UML-Based Specification of State-Oriented Real-Time Properties

Dissertation

A thesis submitted to the
Faculty of Computer Science, Mathematics and Electrical Engineering
of the Universität Paderborn in partial fulfillment
of the requirements for the degree of Dr. rer. nat.

by

Stephan Flake

Paderborn, December 2003

Supervisors:

1. Prof. Dr. rer. nat. Franz J. Rammig, Universität Paderborn
2. Prof. Dr. rer. nat. Gregor Engels, Universität Paderborn
3. Prof. Dr. rer. nat. Martin Gogolla, Universität Bremen

Date of public examination: December 19, 2003

Dedicated to my family

Abstract

In recent years, the Unified Modeling Language (UML) has received significant attention by software designers to model object-oriented software systems. Complementary to UML diagrams, modelers can make use of the textual Object Constraint Language (OCL) to specify additional constraints for their models. OCL is particularly used to formulate constraints over a given UML model in form of class invariants and operation pre- and postconditions. However, the semantics of OCL is still incomplete, even in the latest OCL 2.0 proposal that has recently been adopted by the Object Management Group. While it is allowed to make use of states from State Diagrams in OCL expressions to reason about their activations, there is currently no corresponding semantics defined in the adopted OCL 2.0 specification. As a first goal, this thesis closes that gap and provides a formal notion of state configurations over UML State Diagrams that is integrated into the formal semantics of the adopted OCL 2.0 specification.

The second major goal of this thesis is to extend OCL to support specification and analysis of temporal state-oriented constraints for UML models of time-critical software-controlled systems. In order to demonstrate the applicability of this extension, the thesis focuses on early stages of the software development process and applies time-bounded state-oriented OCL constraints to specify requirements in the domain of modeling time-constrained manufacturing systems. For the structural modeling of manufacturing systems a restricted version of UML Class Diagrams is employed, and for the behavioral modeling a timed UML State Diagram variant with a corresponding time-related semantics is presented.

In addition, this thesis also provides a semantics for both the regarded kind of UML models and the time-bounded state-oriented OCL extension by mappings to formal target languages, i.e., time-annotated state-transition systems and temporal logics. This allows to perform automated formal analysis with existing verification tools. The motivation behind the mapping is the idea to abstract from the rather cryptical input languages of verification tools, in particular the temporal logics used in the formal verification technique called model checking.

Contents

Chapter 1

Introduction

I'm writing a book.
I have the page numbers down...
I just have to fill in the rest.
– Steven Wright

Developing software systems is a difficult and error-prone task. Nowadays, software is still in almost all cases developed in a rather pragmatic way. In the software design process, different software development phases are usually identified. Basically, they can be separated into phases like informal requirement gathering, analysis, specification, design, implementation, and testing. These phases are not strictly sequential but rather overlapping, iterative, or carried out in parallel for different parts of the system under development.

While some approaches focus on the implementation of a system already in early phases of development (e.g., eXtreme programming [Bec00]), others try to separate system modeling from the actual task of implementation. Latter approaches mostly use a *platform-independent model* for the complete design of an application. One of the most popular approaches in this context is the *Model-Driven Architecture* (MDA) by the Object Management Group [OMG] that bases upon industrial modeling standards like the Unified Modeling Language (UML), Meta Object Facility (MOF), and XML Metadata Interchange (XMI). From these standards, UML provides means to build object-oriented models of a system under consideration in form of a rich set of standardized graphical diagrams.

UML. UML unifies a number of different modeling languages and is still undergoing a development under the control of the OMG consortium. At the time of writing this thesis, the latest official version released by the OMG is UML 1.5, published in March 2003 [OMG03d]. More recently, the OMG adopted a number of proposals to build a new version of UML, i.e., UML 2.0. These proposals are still undergoing a finalization process. In the context of this thesis, the adopted proposal for a new version of the Object Constraint Language [OMG03b] is of particular interest, while the proposals for a UML 2.0 Infrastructure [OMG03e] and a UML 2.0 Superstructure [OMG03f] have less impact on this work.

UML defines a number of diagrams to model different aspects of the structure and behavior of software systems. For example, Class Diagrams are used to describe the static structure of a system, while UML State Diagrams model the (reactive) behavior of objects. In addition to the set of diagrams, the textual Object Constraint Language (OCL) is an integral part of the UML to specify restrictions on values of parts of UML models. Basically, OCL constraints are invariants attached to classes or pre- and postconditions of operations. Significant parts of OCL have already been formally defined in [Ric01] in form of a set-theoretic *object model*. That work heavily influenced the formal semantics of the adopted OCL 2.0 specification [OMG03b]. However, the formal semantics of OCL is still incomplete, as it currently lacks an integration of UML State Diagrams, although it is already possible to formulate constraints that refer to the states specified in UML State Diagrams. One aspect of this thesis is to extend the formal semantics of OCL by a formal integration of UML State Diagrams and to provide the formal semantics for state-related OCL operations.

UML has already been applied in different application domains, e.g., to model *time-critical* software-controlled systems such as embedded real-time systems [Dou00]. For time-critical systems, correct time-constrained behavior is an essential requirement to meet, e.g., timing bounds for message delays and progress of system execution. In this context, it is desirable to be able to identify improper behavior w.r.t. these *timing requirements* already in early phases of development. Otherwise, overall goals like meeting project deadlines and adherence to estimated costs may fail due to the need of time-consuming and expensive re-designs at a later stage of development.

The current version 1.5 of UML as well as the corresponding, recently adopted UML 2.0 proposals provide some basic means to specify timing requirements. In particular, timing annotations can be applied in Sequence Diagrams to specify timing bounds for durations of message transfers and replies to messages sent. However, UML does not have an inherent timing model, as it is designed to be a general modeling language with a focus on software systems, such that these means are not well integrated into the core concepts of UML.

Properties concerning the temporal behavior, such as safety or liveness constraints [Lam77], cannot be expressed with standard UML means. Different approaches have already introduced corresponding extensions, in particular, extensions of UML Sequence Diagrams to enhance time-bounded specifications of communication flow among objects have been published in [DK01, FHD$^+$99, KMR02]. In contrast, this thesis introduces a consistent temporal extension of the textual constraint language OCL and focuses on specification of time-bounded state-oriented constraints to reason about the time-critical progress of system execution.

OCL constraints do not make sense without a given model to refer to. In order to have a corresponding timed UML model to refer to, this thesis introduces a timed UML State Diagram variant for behavioral modeling. This State Diagram variant supports a set of UML model elements that have so far not been considered in related work on timed UML State Diagrams (cf. [EW00, DM01, KMR02]). In particular, the presented timed UML State Diagram variant preserves UML model elements like interlevel transitions, synchronous and asynchronous event communication, elapsed time events, and activities that have a notable duration.

MFERT. To validate the applicability of the temporal state-oriented OCL extension, the domain of modeling manufacturing systems is chosen. Manufacturing systems are time-critical w.r.t. production flow, i.e., production progress is time-bounded and corresponding deadlines have to be met.

This thesis builds upon an existing graphical notation for modeling manufacturing systems, i.e., MFERT.[1] MFERT is a general description scheme for modeling in the domain of manufacturing systems [DW93, Sch96, DW97]. It has been successfully applied in different projects with various industrial partners and is acknowledged by the German science award of logistics. Similar to Petri Nets, the structure of an MFERT model is a bipartite graph of nodes that represent either production processes or storages for production elements. Directed edges between nodes denote the flow of production elements.

Different variants of MFERT have already been investigated. Based on Schneider's functional general description scheme [Sch96], Holtkamp defined and implemented a corresponding framework of C++ functions [Hol99] and Quintanilla de Simsek presented a formal verification approach for MFERT models [Qui01]. However, Quintanilla de Simsek defined the graphical notation to model system behavior from scratch and did not consider explicit time, whereas this thesis builds upon UML concepts and employs a notion of time for MFERT models as well as corresponding requirements. In particular, this theses defines the structural elements of MFERT as stereotypes in a UML Profile and employs the timed UML State Diagram variant mentioned above for behavioral modeling. Through this approach, OCL constraints can be directly applied to MFERT models.

Concerning the evolution of time – especially w.r.t. the dynamic semantics of the timed variant of UML State Diagrams – a discrete underlying timing model is considered to be sufficient for the chosen modeling approach, as a broad range of manufacturing system behavior can be described based on message exchange by discrete events. For manufacturing processes (such as milling or drilling), we assume a finite time duration x or finite duration interval $[x, y]$. The basic discrete timing unit has to be chosen to be precise enough to represent the actual physical time.

Formal Verification by Model Checking. Nowadays, simulation and testing is still frequently applied to validate the correct behavior of time-dependent software systems. But due to the increasing complexity of software systems, it is getting more and more difficult to identify and examine all possible execution paths during the design process, e.g., by simulation, as the state space grows exponentially with the number of inputs and internal states. On the other hand, in recent years different formal verification methods, e.g., equivalence checking, model checking, and SAT solving, have been successfully applied to *formally verify* the correctness of hardware and software designs.

In particular, model checking has been successfully applied to formally verify digital circuits and communication protocols. Model checking is due to the work by Clarke and Emerson [CE81] and takes a set of finite state machines (the model) and a set of temporal logics formulae (the properties to fulfill) as its input. Then – and this is the most remarkable advantage of model checking – the task of verifying a model over the specified properties is fully automated, i.e., a model checking tool lists for each property whether it is true or false for the given model. Moreover, a model checking tool typically generates a counter example in cases when the model

[1]MFERT is an acronym for ”Modell der FERTigung” (German for: Model of Manufacturing).

does not satisfy a property. A counter example demonstrates an execution of the model that leads to a situation which falsifies the property. This is very helpful for detailed error analysis. One of the most popular model checking tools is SMV[2], but that model checker does not support explicit modeling of time. Nevertheless, there are also model checkers that allow models and property specifications with explicit time, e.g., UPPAAL[3] and RAVEN (Real-Time Analyzing and Verification ENvironment) [Ruf01]. RAVEN also has algorithms to perform timing and data analysis, i.e., not only requirements with yes/no answers can be checked, but also minimal and maximal execution times or data values can be determined.

While formal verification by model checking is a helpful method to formally and automatically verify a model over specified properties, widespread acceptance is still not achieved. One reason is that model checking has to explore the complete state-space of the model, and the resulting structure to investigate requires a lot of run-time computer memory (due to the so-called *state explosion problem*). As a consequence, model checking tools have to build and use an efficient symbolic internal representation to cope with the huge overall state space, e.g., SMV and RAVEN make use of a symbolic model representation by means of so-called Binary Decision Diagrams (BDDs) [Bry86, BCM$^+$90]. Enhanced approaches to speed up state space exploration and different reduction techniques are often applied to leave out and abstract from parts of the model which are not needed to prove certain properties. However, automated support for these reduction techniques is limited.

Moreover, it is often difficult for system designers, software engineers, and programmers to formulate required properties in an unambiguous formal way, as they are usually not trained in temporal logics. Thus, it is hard for them to read, understand, or even formulate temporal formulae. Several approaches have already been regarded to overcome these problems. E.g., in order to abstract from temporal logics for property specification, timing diagrams or structured natural language are applied [DKM$^+$94, JMM$^+$99]. Other approaches consider catalogues of patterns, as practice has shown that the entire expressive power of temporal logics is not needed [DAC98a]. Unfortunately, acceptance of these approaches lacks due to a missing modeling standard to base upon. An important aspect of the temporal OCL extension introduced in this thesis is that it has the expressive power to formulate all of those properties that are listed in the catalogue of specification patterns by Dwyer, Avrunin and Corbett [DAC98a].

Scope. This thesis introduces an approach to specify state-oriented real-time system properties based on concepts of the UML, especially its textual expression language OCL. With an extension that is consistent with common OCL concepts, we abstract from temporal logics formulae that are usually applied for property specifications.

