production Control Systems Using the Fujaba Environment; Proc. International Workshop and Symposium on Applications Of Graph Transformations With Industrial Relevance (AGTIVE), Kerkrade, The Netherlands, LNCS 1779, pp. 449-456, Springer, 2000.

[Ous94] J. Ousterhout: Tcl and the Tk Toolkit; Addison-Wesley, Reading, MA, 1994.

[RBPEL91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: Object–Oriented Modeling and Design; Prentice Hall, Englewood Cliffs, N. J. 07632, 1991.

[RG92] K. S. Rubin, A. Goldberg; Object Behavior Analysis; Comunications of the ACM, Vol. 35, No. 9, pp. 48-62, Springer, 1992.

[Rhap] The Rhapsody case tool reference manual; Version 1.2.1, ILogix, http://www.ilogix.com/

[RJB99] J. Rumbaugh, I. Jacobson, G. Booch: The Unified Modeling Language Reference Manual; Addison-Wesley, ISBN 0-201-30998-X, 1999.

[Rock00] I. Rockel: Merging Graph-Like Object Structures as basis for Configuration Management; Master Thesis (in German), University of Paderborn, Germany, 2000.

[Roz97] G. Rozenberg (ed.): Handbook of Graph Grammars and Computing by Graph Transformation; Vol. 1, Singapore, World Scientific Publ. Co. Pte. Ltd., ISBN 9810228848, 1997.

[RR-RT] The Rational-Rose Realtime CASE-tool, http://www.rational.com

[RT97] J. Rosenhäger, M. Tubbesing: Konzeption und Realisierung des Stundenplanerfassungs- und -planungssystems; Master Thesis (in German), University of Paderborn, Germany, 1997.

[Rud97] M. Rudolf: Konzeption und Implementierung eines Interpreters für attributierte Graphtransformation; Master Thesis (in German), TU Berlin, Germany, 1997.

[San99] T. Sander: Tool Support for the Design and Generation of Agent Systems; Master Thesis (in German), University of Paderborn, Germany, 1999.

[SB91] G. Schmidt, R. Berghammer (eds.): Graph-Theoretic Concepts in Computer Science; Proc. WG '91, LNCS 570, Springer, 1991.

[Sced] The Sced environment; http://www.cs.tut.fi/~tsysta/sced/

[Sch91] A. Schürr: Operational Specification with Programmed Graph Rewriting Systems; PhD Thesis (in German), Deutscher Universitäts-Verlag, Wiesbaden, 1991.

[Sch91b] A. Schürr: PROGRESS: A VHL-Language Based on Graph Grammars; in [EK91], pp. 641-659, 1991.

[SGW94] B. Selic, G. Gullekson, P. Ward: Real-Time Object-Oriented Modeling, John Wiley & Sons, New York, NY, 1994.

[SL95] A. Stepanov, M. Lee: The Standard Template Library; Technical Report, Hewlett Packard Laboratories, Palo Alto, cf. http://www-leland.stanford.edu/~iburrell/cpp/stl.html, 1995.

[Sne96] G. Snelting: Reengineering of Configurations Based on Mathematical Concept Analysis; ACM Transactions on Software Engineering and Methodology 5(2), pp. 146-189, April 1996.

[SWZ95] A. Schürr, A. Winter, A. Zündorf: Graph grammar engineering with PROGRES; in W. Schäfer, P. Botella (eds.): European Software Engineering Conference (ESEC '95), Barcelona, Spain, LNCS 989, pp. 219-234, Springer, 1995.

[SWZ95b] A. Schürr, A. Winter, A. Zündorf: Visual Programming with Graph Rewriting Systems; Proc. 11th International IEEE Symp. on Visual Languages (VL '95), Darmstadt, Germany, IEEE Computer Society Press, 1995.

[SWZ95c] A. Schürr, A. Winter, A. Zündorf: Specification and Prototyping of graph based Systems; (in German: Spezifikation und Prototyping graphbasierter Systeme) Proc. STT '95, Braunschweig, Germany, 1995.

[SWZ96] A. Schürr, A. Winter, A. Zündorf: Specification and Prototyping of graph based Systems; (in German: Spezifikation und Prototyping graphbasierter Systeme) extended and improved version of [SWZ95c], Informatik - F&E, Berlin, Vol. 11, No. 4, pp. 191-202, Springer, 1996.