For a concise, well-defined modeling language, we consider the domain of time-critical manufacturing systems and define a restricted UML-based version of the graphical MFERT notation. On the one hand, the structural elements of MFERT are embedded into the general concepts of UML by means of a UML Profile and a timed variant of UML State Diagrams is employed. On the other hand, as UML provides a vast variety of additional modeling elements, the set of regarded UML model elements is restricted by additional validation constraints. MFERT

[2]http://www-2.cs.cmu.edu/ modelcheck/smv.html
[3]http://www.uppaal.com

models that comply to these constraints are translated to the formal language of *I/O-Interval Structures* that constitutes the input language of the RAVEN model checker.

In order to develop a temporal state-oriented extension based upon standard OCL, the formal semantics definition of OCL has first to be equipped with a notion of state configurations over State Diagrams. The set-theoretic object model of Richters [Ric01] is extended correspondingly and a formal semantics for static and temporal state-related OCL operations is defined by means of interpretation functions over this extended object model. Additionally, state-oriented real-time OCL constraints are mapped to a time-annotated temporal logics called *CCTL* (Clocked Computation Tree Logic) that constitutes the property specification language of the RAVEN model checker.

Thus, a formal semantics of state-oriented real-time OCL constraints over MFERT models is established by the formal relation of temporal CCTL formulae and I/O-Interval Structures. However, note that this thesis focuses on modeling and specification of time-related system properties, such that it is not in the scope of this thesis to perform optimizations for verification purposes, e.g., to overcome the state explosion problem. This topic is addressed in Zabel's diploma thesis [Zab03].

A case study in the domain of manufacturing systems with automated guided vehicles is used as a running example throughout this thesis. The scenario description is given in Section 1.2.

1.1 Research Goals and Contributions

The overall goal of this thesis is to develop a formalization of parts of UML together with a constraint language to enable modelers to specify required system properties in a more practical way compared to pure temporal logics formulae. A focus is put on time-critical manufacturing systems and their time-bounded behavioral properties. In particular, this thesis addresses the following issues:

- An underlying general formal model for a part of the UML, namely Class Diagrams and State Diagrams, is defined. We denote this model as *extended object model*.

- Based upon the extended object model, the formal semantics of the OCL operation `oclInState()` that is part of the OCL Standard Library of operations is defined.

- A timed UML State Diagram variant is presented and a formal semantics is given to that notation by a translation to timed finite state machines, i.e., I/O-Interval Structures.

- A UML Profile for the structural elements of MFERT is introduced. Additional validation constraints restrict the rich set of UML model elements that are considered for translation into the formal language of I/O-Interval Structures. In particular, the timed UML State Diagram variant is employed for behavioral modeling of production processes.

- A temporal extension of OCL is presented that is consistent with existing OCL concepts and enables modelers to specify state-oriented time-bounded properties. The semantics of the temporal OCL extension is defined over *traces*, i.e., sequences of system states over (instantiations of) the extended object model.

Temporal OCL expressions can also be directly mapped to time-annotated temporal logics formulae in CCTL. This establishes a formal relationship between temporal OCL constraints and the considered UML-based MFERT models, as CCTL formulae have a formal semantics over I/O-Interval Structures.

- In conformance with the latest attempts to interpret OCL as a more general expression and query language, an OCL extension towards specification of *timing and data analysis queries* is introduced.

- A case study in the domain of manufacturing systems validates the applicability of the approach.

The chapters are based upon publications of recent years. The overall approach has been first presented in [FMPR01]. More recently, an enhanced version of the general design process has been presented in [Fla03a]. In [FGM$^+$01], the application within a framework focusing on user-centered design was outlined. An informal description of the considered part of MFERT and an outline of a mapping to the target language of I/O-Interval Structures was published in [FMPR00].

A formal semantics for state-oriented OCL operations was published in [FM03c]. In this context, a redefinition of the OCL 2.0 Standard Library w.r.t. the representation of OCL types on the level of UML user models was proposed in [Fla04].

Based on our experiences from property specifications in structured language by means of patterns [FMR00], a more general approach in the style of a programming language – leading to the temporal OCL extension – was presented in [FM02d, FM02c]. In addition, [FM03a] shows that the temporal OCL extension covers all specification patterns that are considered as relevant w.r.t. the pattern library by Dwyer, Avrunin and Corbett [DAC98a].

In [FM02b], two UML Profiles were presented to demonstrate how the considered extensions can be consistently applied to the UML metamodel. An extended version of this article has recently been published in a special issue of the Journal of Software and Systems Modeling (SoSyM) [FM03b].

1.2 Example: Manufacturing Case Study

A manufacturing system case study introduced by the IMS Initiative as a test case [WHS94] is applied as a running example throughout the remainder of this thesis. It is composed of a set of different manufacturing stations and a transport system as it is illustrated by the virtual 3D model in Figure 1.1.[4]

The different manufacturing stations transform items, e.g., by milling, drilling, or washing. An input buffer at each station can keep up to 3 items before they are actually transformed. Similarly, output buffers keep up to 3 items to be picked up by AGVs for further transport.

[4]Details about this 3D Animation can be found in [BFMW00, FM01].

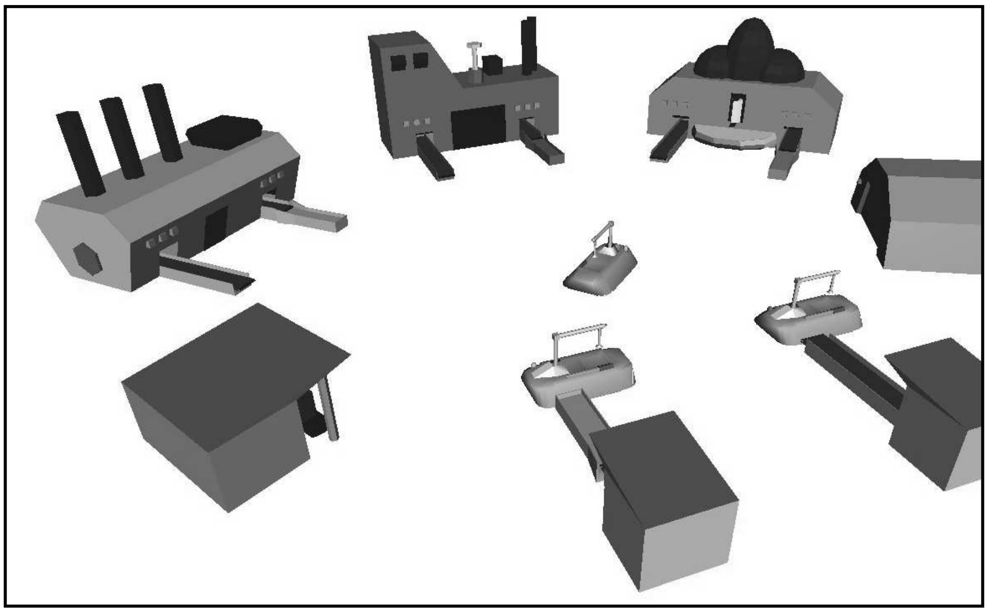


Figure 1.1: 3D Model of the Manufacturing Scenario

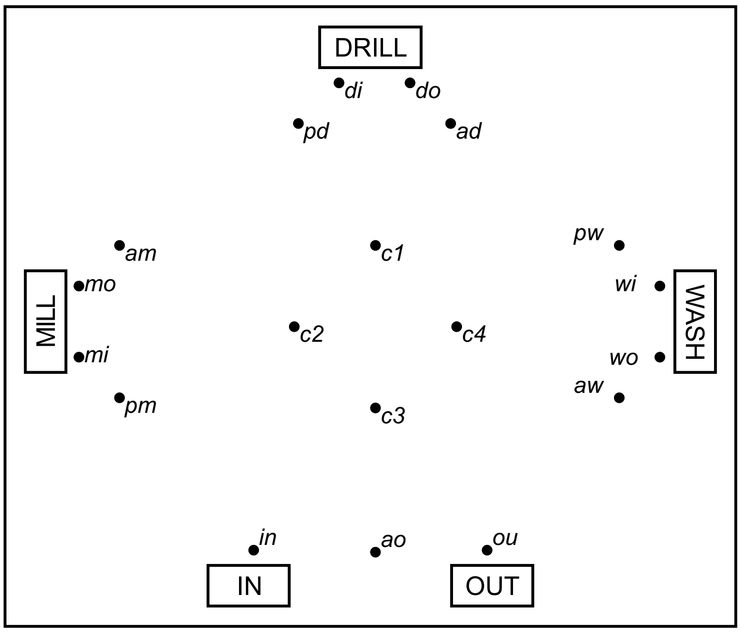
The transport system consists of a set of automated guided vehicles (AGVs), i.e., autonomous vehicles that carry items between stations. Each AGV can take only one item at a time. For input and output of items in the system we assume an input station and an output station. The items to process in the considered manufacturing systems are of two different kinds:

- Items of type *engine* have to be carried from the input station to the station responsible for milling, then to drilling, to washing, and finally to the output station.
- Items of type *shaft* have to be carried from the input station to the station responsible for drilling, then to washing, and finally to the output station.

Negotiation for transporting items is carried out by a station output buffer offering an item to transport and AGVs proposing to actually perform the transport.

- An AGV a_i is idle until it receives a request for delivery from a station s_k. Then, it
  1. sends a bid in form of the distance from its current position to s_k,
  2. moves to s_k on notification of acceptance from s_k,
  3. takes the item from the output buffer of s_k and moves to the next destination station,
  4. moves to a parking position and returns to Step 1.

  For position management, we assume that the vehicles are equipped with sensors to measure the distance from obstacles and define some essential intermediate positions the vehicles have to pass on their ways between the stations (see Figure 1.2). As the legend in that figure shows, the positions are named by identifiers, e.g., mi stands for the position at the *input buffer* of station *mill*.



Position Names

in	:	input storage
ou	:	output storage
ao	:	after output storage
pm	:	pre mill
mi	:	mill input buffer
mo	:	mill output buffer
am	:	after mill
pd	:	pre-position drill
di	:	drill input buffer
do	:	drill output buffer
ad	:	after drill
pw	:	pre-position wash
wi	:	wash input buffer
wo	:	wash output buffer
aw	:	after wash
c1	:	center position 1
c2	:	center position 2
c3	:	center position 3
c4	:	center position 4

Figure 1.2: Managed Intermediate Positions for AGVs

- Once having located a completed workpiece at its output buffer, a station s

  1. sends a request for delivery to the next destination station (more specifically, its input buffer) $inBuffer_{dest}$,
  2. is waiting for a notification from $inBuffer_{dest}$ for a specific time period,
  3. returns to Step 1 if $inBuffer_{dest}$ does not reply or answers with a reject to the request,
  4. broadcasts a request for delivery to all AGVs,
  5. is collecting bids with distances d_i from idle AGVs a_i for a specific time period,
  6. returns to Step 4 if no AGV replies,
  7. selects one AGV a_i from all received distances d_i, notifies AGV a_i for its acceptance and notifies the other AGVs for their rejection.

In addition to this scenario description of the manufacturing system, different requirements concerning the time-bounded execution have to be met. Time-bounds are essential to be able to determine the performance and throughput of the system under consideration. For example, it could be required that each transport by means of an AGV has to be completed within 300 time units. Additionally, in order to guarantee production progress, it might be required that each AGV is idle for at most 400 time units. Note that similar time-bounded requirements apply for the stations as well. UML modelers will be able to express such kind of requirements with the temporal OCL extension that is developed in this thesis.

1.3 Outline

In Figure 1.3, an overview of the main thesis chapters is given. An edge between two chapters indicates a dependency, i.e., the chapter at the origin of an edge provides concepts for the chapter to which the edge leads to. In chapters that have more than one incoming edge, concepts from different sources are (formally) integrated.

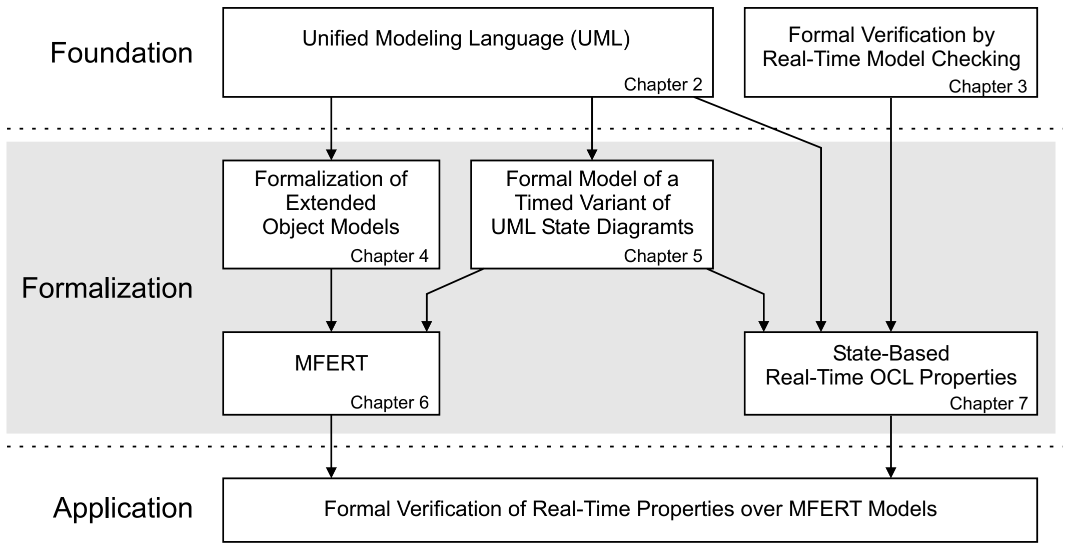

Foundation
Unified Modeling Language (UML)
Chapter 2
Formal Verification by Real-Time Model Checking
Chapter 3
Formalization
Formalization of Extended Object Models
Chapter 4
Formal Model of a Timed Variant of UML State Diagramts
Chapter 5
MFERT
Chapter 6
State-Based Real-Time OCL Properties
Chapter 7
Application
Formal Verification of Real-Time Properties over MFERT Models

Figure 1.3: Chapter Overview

In Chapter 2, an overview of UML is given, putting an emphasis on those parts of UML that are of particular relevance for this thesis, i.e., UML Class Diagrams, State Diagrams, and OCL.
Chapter 3 provides an introduction to formal verification, especially for the method of model checking under timing aspects (so-called real-time model checking).
Chapter 4 presents an extension of object models, which are a formalization of Class Diagrams introduced by Richters in [Ric01]. In that chapter, UML State Diagrams are formally integrated into object models, and a general notion for sequences of system states is defined.
Chapter 5 introduces a variant of UML State Diagrams. As the UML standard does not make concrete assumptions about times for the execution of actions, activities, and transitions, we define a specific semantics for a significant sublanguage of UML State Diagrams.
In Chapter 6, the formal models developed in the previous two chapters are used to define a formal model for MFERT. In particular, the structure of MFERT graphs is defined by a UML Profile, and behavioral descriptions are based upon the timed State Diagram variant of Chapter 4.
In Chapter 7, we extend the Object Constraint Language, such that it is possible to specify state-oriented temporal properties over UML State Diagrams. A formal semantics of this extension is given by a mapping to temporal logics formulae. Moreover, timing and data analysis queries are introduced.
Chapter 8 shows how the OCL extension can be applied to MFERT models by a case study with a manufacturing system scenario.
Chapter 9 concludes this thesis and gives an outlook on future research issues.

Chapter 2

Unified Modeling Language

”What do you think will be the biggest problem in computing in the 90’s?”
”There are only 17,000 three-letter acronyms.”
– Paul Boutin, 1989

The Unified Modeling Language (UML) is a graphical language for specifying object systems, in particular, object-oriented software systems. UML is a standard modeling language endorsed by the Object Management Group [OMG] in 1997. The current version at time of writing this thesis is UML 1.5, which was adopted in March 2003 [OMG03d]. In this chapter, we focus on the *language definition* of parts of UML, while the actual application of UML diagrams to build models is only treated by example based on the manufacturing case-study presented in the previous section. For more detailed information about modeling with UML, interested readers are referred to one of the numerous available textbooks, e.g., [BRJ99, JRB99, FS99, HK03]. Note that UML is intended to be a *modeling language* and leaves it to the responsibility of the modeler what diagram suits best to represent a specific part of the software system under development.

In this chapter, we first give some information on the approach taken by the OMG to define the UML language. In Section 2.2, a brief survey of the different UML 1.5 diagrams is given. For those parts of UML that are of particular relevance for this thesis, a more detailed description is provided in Section 2.3. As one goal of this thesis is to specify real-time properties using UML concepts, we also consider how real-time systems can be modeled with UML in Section 2.4. That section also provides an overview of approaches that extend UML for more suitable modeling of real-time systems. They are separated into those that deal with modeling architectures of real-time systems and those that consider time-related behavior modeling.

2.1 UML Language Definition

The OMG positions UML within a 4-layer architecture (see Table 2.1), in which elements of one layer are defined by means of the constructs introduced in the superior layer. On the highest layer M3, the Meta Object Facility (MOF) specification defines a common framework for representing metadata [OMG02]. It is used to model the three kinds of building blocks for *meta models* on layer M2, which are

Table 2.1: UML Architecture

Modeling Layer		Application
M3	Meta Meta Model	Meta Object Facility (MOF) [OMG02]
M2	Meta Model	UML Specification [OMG03d] as well as other standards, e.g., the Common Warehouse Metadata Specification [OMG01]
M1	Models	User-defined UML models
M0	Objects	Instances of (parts of) user-defined UML models

- objects (described by MOF Classes),
- links that connect objects (described by MOF Associations), and
- data values.

The UML is one application of the MOF. To describe its semantics, the specification is divided into several packages, e.g., for core concepts (package `UML::Foundation::Core`) or state machines (package `UML::BehavioralElements::StateMachines`). Each package is described by four sections:

1. **Abstract Syntax.** The abstract syntax of a package is defined by means of MOF compliant Class Diagrams. They present the metaclasses that define the UML language constructs and their relationships with multiplicity and ordering requirements. Figure 2.1 gives a sample MOF Class Diagram that shows the backbone part of package `UML::-Foundation::Core`, taken out of [OMG03d, Section 2.5.2]. That diagram is extended by some metaclasses that are particularly relevant in the remainder.

   In the diagram, rectangle boxes represent general language concepts, e.g., the box named `Class` is a concept for descriptors on level M1 that represent a set of objects with similar structure, behavior, and relationships. Within these rectangle boxes, additional attributes may be inscribed to specify inherent characteristics of a concept, e.g., instances of `Class` can be passive or active (with an own thread of control). The basic metaclass is `ModelElement`, representing a named entity in a model. It is the base for all modeling metaclasses in the UML.

   Note that UML uses the term *property* to generally refer to values attached to a model element, e.g., attributes, associations, and tagged values, while *features* are properties encapsulated in classifiers, i.e., operations, methods and attributes. However, in this thesis we will use the term *property* in the sense of *a non-functional specification of required dynamic behavior*.

   In the remainder, we refer to the boxes of Figure 2.1 as *UML metaclasses*. For convenience, we may speak of, e.g., ‘a `Class`’ instead of ’an instance of metaclass `Class`’.

Boxes with names in italic font are abstract concepts that cannot be instantiated on level M1, e.g., *`Classifier`*. These abstract concepts are usually part of (transitive) inheritance relationships. Inheritance relationships are indicated by connecting lines with a triangle on the inheriting concept side. Thus, *`Classifier`* inherits all characteristics (i.e., attributes and relationships) from *`GeneralizableElement`*, and `Class` in turn inherits from *`Classifier`*.

Diamonds indicate aggregation relationships, i.e., whole-part relationships. E.g., a *`Feature`* is part of a *`Classifier`*. More precisely, as *`Feature`* and *`Classifier`* are abstract concepts, each `Attribute`, `Operation`, or `Method` is part of a `Class`, `Interface`, or `DataType`. As such features must not exist without the classifier they belong to, the aggregation relationship is in this case marked with a *filled diamond*; this is also referred to as a composition relationship.

We will see in the next paragraph that such relationships have to be restricted by additional rules. E.g., it does not make sense to allow attributes in `DataTypes`, as elements of data types are immutable values like integer values or strings.

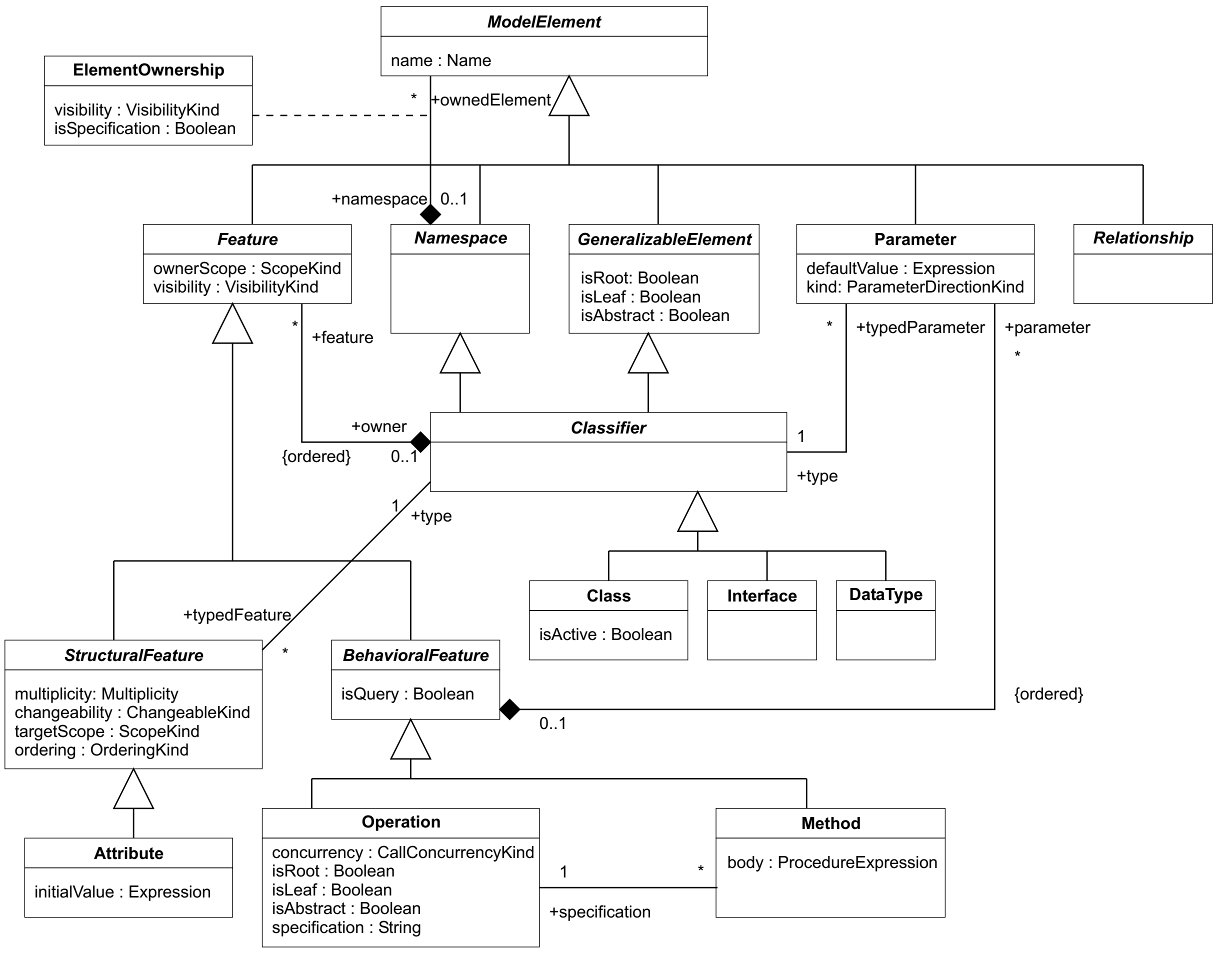


Figure 2.1: Modified Version of the UML Core Package [OMG03d, Section 2.5, Figure 2-5]

2. **Well-Formedness Rules.** Well-formedness rules are necessary to restrict the static structure of the concepts defined by MOF compliant Class Diagrams. These rules form the *static semantics* of UML, i.e., they define how an instance of a construct may be connected to other instances to be meaningful.