[SWZ96b] A. Schürr, A. Winter, A. Zündorf: Developing Tools with the PROGRES Environment; Chapter 3.6 in [Nagl96], pp. 356-369, 1996.

[SWZ99] A. Schürr, A. Winter, A. Zündorf: PROGRES: Language and Environment; in: G. Rozenberg (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2 - Applications, World Scientific, Singapore, 1999.

[Sys97] T. Systä: Automated support for constructing OMT scenarios and state diagrams in SCED; Licentiate's Thesis, University of Tampere, Finland, Report A-1997-8, 1997.

[SZ91] A. Schürr, A. Zündorf: Nondeterministic Control Structures for Graph Rewriting Systems; Proc. Workshop in Graph-Theoretic Concepts in Computer Science (WG '91), LNCS 570, pp. 48-62, Springer, 1992; also: Technical Report AIB 91-17, RWTH Aachen, Germany, 1991.

[SZ96] A. Schürr, A. Zündorf: Specification of Logical Documents and Tools; Chapter 3.3 in [Nagl96], pp. 297-323, 1996.

[SZ99] W. Schäfer, A. Zündorf: Round-Trip Engineering with Design Patterns, UML, Java, and C++; Proc. 21st International Conference on Software Engineering (ICSE '99), Los Angeles, California, USA, IEEE Computer Society Press, 1999.

[SZ99b] W. Schäfer, A. Zündorf: Round-Trip Engineering with Design Patterns, UML, Java, and C++; European Software Engineering Conference (ESEC '99), Tolouse, France, 1999.

[SZ00] W. Schäfer, A. Zündorf: Methods and Tools for Round-Trip Engineering with UML, Design Patterns, Java and C++; Foundations of Software Engineering (FSE 8), SIGSOFT 2000, San Diego, California, USA, 2000.

[Toge] TogetherJ: http://www.togethersoft.com/

[UML97] UML Notation Guide vers. 1.3. OMG, Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjectTime, Platinum Technology, Ptech, Taskon, Reich Technologies, Softeam, 1997.

[ZSW99] A. Zündorf, A. Schürr, A. Winter: Story Driven Modeling; Technical Report tr-ri-99-211, University of Paderborn, Germany, 1999.

[Zü89] A. Zündorf: Control Structures for the Specification Language PROGRES; Master Thesis (in German), RWTH Aachen, Germany, 1989.

[Zü92] A. Zündorf: Implementation of the Imperative/Rule Based Language PROGRES; Technical Report AIB 92-38, RWTH Aachen, Germany, 1992.

[Zü93] A. Zündorf: A Heuristic Solution for the (Sub-) Graph Isomorphism Problem in Executing PROGRES; Technical Report AIB 93-5, RWTH Aachen, Germany, 1993.

[Zü94] A. Zündorf: Graph Pattern Matching in PROGRES; in [EER95], 1994.

[Zü96] A. Zündorf: A Development Environment for PROgrammed GRaphrEwritingSystems - Specification, Implementation, and Usage; Ph. D. Thesis (in German), RWTH Aachen, Germany, 1995, Vieweg, 1996.

[Zü96b] A. Zündorf: Graph Pattern Matching in PROGRES; in: J. Cuny, H. Ehrig, G. Engels, G. Rozenberg (eds.): Proc. 5th. International Workshop on Graph-Grammars and their Application to Computer Science, Williamsburg, November 1994, LNCS 1073, Springer, 1996.

[Zü96c] A. Zündorf: Reengineering of Databases using Triple Graph Grammars (Abstract); in: H. Ehrig, U. Montanari, G. Rozenberg, J. Schneider (eds.): Graph Transformations in Computer Science, Dagstuhl-Seminar-Report 155, September 9th - 13th, 1996.

[Zü00] A. Zündorf: Graph Grammar Engineering; slide script to graph grammar engineering course in winter 1999/2000, University of Paderborn, http://www.upb.de/cs/zuendorf.html.

[Zü01] A. Zündorf: From Use-Cases to Code - Rigorous Software Development with UML; Proc. 23rd International Conference on Software Engineering (ICSE 2001), Toronto, Canada, acm press, 2001.

[Zü01b] A. Zündorf: From Use-Cases to Code - Rigorous Software Development with UML; accepted for the European Conference on Software Engineering, ESEC/FSE 2001, Wien, Austria, 2001.