   These restrictions are usually expressed by precise invariants formulated in OCL in the context of UML metaclasses (for more details about OCL, see Section 2.3.3) and by further informal explanations.

   To give an example, we consider metaclass `DataType`. While a `Class` may have attributes, operations, and methods, a `DataType` may only have operations that do not change any data (i.e., query operations). As an OCL invariant, this is expressed by

```
context DataType inv:
  self.allFeatures()->forAll(f:Feature |
    f.oclIsKindOf(Operation) and f.oclAsType(Operation).isQuery())
```

   This OCL expression specifies that for each (user-defined) `DataType` on level M1, all (transitively inherited) features must be of a kind of query operation, i.e., operations without side effects. In such well-formedness rules in the context of UML metaclasses, some additional operations appear that are not defined directly by the metamodel or by the standard OCL library. E.g., operation `allFeatures()` subsumes all features of a data type and the inherited features of its supertypes. In Appendix B, we list some important additional operations defined for metaclass `Classifier` that are relevant for the remainder.

3. **Semantics.** The actual meaning of the language constructs is defined using natural language. In the official UML 1.5 specification, this section is about *dynamic semantics* of UML concepts. In terms of UML, *dynamic semantics define the meaning of a well-formed construct* [OMG03d, Section 2.3.1].

   Only for concrete metaclasses a semantic description is provided, as only these metaclasses have an actual meaning in the language.

4. **Standard Elements.** Additional predefined elements of metaclasses are listed with an informal textual description – in particular, so-called *stereotypes*(see Section 2.3.4). Basically, a stereotype represents a sub-category of a metamodel element with the same attributes and relationships, but for a distinct usage. Sample predefined stereotypes for `Class` are ≪`metaclass`≫, ≪`powertype`≫ (i.e., a user-defined classifier whose instances are classifiers which are children of a given parent on level M1), ≪`datatype`≫, or ≪`interface`≫.

2.2 Survey of UML Diagrams

UML defines twelve different types of diagrams [OMG03d, Section 3] that can be divided into three categories. Four diagram types are for modeling the static system structure; five are for modeling different aspects of dynamic behavior; and three others are for system organization and management. Table 2.2 lists the corresponding diagram names for each category.

Table 2.2: Different UML Diagrams

Type	**UML Diagram**
Structural Diagrams	Class Diagram, Object Diagram, Component Diagram, Deployment Diagram
Behavioral Diagrams	Use Case Diagram, Sequence Diagram, Activity Diagram, Collaboration Diagram, State Diagram
Model Management Diagrams	Package Diagram, Subsystem Diagram, Model Diagram

The following gives a brief overview of the twelve different diagram types available in UML. A more detailed classification based on views supported by each of the diagram types can be found in [RJB98].

- **Use Case Diagrams.** The functionality of a system w.r.t. the users is modeled as a set of use cases. A use case represents an interaction of a user (or: actor) and the system under consideration. Basically, all what is modeled by Use Case Diagram is to list actors and the use cases they participate in.

- **Class Diagrams.** Classes are a core concept of object-oriented software development. Classes, their features (attributes, operations and methods), and their relationships are modeled with Class Diagrams. Class Diagrams are further discussed in Section 2.3.1.

- **Object Diagrams.** An Object diagram shows a possible situation a system may be in at a particular point of execution. Object Diagrams appear to be similar to Class Diagrams, as objects are connected by links in the same manner as classes are connected by associations. In Object Diagrams, object attributes may have specific values. It is important to note that the actual diagram is a model on level M1, while it represents a situation that occurs on level M0.

- **Sequence Diagrams.** Possible messages sent between objects in a system can be modeled with Sequence Diagrams. Objects are placed horizontally, each of which is provided with a vertical *life line*. Horizontal directed arcs between these life lines represent messages sent from one object to another. Each arc is annotated with a label to indicate the kind of message. The vertical order of arcs denotes the order of messages in time.

- **Collaboration Diagrams.** While Sequence Diagrams are ordered according to elapsing time, Collaboration Diagrams emphasis on structural aspects. Basically, a Collaboration Diagram is an extension of an Object Diagram, but in addition to links between objects, Collaboration Diagrams show messages the objects send each other. Like in Sequence

Diagrams, messages among objects are represented by directed arcs that point to the receiving object. The order of messages is determined by explicitly numbering each directed arc. Note that Collaboration Diagrams can be translated into Sequence Diagrams and vice versa.

- **State Diagrams.** Reactive behavior of an object is modeled by State Diagrams. They are an extension of the classical concept of Harel Statecharts [Har87], but with different semantics. A detailed description of UML State Diagrams is given in Section 2.3.2.

- **Activity Diagrams.** Activity Diagrams focus on flows driven by internal processing vs. external events. A column format (referred to as *swimlanes*) may be used to explicitly show activities according to their belonging objects. Object names are put on top of each column, and vertical bars separate the columns. Activity Diagrams are a variant of State Diagrams, in which a state represents an ongoing activity (e.g., an operation) and a transition automatically fires when the source state completes its activity.

- **Component Diagrams.** A Component Diagram describes the organization of the physical building blocks (or: software units, components) in a system. They are shown as rectangle boxes with tabs in the Diagram. Dependencies among components are shown by dashed arrows between the rectangle boxes.

- **Deployment Diagrams.** In Deployment Diagrams, nodes represent physical resources that execute component instances. A Deployment Diagram shows which component instances are placed on which processing nodes.

- **Package Diagrams.** A Package Diagram shows how classes and other packages are grouped into packages.

- **Subsystem Diagrams.** A Subsystem Diagram shows how logically related sets of components form subsystems. While a Package Diagram is used to organize model elements into groups, a subsystem represents a behavioral unit of the model. A subsystem may have interfaces and operations and is divided into specification and realization elements.

- **Model Diagrams.** A Model Diagram includes all of the other diagrams to show how the complete system is structured and functions. Note that different Model Diagrams can be defined for the same physical system. Each Model Diagram represents a view of the physical system, depending on its purpose and level of abstraction. E.g., in a system development process, we may have an analysis model, a design model, and an implementation model for the same physical system.

In this thesis, we focus on Class Diagrams for modeling the static system structure and on State Diagrams for modeling dynamic behavior of objects. Additionally, restrictions on UML models are expressed in OCL. These parts of UML are more detailly described in the following section.

2.3 Details of Selected Parts of UML

2.3.1 UML Class Diagrams

UML Class Diagrams are used to describe the static structure of a system. As an example, Figure 2.2 shows main parts of the classes used in the manufacturing case study.

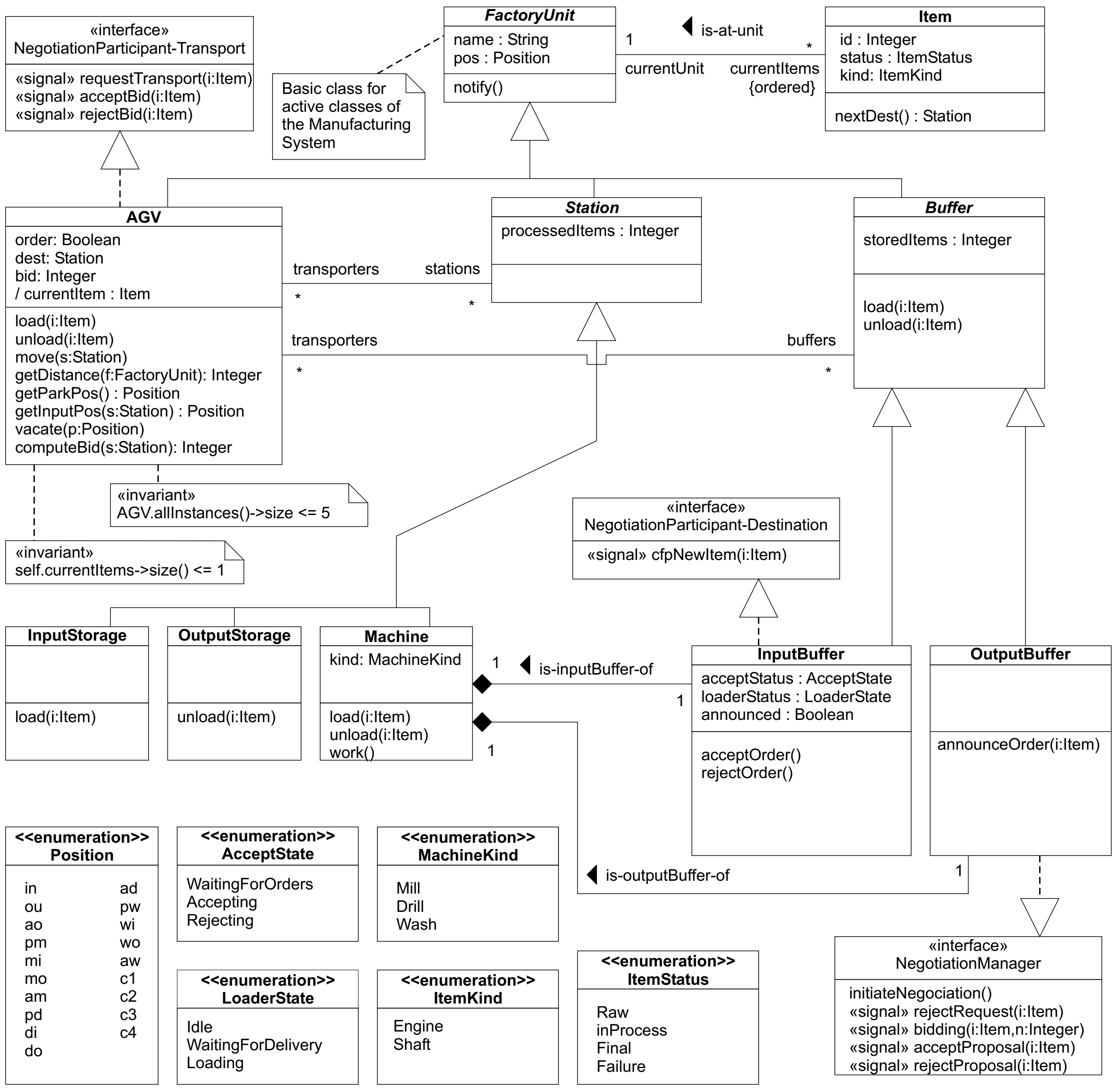


Figure 2.2: Case Study Class Diagram (M1 Level)

2.3.1.1 Classes

Classes are given by rectangular boxes separated into horizontal sections. In the uppermost compulsory section, a class name and optionally a stereotype and class properties are given;

the other sections are optional and are used to specify attributes and operations. As abstract classes are frequently used, for abbreviation purposes italic font is applied instead of explicitly writing `classname` {`abstract=true`} (e.g., *`FactoryUnit`* in Figure 2.2). Several additional annotations for attributes can be applied that concern – among others – scope (whole class or instances), visibility (public, package, protected, private), multiplicities, and initial values. Similarly, operations can be annotated with scope and visibility, and additionally with typed parameters (with kinds in, out, and inout) , a return type, and other properties, e.g., {`query`} for operations without side effects.

2.3.1.2 Associations

Edges between rectangles represent static relationships among class instances and are called associations. Most frequently, *binary* associations, i.e., associations connecting two classes, are applied in practice, but there are certain kinds of associations that need to be investigated in more detail, i.e., aggregation and composition.

In order to improve expressiveness, associations can be annotated by different means:

- An association can be named, e.g., association `is-at-unit` between `Item` and *`Facto-ryUnit`* in Figure 2.2. A filled arrowhead shows the reading direction in this context.
- Association ends can optionally be named with *rolenames*. Such an annotation represents the role a class plays in the association, e.g., `currentUnit` in association `is-at-unit`. If no role name is given, the class name with a lower case first letter is used to navigate along an association in the Class Diagram.
- Multiplicity restrictions attached to an association end are used to specify the number of objects a given object on the opposite association end(s) can be associated with, e.g., in association `is-at-unit`, each `Item` object is associated with exactly one factory unit object – nevertheless, that factory unit may change during run-time, as the association end is not marked with {*frozen*}. A star '`*`' indicates an arbitrary number of objects, i.e., `0..*` or simply `*` means that an object may be associated with any number of objects on the opposite association end.
- Associated objects can be ordered as it is specified for items at factory units which are ordered by the time of arrival in the corresponding factory unit.
- Arrowheads at association ends are used to explicitly specify the direction in which it is possible to navigate from one object via an association to associated object(s). Nevertheless, in practice undirected edges (i.e., no arrowheads at the association ends) are interpreted as navigable in all directions, though this is not the standard!

Aggregation and Composition. Aggregation is often classified as *part-whole-relationship* and is represented by a hollow diamond at the association end which plays the role of the *whole*. Note that aggregation does not add any semantics to the association, because the participating objects are still independent of each other, i.e., the existence of an object is not restricted by establishing an aggregation relationship. This is different in *composition*s, a stricter variant of *whole-part-relationship* that is denoted by a filled diamond at one association end. In compositions, *parts* may belong to at most one *whole*, e.g., in association `is-buffer-of`, a buffer in the case study belongs to (or: is part of) exactly one station, and is only existing in the system as long as the corresponding station is in the system.

Though the concept of aggregation and composition seems to be quite clear, a couple of semantic issues arise, as one can distinguish between *dependency* and *exclusiveness* in part-of-relationships. A dependent component object in this context refers to its existence in accordance to its belonging whole. An exclusive component object refers to whether the part belongs to one whole or more than one whole. It turns out that *dependent, not exclusive* component objects cannot directly be modeled with UML concepts [HK03].

2.3.1.3 Generalization

Generalization is a relationship between a more specialized subclass and a more general parent class (or: base class). Subclasses inherit characteristics like attributes, operations, and associations from parent classes. Generalizations are indicated by edges with a hollow triangle attached to the parent class, e.g., `AGV` is a subclass of *`FactoryUnit`* in Figure 2.2. Semantically, the notation with one superclass and several subclasses (as for *`FactoryUnit`*) is equivalent to a corresponding number of simple binary generalizations, as, e.g., for superclass *`Buffer`* and its subclasses `InputBuffer` and `OutputBuffer`. In UML, the concept of generalization is not clearly separated from the classical concepts known for generalization. Classically, three different conceptual kinds of generalization are distinguished [HK03, pages 48-51]:

1. **Specification Inheritance.** This concept basically complies to the substitutability principle. One main aspect is contra variance: re-defined operations in subclasses may only have a weakened pre-condition (or: larger definition set) and restricted post-condition (or: restricted value set).

2. **Specialization Inheritance.** This is also called *is-a*-relationship and can be best distinguished from specification inheritance by an example: Consider the two types `Real` and `Integer`. Each integer *is-a* real value, so we have a specialization inheritance. Nevertheless, re-definition of common `Real`-operations usually require different inputs in subclass `Integer`, e.g., the add-operation. Thus, the concept of contra variance mentioned above for specification inheritance is violated, as the operation pre-condition is strengthened.

3. **Implementation Inheritance.** This kind of inheritance is mainly applied to re-use existing code and is usually established in late phases of development. A semantic relation does not need to exist. Nowadays, it is refrained from implementation inheritance in favor of aggregation. In UML, explicit annotation of generalization by {`is-a`} denotes implementation inheritance.

Generalization in UML is heavily identified with the concept of substitutability, i.e., an instance of a class may be used whenever an instance of a superclass is expected. Substitutability, however, is a particular characteristic of only one of the classical generalization semantics.

2.3.1.4 Other Concepts

User-defined (finite) data types can be modeled by *enumerations*. The elements of enumerations are called *literals*, e.g., `Mill`,`Drill`, `Wash` for enumeration `MachineKind` in Figure 2.2. For explicitly naming literals, double colon notation is used, e.g., `MachineKind::Mill`. This notation is in accordance with (hierarchical) states in UML State Diagrams (see Section 2.3.2).

Comments and *constraints*, in particular OCL invariants (see Section 2.3.3), are associated by a dotted line with the corresponding class (see Figure 2.2).

In UML, an *interface* is is denoted by a rectangular box labelled with ≪`interface`≫ in front of its name compartment. With interfaces, a particular set of operations is modeled. Classes can realize interfaces, i.e., they provide the operations defined for an interface, when staying in a realization relationship, indicated by a dotted line and a triangle at the interface end. E.g., in Figure 2.2, output buffers realize the interface `NegotiationManager` for performing negotiations among execution of transports, i.e., an output buffer object can In contrast, a UML *type* in the sense of an abstract data type defines a value set for objects together with appropriate operations, but in this case it is not intented to actually implement objects of the type.

Often, classes are needed that are mainly responsible to keep data (i.e., container classes) of a certain type. In static modeling languages (like UML), this means that for each container class that has a different data type to manage, a different class needs to be modeled. To overcome this problem, parameters can be attached to classes. Notationally, this is indicated by a dashed rectangle positioned over the upper right corner of the *parameterized class*. Inside the dashed rectangle, the parameters are listed. For an example of a parameterized class, consider the OCL standard library types description in Figure 2.11 on page 47.

There are several more concepts available in UML Class Diagrams, which are not of relevance for the rest of this thesis and therefore not covered in detail, e.g., qualified associations, dependencies, and roles. The reader is referred to the official specification [OMG03d] or the several textbooks on UML, e.g. [BRJ99, FS99, HK03].

2.3.2 UML State Diagrams

For modeling (reactive) behavior of objects and operations in the context of UML, State Diagrams are applied. The UML State Diagram notation bases upon Harel's Statecharts [Har87]. Generally, State Diagrams are graphs in which nodes represent (composite) states and directed arcs represent transitions between states. Transitions are usually triggered on (external) stimuli, i.e., perceived and dispatched events. In this section, we give an informal introduction on the main concepts of UML 1.5 State Diagrams, so that it is possible to understand the formalizations in Chapters 4 and 5.

Besides such *behavioral state machines*, the UML 2.0 Superstructure Proposal introduces *protocol state machines* that are employed to model usage protocols [OMG03f, Section 15]. The corresponding graphical notation is especially tailored to define the lifecycle of objects and the required order of operation invocations. In this context, *interfaces* and *ports* are new language concepts introduced in UML 2.0. However, we focus on behavioral state machines in this thesis and therefore mainly refer to the UML 1.5 specification.

The official UML 1.5 specification informally defines State Diagram states and transitions as follows [OMG03d, Section 2.12.2.10].

> *A state [...] models a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some activity; that is, the model element under consideration enters the state when the activity commences and leaves it as soon as the activity is completed.*

Conceptually, an object remains in a state for an interval of time and then changes to another state without consuming time. It is commonly assumed that transitions take no time, conforming to Harel Statecharts [Har87]. But generally, UML State Diagram semantics allow modeling of non-instantaneous transitions as well as instantaneous, flow-through states (see [OMG03d, Section 3.75.1]). However, there is currently no standard notation provided to explicitly specify transition times.

Actions, Events, and Activities. At this point, it must first be clarified what UML considers as an *action* and as an *activity*. In UML, an action is a specification of an executable statement. Basically, an action can refer to sending a synchronous or asynchronous message (call action or send action) to an object, creating or destructing an object, modifying a link or an attribute value, or returning from a previously called operation. Execution of a call or send action generates an *event* that is perceived by the addressed receiving object. Generally, the kinds of known events are

- signal events (generated by asynchronous send actions),
- call events (generated by synchronous call actions),
- change events of form `when (booleanExpr)`, used to continuously observe the boolean condition to become true,
- time events, specified by `after (timeSpec)` and used to denote a time at which the transition has to be fired after entering the current state,
- (implicit) completion events raised after finishing an actions and activity within a state.

In contrast to actions that are conceptually seen to be atomic without consuming time, an *activity* is interpreted as ongoing and time-consuming, though there are only rudimentary specification means to explicitly model the time or timing interval, in which an activity is completed. Activities are specified either by computational expressions within a particular section of a state specification ('do/-activity') or by explicitly providing another State Diagram to describe the corresponding behavior (e.g., as a nested state or as a stand-alone State Diagram associated with the operation representing the activity).

States and Transitions. States are represented as boxes with a name and an optional compartment for specification of actions and activities. An entry/-action is executed when the state is entered (or: activated), then – during activation of the state – an activity may be performed until the state is exited (or: deactivated). Just before exiting, an exit/-action may be performed. There are several more advanced concepts for states, e.g., internal transitions, deferred events, as well as so-called *pseudostates*, in particular, initial states (denoted as black circles), shallow and deep history states (denoted by an H or H* in a circle), final states (denoted by a 'bull's eye'), static and dynamic choice states, and synchronization states. For more details on these, the reader is referred to the previously recommended UML textbooks.

Transitions have a source and a destination state and can be annotated by an event specification `evt`, a guard condition `[grd]`, and actions to execute `/act`. Perceived events matching the event specification `evt` enable the transition to be taken, provided that the transition source state is activated and that the transition guard `[grd]` evaluates to true. If there is no transition event specified, the transition is fired after the activity of the source state has finished.

Composite States. State Diagrams can be hierarchically refined using OR-states and AND-states. An OR-state is a composite state, in which at each time exactly one of its substates is activated, when the OR-state is activated. In this context, the previously mentioned history states can be applied. Entering an OR-state via a transition to a (shallow) history state re-activates the substate that was the latest active one before the OR-state was previously left. By deep history states, re-activation takes place on all subsequent hierarchy levels.

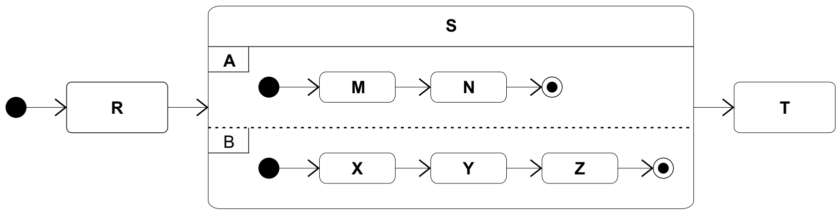
With AND-states, concurrent activated substates are modeled: these *orthogonal regions* are graphically separated by dashed horizontal lines. This notion of refined states leads to a precedence rule for transition selection that is different from former Statechart semantics; in UML, transitions on lower levels take precedence over transitions of upper levels. More precisely, for each pair of transitions with the same specified triggering event and valid guards, the rule requests that if the transitions are ancestrally related (i.e., they do not only affect orthogonal regions), the transition with the source state on the lower level is selected to be taken. Nevertheless, transitions that are not ancestrally related can be taken in parallel.

It is allowed to draw transitions among several state levels by crossing state boundaries (i.e., interlevel transitions). For complex transitions that enter or leave AND-states, so-called fork and join transitions can be applied, especially when other than the default initial states should be activated in the entered orthogonal regions. However, in practice this notation is more common in activity diagrams.

Dynamic Semantics. Execution of UML State Diagrams is controlled by (a) an *event dispatcher* that subsequently selects events from an implicit *event queue* and (b) an *event processor* that performs *run-to-completion steps* (RTC-steps). In an RTC-step, first the set of fireable transitions is determined based on the previously dispatched event and additional transition conditions, i.e., the set of transitions that can potentially be taken. UML does not make assumptions about the general functionality of the event dispatching mechanism, e.g., a first-in first-out queue or based upon priority schemes for events. Nevertheless, some basic processing rules are given, e.g., internally generated completion events take priority over external events. It can happen that the set of fireable transitions for a dispatched event is empty; in this case the event is ‘consumed’ without any effect on the State Diagram. If the event is specified as deferrable in the currently activated state(s), the event is re-entered into the event queue, or otherwise just dropped. It can also happen that fireable transitions compete with each other, as it is not allowed to fire them in parallel. In such cases, UML provides some rules (e.g., inner transitions have higher priority) to help to determine the maximum consistent set of transitions from the set of fireable transitions. Though, this does not prevent from situations in which non-deterministic choices still have to be made. Having a set of consistent (or: enabled) transitions, these transitions are then fired, subsequently executing the exit actions of the state(s) to leave, the actions specified for the chosen transition(s), and the entry-actions and activities specified for the entered state(s).

The official UML specification currently provides only an informal dynamic semantics of UML State Diagrams by natural language in [OMG03d, Section 2.12.4], leaving open a couple of semantic variation points, e.g., for event dispatching or storing deferred events. Different approaches have captured the dynamic semantics of UML State Diagrams in a formal way, e.g., by an abstract interpreter [LP99] or ASMs [BCR00].

Active State Configuration. When a State Diagram is modeled with composite states, more than one state can be activated at a certain point of time, and it can get confusing to speak about *the current state* of a State Diagram. For dealing with such situations, the notion of an *active state configuration* is used in UML [OMG03d, Section 2.12.4.3]. Unfortunately, that informal definition is not concise, as final states are not considered to be part of the active state configuration, although a UML State Diagram may reside in a final state for a notable period of time, as illustrated in Figure 2.3. In that figure, possible active state configurations are listed using the keyword `FinalState` for the final states.



```
List of States:

R
S
S::A
S::B
S::A::M
S::A::N
S::A::FinalState
S::B::X
S::B::Y
S::B::Z
S::B::FinalState
T
```

```
List of Active State Configurations:

{R}
{S, S::A, S::B, S::A::M, S::B::X}
{S, S::A, S::B, S::A::M, S::B::Y}
{S, S::A, S::B, S::A::M, S::B::Z}
{S, S::A, S::B, S::A::M, S::B::FinalState}
{S, S::A, S::B, S::A::N, S::B::X}
{S, S::A, S::B, S::A::N, S::B::Y}
{S, S::A, S::B, S::A::N, S::B::Z}
{S, S::A, S::B, S::A::N, S::B::FinalState}
{S, S::A, S::B, S::A::FinalState, S::B::X}
{S, S::A, S::B, S::A::FinalState, S::B::Y}
{S, S::A, S::B, S::A::FinalState, S::B::Z}
{T}
```

Figure 2.3: Active State Configurations

The State Diagram can be in the situation that OR-state `S::A` has reached its final state, but `S::B` is still in state `Z`. Thus `S::A` resides in state `S::A::FinalState` until `S::B` also reaches its final state. At that point of time, a completion event is implicitly generated and in the next RTC-step, the transition to state T is fired. As this (conceptually) happens without consuming time, there is no active state configuration in which both substate reside in their final states.

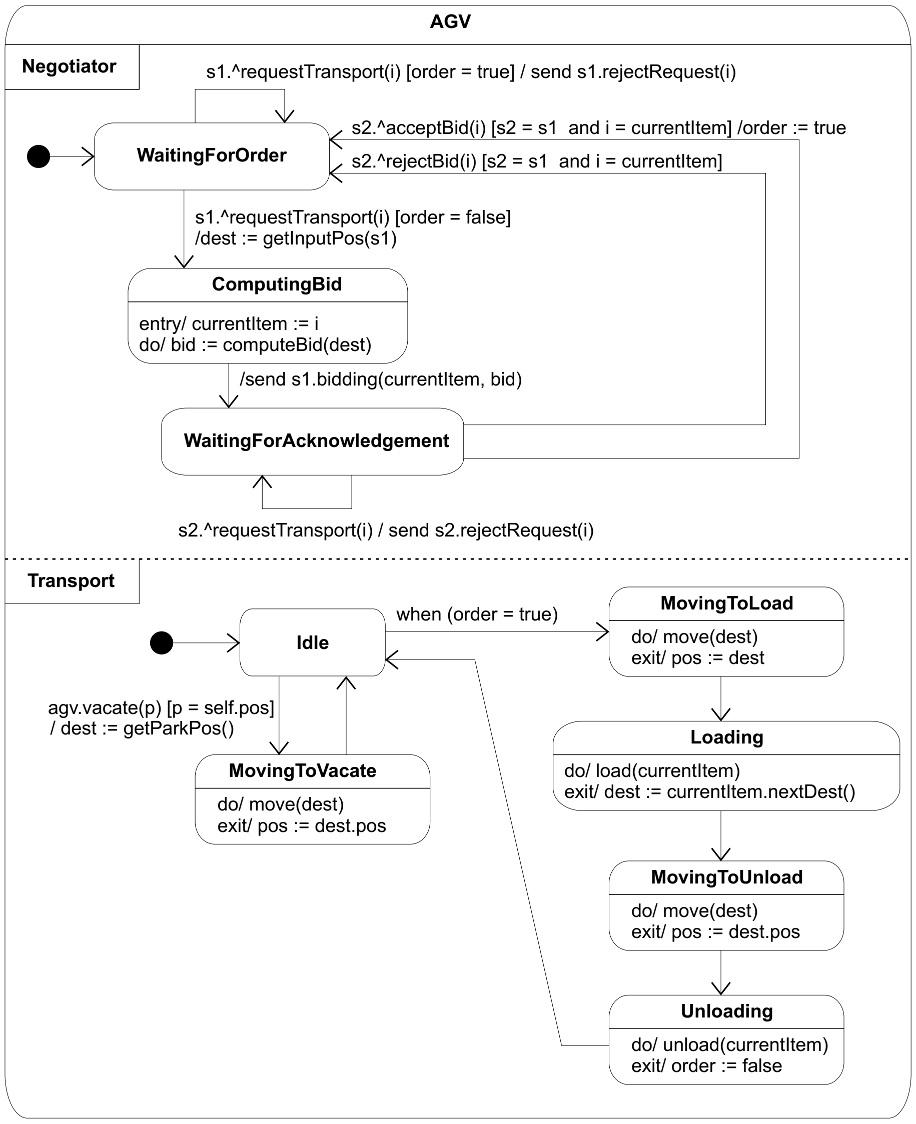

AGV
Negotiator
s1.^requestTransport(i) [order = true] / send s1.rejectRequest(i)
WaitingForOrder
s2.^acceptBid(i) [s2 = s1 and i = currentItem] /order := true
s2.^rejectBid(i) [s2 = s1 and i = currentItem]
s1.^requestTransport(i) [order = false]
/dest := getInputPos(s1)
ComputingBid
entry/ currentItem := i
do/ bid := computeBid(dest)
/send s1.bidding(currentItem, bid)
WaitingForAcknowledgement
s2.^requestTransport(i) / send s2.rejectRequest(i)
Transport
Idle
when (order = true)
MovingToLoad
do/ move(dest)
exit/ pos := dest
agv.vacate(p) [p = self.pos]
/ dest := getParkPos()
MovingToVacate
do/ move(dest)
exit/ pos := dest.pos
Loading
do/ load(currentItem)
exit/ dest := currentItem.nextDest()
MovingToUnload
do/ move(dest)
exit/ pos := dest.pos
Unloading
do/ unload(currentItem)
exit/ order := false

Figure 2.4: State Diagram: Automated Guided Vehicle

Example 1: AGV State Diagram. To give a more concrete example, Figure 2.4 shows the internal behavior of an AGV by two orthogonal regions. The upper one is for negotiating orders w.r.t. transporting items. By default, the AGV waits for incoming requests to perform a transport in state `WaitingForOrder`. After a request from a station to execute a transport `requestTransport(i)`, the AGV computes the estimated costs to get to the requesting station `s1` in state `ComputeBid` and sends a message with a bid (i.e., the distance to move) to `s1`. After bidding, the AGV is waiting for an acknowledgement or rejection of the bid in state `WaitingForAcknowledgement`. We assume that an AGV can only take part in one negotiation at a time, thus all other requests have to be rejected while being in state `WaitingForAcknowledgement`. Similarly, if the AGV is in state `WaitingForOrder`, but a transport is still currently performed, i.e., `order = true`, an AGV will reject any request for transporting items.

The lower orthogonal region called `Transport` is for actually executing a transport by activities like moving to the station for loading, loading an item, moving to the destination station, and unloading the item. In addition, when the AGV is standing at a position that needs to be used by a different AGV, it can be required to vacate that position. Note here that the transition from state `Idle` to state `MovingToLoad` is triggered by a change event `when (order=true)`. Conceptually, it is permanently checked for this event to become true while state `Idle` is activated, and the transition is taken as soon as the condition `order = true` becomes true. The states `MovingToLoad`, `Loading`, `MovingToUnload`, and `Unloading` are left as soon as the associated activities and actions are finished, as their outgoing transitions are not annotated by a triggering event.

2.3.3 Object Constraint Language

The Object Constraint Language (OCL) is a language to express restrictions in object-oriented models. Originally called *business modeling language*, OCL was developed at IBM as part of an object-oriented modeling method called *Syntropy* [CD94]. In UML version 1.3, OCL became part of the official UML specification. OCL expressions are either directly applied by textual annotations within different UML diagram types or by separately listing them with an additional explicit specification of their context. One remarkable application of OCL is its use to formulate well-formedness rules for UML diagrams on the UML metamodel level (M2). Precise OCL expressions have thus replaced informal and ambiguous semantical descriptions that were given in English language in earlier versions of UML.

The concept of constraints in the field of object-oriented languages is not new. In [Mey97], constraints are called *assertions*, which are used to formulate pre- and postconditions for operations as well as invariants. These concepts became an integral part of the Eiffel programming language already in the 1980s. Formulation of pre- and postconditions for operations is also referred to as the principle of *design by contract*, i.e., an agreement on correct execution of a service between a client and a supplier. Both parties have obligations and rights: Concerning the execution of the service (i.e., the operation) the client (i.e., the operation caller) is obliged to meet the precondition, while the supplier (i.e., the operation callee) has the right to assume that the precondition holds. The client can then expect that the supplier delivers the corresponding result promised in the postcondition, and it is the obligation of the supplier to give that promised result. James Rumbaugh, one of the co-founders of UML, classified constraints as a '*functional relationship between entities of an object model*' [RBP$^+$91] in the sense that a constraint restricts the potential values of model entities. Influenced by this work, Jos Warmer, the main developer of OCL, similarly defines a constraint as follows [WK03, Section 1.5.1].

> *A constraint is a restriction on one or more values of (part of) an object-oriented model or system.*

Naturally, it does not make sense to formulate constraints without a model to refer to. In the context of OCL constraints, we will call the corresponding UML model the *referred UML user model*. Each Classifier[1] defined within the referred UML user model represents a distinct OCL type and is implicitly included in the OCL Standard Library as a subtype of `OclAny`.

In the context of UML models, a constraint written in OCL can be one of the following.

- An *invariant* defined for a class, type, interface, stereotype, or state,
- a *pre-* or *postcondition* attached to an operation,
- a *guard condition* attached to a transition (in State and Activity Diagrams) or a message (in Sequence and Collaboration Diagrams),
- a *well-formedness rule* in the UML metamodel, given as an invariant over a metaclass.

Besides the official OCL language description as part of the official UML specification [OMG03d, Chapter 6], available literature on OCL is limited to two textbooks with several examples [WK99, WK03], a book on recent research efforts w.r.t. OCL semantics and applications [CW02], and a number of conference articles[2].

OCL has a simple non-symbolic syntax defined by a context-free grammar. It claims to be precise and unambiguous, but still easy understandable by designers in the area of object-oriented technology, as OCL expressions are syntactically held in the style of a programming-language notation [WK03]. OCL has a number of core concepts, e.g., it is declarative without side-effects and has a set of predefined types dedicated to deal with object collections. Though OCL is by now an official part of UML, it is currently only loosely coupled to the rest of UML, as OCL is still missing a metamodel definition. Currently, there is just a context-free grammar and an informal specification of the language concepts and its standard operations provided in UML 1.5 [OMG03d, Chapter 6]. To overcome this deficiency, a *UML 2.0 OCL Request for Proposals* has been issued by the OMG [OMG00a]. An extensive answer to that call developed by a team of leading OCL experts is can be found in [IHJ$^+$03]. More recently, it has been adopted by the OMG in October 2003 [OMG03b]. At the time of writing this thesis, the latter document is still being finalized.

In the next two sections, we study the concepts and semantics of OCL based upon the adopted version of the OCL 2.0 specification [OMG03b]. In Section 2.3.3.3, we will then discuss open issues of the OCL specification.

2.3.3.1 Concepts

In terms of OCL, we generally speak of constraints for *types* instead of using the UML terms classifier, classes, interfaces, and data types.

[1]More precisely, one has to speak of 'each instance of metatype `Classifier`', but we here adopt the terms used in the UML specifications.

[2]A list can be found online at http://www.db.informatik.uni-bremen.de/umlbib

Invariants. Invariants are restrictions on values of objects that have to hold 'all the time'. Invariants are defined in the context of classes, types, interfaces, and stereotypes. UML 1.5 also defines *state invariants*, i.e., a special form of invariant for active classes. An invariant must hold for every object of the specified contextual type. Within Class Diagrams, invariants are notated in a comment box stereotyped by ≪invariant≫. An invariant is associated with a context. E.g., in the context of type `AGV` in Figure 2.2, the following expression specifies the invariant that restricts the number of items carried to be less or equal to 1:

```
self.currentItems->size() <= 1
```

where `self` is an object of type `AGV`. The keyword `self` refers to the object from where we start to evaluate an OCL expression.

OCL invariants can also be specified in a separate document. In this case, the type of the contextual instance is explicitly specified in a context clause as illustrated in the following example.

```
context AGV inv: self.currentItems->size() <= 1
```

The context clause starts with the keyword `context` and is followed by the type. The label `inv` indicates that the constraint is an invariant. If the particular context is clear, the `self` keyword can be omitted. As an alternative for `self`, an arbitrary name can be declared and used instead:

```
context vehicle:AGV inv: vehicle.currentItems->size() <= 1
```

Optionally, a name can be given to the constraint after the keyword `inv`, e.g.,

```
context vehicle:AGV inv maxNumberOfTransportedItems:
vehicle.currentItems->size() <= 1
```

In the UML metamodel, that name is an attribute of the metaclass `Constraint`.

We now briefly explain how to interpret an OCL expression. The dot '.' is used to access object *features*, i.e., attributes, opposite association ends (identified by their role names), and operations and signals defined for the class of an object. In the examples above, the dot-notation is used to navigate within the Class Diagram and yield those objects associated to the AGV object `vehicle` on the left via the association name `currentItems` on the right. In this case, we get the set of all instances of class `Item` that are currently associated to a particular `vehicle` object. An arrow `->` indicates that the expression to its left represents a collection of objects. OCL distinguishes between three kinds of collections: sets, multisets (or bags), and sequences. The operation to the right of the arrow is applied to this collection. In our example, operation `size()` returns the number of elements of the previously determined set of items.

To uniquely interpret the dot-notation used in OCL, some additional syntactical restrictions apply to Class Diagrams. As the OCL dot-notation is used for both navigation by role names (e.g., `self.currentItems`) and accessing attributes (e.g., `self.order`), all navigable role names and attribute names of a classifier must be pairwise distinct. Similarly, operation and signal names (in combination with their parameters) of a classifier must be pairwise distinct. Though these restrictions do not explicitly appear in the abstract syntax definition of Class Diagrams, they are generally advisable to prevent misinterpretations. We therefore assume in the remainder that Class Diagrams comply to these additional restrictions. In Section 4.1.5, we will present corresponding formal definitions and restrictions.

Pre- and Postconditions. As already mentioned, specification of pre- and postconditions can be seen as a contract between an operation caller and the operation callee. Semantically, a precondition has to be true at the beginning of execution of an operation, and a postcondition has to be true right after the operation has ended. Similarly to invariants, pre- and postconditions can be directly applied within Class Diagrams in comment boxes attached by a dotted line to an operation. Figure 2.5 shows how pre- and postconditions can be attached to operations in Class Diagrams. In this case, a precondition is attached to operation `nextDest()` of class `Item`. In the considered operation, the next factory unit for an item has to be determined. The operation determines the next destination when a transport by an AGV is required, i.e., when an item currently is at the input storage or at one of the output buffers. The corresponding precondition is shown in the figure.

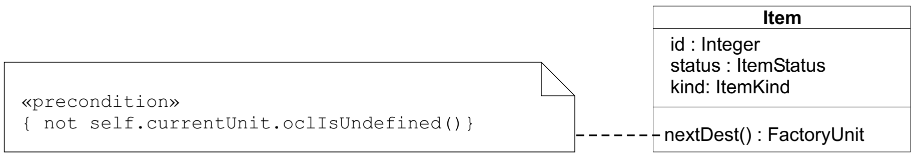


Figure 2.5: Pre- and Postcondition Notation

To show another form of representation, we now formulate the postcondition separately and therefore have to explicitly specify the contextual operation as follows.

```
context Item::nextDest() : Station
post: let f = self.currentUnit in
if self.kind = ItemKind::Engine then
  if f.oclIsTypeOf(InputStorage) then
    result = Machine.allInstances()->
               any(m:Machine | m.kind = MachineKind::Mill).inputBuffer
  else
    -- in this case it holds: i.currentUnit.oclIsTypeOf(OutputBuffer)
    if f.oclAsType(OutputBuffer).machine.kind = MachineKind::Mill then
      result = Machine.allInstances()->
                 any(m:Machine | m..kind = MachineKind::Drill).inputBuffer
    else if f.oclAsType(OutputBuffer).machine.kind = MachineKind::Drill then
      result = Machine.allInstances()->
                 any(m:Machine | m.kind = MachineKind::Wash).inputBuffer
    else
      -- in this case the machine kind must be Wash
      result = OutputStorage.allInstances()->any(true)
      endif
    endif
  endif
else     -- in this case the item kind is ItemKind::Shaft;
   [...] -- as this case is very similar, it is omitted in this example
endif
```

A postcondition is particularly used to specify the result of an operation. In the example postcondition, only the case for item kind `Engine` is shown, while the other case for item kind `Shaft` is omitted here, as it is very similar and does not show any new aspects. Recall that in our case study, items of kind `Engine` have to be subsequently transported to machines Mill, Drill, and Wash, and then finally to the output storage.

The `let`-statement defines a local variable `f` referencing to the factory unit the item currently belongs to. The type of `f` is either `InputStorage` or `OutputBuffer`, as we can expect that the precondition is met.

Assume now that `f` is an instance of `OutputBuffer`, i.e., `f.oclIsTypeOf(OutputBuf-fer)` evaluates to true. In order to be type consistent, we have to perform a type cast `f.oclAsType(OutputBuffer)` to be able to refer to the features of `OutputBuffer` objects. Then we can navigate to the machine object that `f` belongs to, using the default association name `machine`, i.e., the class name of `Machine` written with a first lower case letter. From there, we can access attribute `kind` that has one of the enumeration values `Mill`, `Drill`, or `Wash`. Based on that kind of machine, the next destination can be determined. The corresponding destination input buffer is assigned to the predefined variable `result`, indicating that this is the value to be returned.

In a concrete system, there will be several machine objects, i.e., instances of class `Machine`. Expression `Machine.allInstances()` then results in the set of currently existing `Machine` objects. We can manipulate that set by different operations, one of which is `any(expr)`. That operation selects one arbitrary element that complies to the parameter expression `expr`. Thus, if there were 2 machine objects of kind `Drill`, one of these is arbitrarily chosen. An implementation may consider additional information which machine is actually to be selected (e.g., the current load of the machines, their reliability, their age), but this is not in the scope of the OCL constraint at this stage of modeling. Finally, in order to extract the actual input buffer object, we navigate from `Machine` to association end `inputBuffer`.

OCL Types. OCL is a typed language, i.e., each OCL expression has a type that is either explicitly declared or can be statically derived. Generally, an OCL expression can be built out of subsequent basic expressions, separated from each other by dots and arrows. Evaluation of a left hand expression part determines the domain of possible result values which is called a *type* in terms of OCL. In turn, that type constitutes how the expression can be extended on the right hand side, i.e., it must be one of the features (attributes, operations, association ends) defined for the type determined by the left hand side expression. In order to reason about typed expressions, we have to have a *type system* as a basis that takes user-defined types (classes, interfaces, etc.) as well as fundamental and generic types into account. The latter ones form a core of predefined OCL types on the M1 level and are provided as a standard library in the official UML specification (see Figure 2.6).

For a tight integration of OCL into the common UML language definition, a corresponding metamodel for OCL types and expressions is needed that is still missing in the official UML 1.5 specification. There are publications that deal with defining a metamodel for OCL [RG99, BH00], resulting in an extensive metamodel for OCL types and expressions as part of OCL 2.0 [OMG03b]. Figure 2.7 shows a slightly modified version of the OCL types metamodel of the adopted OCL 2.0 specification.

The basic OCL metaclass is `Classifier`, taken from the UML core metamodel. Note that all subclasses of `Classifier`, in particular `Class`, `Interface`, and `DataType`, are implicitly included. The new metaclasses introduced for OCL are shown in grey boxes. `VoidType` is

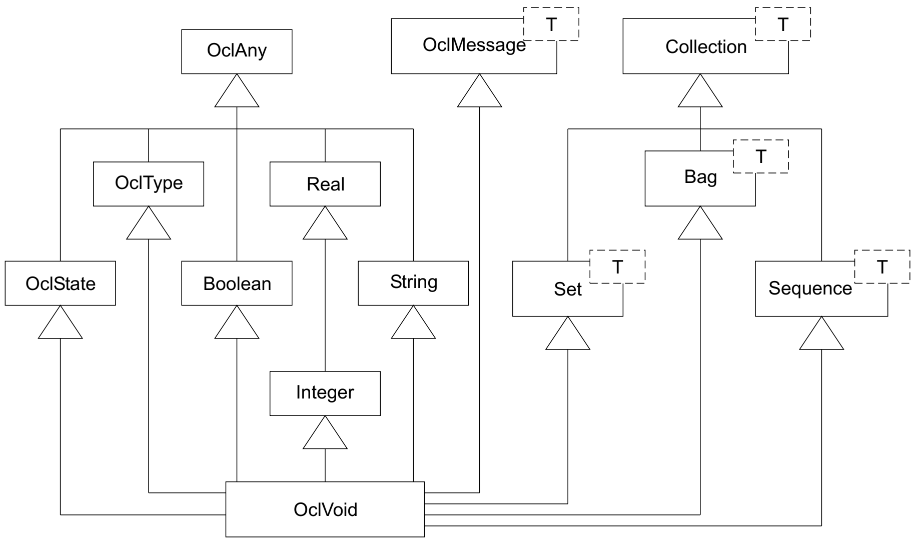


Figure 2.6: Predefined Standard OCL Types (Level M1) [OMG03b, Section 11.2, Figure 28]

the metaclass for the predefined OCL standard type `OclVoid` on level M1. `TupleType` is the metaclass for (parameterized) user-defined tuple types. Mathematical tuples together with the usual operations for manipulation (e.g., projection and product) are a new concept in OCL 2.0. For more details on tuples, we refer to the discussion in Section 2.3.3.3. Metaclass `OclModelElementType` represents those types that are needed to reference names of user-defined model elements, in particular, class names of Class Diagrams and states of State Diagrams. Instances of `OclModelElementType` are predefined enumerations on level M1, such as `OclType` and `OclState`. The definition of metaclass `CollectionType` and its subclasses for sets, ordered sets, bags (or: multisets), and sequences is of particular interest and deserves a separate section. The same holds for `OclMessageType` that is the metatype of the new OCL 2.0 concept of OCL messages.

Table 2.3 gives a summary of predefined OCL types and their corresponding metaclasses to explicitly show the interconnection of Figures 2.6 and 2.7.

Collections in OCL. Collections have a specified element type. All collection elements are of that type. Note that not all possible collection types are actually instantiated, as that would result in an infinite number of types when nested collections are taken in account. Though all these types conceptually exist, particular collection types are only instantiated when actually needed in an OCL expression.

Nested collections are not considered in the current UML 1.5 standard, i.e., when a nested collection would appear as the result type of an OCL expression, it is always implicitly flattened. E.g., assume that we have three `Machine` objects `obj1`, `obj2`, and `obj3`, i.e. expression `Machine.allInstances()`, is of OCL type `Set(Machine)` and results in `Set{obj1,obj2,obj3}`. Extending that OCL expression by navigating along associations to classes `InputBuffer` and `Item`, the OCL expression

```
Machine.allInstances().inputBuffer.currentItems
```

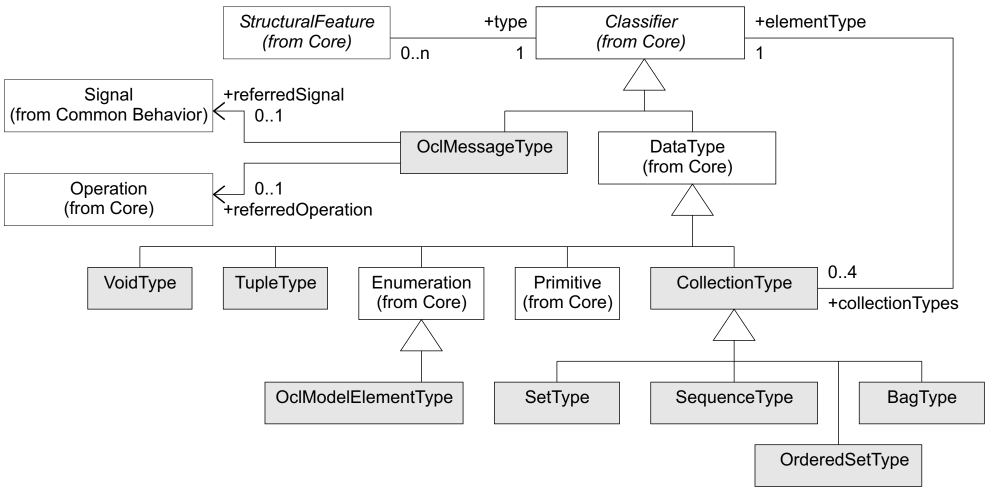


Figure 2.7: OCL Metaclasses (Level M2) [OMG03b, Section 8.2, Figure 5]

results in a set of set of items currently associated with the machine input buffers, i.e., the expected result type of that expression is `Set(Set(Item))`. But in the current OCL version as defined in UML 1.5, this nested set is implicitly flattened to type `Bag(Item)`, i.e., a flat multiset of items (note that in this particular example, the bag is of course a set, as items cannot be at two input buffers at the same time).

In contrast, OCL 2.0 now allows nested collections and introduces a new operation called `flatten()` to explicitly flatten nested collections if necessary. With the latter approach, we have to write

```
Machine.allInstances().inputBuffer.currentItems->flatten()
```

to get a simple set of items, i.e., `Set(Item)` as a result type. As nested collections are a reasonable concept that is also suitable for our intended OCL extension, we assume for the rest of this thesis that OCL collections are not automatically flattened.

OCL Messages. Finally, the concept of OCL messages has been newly introduced to OCL 2.0 to specify behavioral constraints over messages sent by objects. The notion of OCL messages is based on work presented by Kleppe and Warmer in [KW00, KW02]. Basically, an OCL message refers to a signal sent or a (synchronous or asynchronous) operation called. While signals sent are asynchronous by nature and the calling object simply continues its execution, synchronous operation calls make the invoking operation wait for a return value. In contrast, an asynchronous operation call is like sending a signal, such that a potential return value is simply discarded. For more details about messaging actions, see the action semantics of UML 1.5 [OMG03d, Section 2.24]. Note here that the UML action semantics also define *broadcast signal actions*, while a corresponding kind of OCL message is not yet defined.

The concept of OCL messages enables modelers to specify postconditions that require that specific signals must have been sent, operations must have been called, or operations must have been completely executed and returned.

Table 2.3: OCL Standard Library Types and their Metaclasses

Metaclass	M1 Level Type	Description
`Classifier`	`OclAny`	OclAny is the supertype of all types of the UML model and the predefined types of the standard library.
	`OclType`	For each `Classifier` in a UML model there is a corresponding instance in (power)type `OclType`.
`VoidType`	`OclVoid`	OclVoid is a type that conforms to all other types. The only instance of OclVoid is OclUndefined.
`OclModelElementType`	`OclState`	For each `State` in a UML model there is a corresponding enumeration literal in type `OclState`.
`Primitive`	`Real`, `Integer`	Common data types for numbers, boolean
	`Boolean`, `String`	values, and strings.
`CollectionType`	`Collection(T)`	Parameterized abstract type as a supertype for sets, bags, and sequences. The template parameter `T` is substituted by a concrete type, e.g., `Integer` or `Item`.
`SetType`	`Set(T)`	Parameterized type to manage sets of objects of a type `T`.
`BagType`	`Bag(T)`	Parameterized type to manage bags of objects of type `T`, i.e, objects may appear more than once in a bag.
`SequenceType`	`Sequence(T)`	Parameterized type to manage sequences of objects of type `T`, i.e, objects may appear more than once and are indexed by numbers.
`TupleType`	`Tuple(T)`	Parameterized type to define and reason about mathematical tuples. Actually, the template parameter `T` denotes the *sequence of types* that represent the tuple components.
`OclMessageType`	`OclMessage(T)`	Parameterized type to capture messages sent from a source object to a target object. Template parameter `T` denotes a signal or operation.

OCL messages are obtained by the message operator `^^` that is attached to a *target object*. For example, the OCL expression `vehicle^^move(aStation)` results in the sequence of messages `move(aStation)` that have been sent to the object determined by `vehicle` during execution of the considered operation – recall that the considered expression must have been specified in an operation postcondition. Each element of the resulting sequence is an instance of type `OclMessage(T)`. For example, the type of OCL expression `vehicle^^move(aStation)` is `Sequence(OclMessage(move(s:Station)))`.

One can make use of so-called *unspecified values* to indicate that an actual parameter does not need to have a specific value. Unspecified values are denoted by question marks, e.g., `vehicle^^move(?:Station)`. Parameter types can be omitted in OCL message expressions, but note that they might be necessary in order to refer to the correct operation when the operation is specified more than once with different parameter types.

For example, assume that operation `announceOrder(i:Item)` of an `Input Buffer` should send a request for transport to all known AGVs. The following postcondition uses a navigation to the station (here: machine) to which the input buffer belongs in order to obtain all AGVs.

```
  context InputBuffer::announceOrder(i:Item)
    post: machine.transporters@pre->forAll(vehicle:AGV |
                                    vehicle^^requestTransport(i:Item)->size() = 1)
```

By `machine.transporters@pre`, all AGVs known at the start of operation execution are obtained, i.e., the expression represents a *set of AGV objects* that were known when the operation started. The postcondition then requires that exactly one message `requestTransport(i:Item)` must have been sent to each AGV.

To check whether a message has been sent, the *hasSent operator* ^ can be used, e.g., the expression `vehicle^move(aStation)` results in true iff a message `move(aStation)` has been sent (at least once) to `vehicle` during execution of the considered operation. However, this operator cannot be used to restrict the number of times this message has been sent. More examples can be found in [OMG03b, Section 7.7.3].

A formalization of OCL messages is not covered in the formal semantics of the adopted OCL 2.0 specification, but a corresponding extension is proposed in [FM04].

Predefined Operations. For each of the standard library types, a number of operations is defined to access and compare objects and values. Some were already introduced in the examples discussed above, and we here only briefly summarize the predefined operations for `OclAny` in Table 2.4 and further discuss collection type operations. Operations for primitive types such as for types `Real` or `Integer` are defined in a straight forward way and do not need further explanations.

For all kinds of collections, operations like `size()`, `count()`, `includes()`, and `isEmpty()` are available. For more specific collection kinds, additional operations are defined. E.g., on sets of objects, operations like `union()` or `intersection()` as well as type casts `asBag()` and `asSequence()` are available.

For example, we require that each AGV object is associated with (at least) one input storage and one output storage (note the two equivalent notations to express this property) and that it knows all buffers in the system (as each AGV needs to access the positions of buffers).

```
-- each AGV object knows an input and an output storage and the
-- buffers of all stations
context AGV inv:
  self.stations->select(s:Station | s.oclIsTypeOf(InputStorage))->notEmpty()
  and
  self.stations->select(s:Station | s.oclIsTypeOf(OutputStorage))->size() > 0
  and
  self.buffers->size() = Machine.allInstances().inputBuffers->size()
                        + Machine.allInstances().outputBuffers->size()
```

The *iterate operation* needs a special explanation. Starting from a collection of elements, each element is subject to evaluation of an expression that results in accumulation by means of an (implicit) result variable. For instance, if we wanted to sum up the overall number of transformations on items, we can write

Table 2.4: Operations of OclAny

Operation	Return Type	Description
`obj = (obj2:OclAny)`	Boolean	Checks for equality of two objects. Operation = can also be used with inline notation, i.e., `obj = obj2`.
`obj <> (obj2:OclAny)`	Boolean	Dual to operation =, inline notation is also allowed, i.e., `obj <> obj2`.
`obj.oclAsType(t:OclType)`	`t`	Performs a type cast for object `obj`.
`obj.oclIsTypeOf(t:OclType)`	Boolean	Checks whether `obj` is an instance of type `t`.
`obj.oclIsKindOf(t:OclType)`	Boolean	Checks whether `obj` is of type `t` or one of its supertypes.
`obj.oclIsNew()`	Boolean	Used in postconditions to check whether an object has been newly created when executing the corresponding operation.
`obj.oclInState(s:OclState)`	Boolean	Checks whether `obj` currently is in state `s`, which is a state of one of the State Diagrams attached to `obj`'s class.
`obj.oclIsUndefined()`	Boolean	Evaluates to true if `obj` is not defined. Used in postconditions to check whether an object has been destroyed.

```
Machine.allInstances()->
  iterate(processedItems:Integer; res:Integer=0 | res + processedItems)
```

There are several predefined useful operations that are directly derivable from operation `iterate()`, e.g., `forAll()`, `exists()`, `select()`, `reject()`, `any()`, and `sum()`. The previous example can thus also be formulated by

```
Machine.allInstances()->collect(processedItems)->sum()
```

As there is a shorthand notation for `collect()` defined in OCL, we might even write

```
Machine.allInstances().processedItems->sum()
```

To give another example of a frequently applied kind of constraint, we require that item objects must have different identifiers to uniquely distinguish them. The corresponding invariant makes use of operation `forAll()` and two iterator variables `item1` and `item2`.

```
context Item inv:
  Item.allInstances()->
       forAll(item1 : Item | Item->allInstances()->
              forAll(item2 : Item |
                     item1 <> item2 implies item1.id <> item2.id))
```

Alternatively, a shorthand notation is allowed that replaces the two nested iteration operations `forAll()`:

```
context Item inv:
  Item.allInstances()->
       forAll(item1, item2 : Item |
              item1 <> item2 implies item1.id <> item2.id)
```

In the latest OCL version, operation `isUnique()` was introduced to express this property in a more compact form by:

```
context Item inv:
  Item.allInstances()->isUnique(i:Item | i.id)
```

Undefined Expressions. It can happen that an OCL expression evaluates to an undefined value. The following examples are all invalid OCL expressions.

```
-- (a)
       42 + 'drill'
-- (b)
       3.14 and true
-- (c)
       AGV.allInstances()->collect(a:AGV | a.oclAsType(Station))
```

The first example applies a parameter of the wrong type (i.e., a string) to an Integer value operation. The second example tries to apply a logical operation to a Real value, which is not defined. And the third expression is undefined, as subexpression `a.oclAsType(Station)` evaluates to undefined, as that kind of type cast is not possible in our model. In all cases, the result is referred to `OclUndefined`, i.e., the only instance of type `OclVoid`.

Expressions Metamodel. We have seen various examples with more or less complex OCL expressions. These are formed based upon a concrete syntax given by an attributed context-free grammar. In order to give a more general definition of OCL expressions on the level of an abstract syntax, an additional *OCL expressions metamodel* is built that shows the general structure an OCL expression may have. An overview of the expression types in that part of the OCL metamodel is given in Figure 2.8. Associations and well-formedness rules among these metaclasses and common UML metaclasses (e.g., Classifier, Operation, Attribute) then define the general structure of OCL expressions.

We do not go into more details of the expressions metamodel here and refer to the adopted OCL 2.0 specification for further readings [OMG03b, Section 8.3]. We are rather concentrating on the concrete OCL syntax in the following.

Concrete Syntax: Attributed Grammar. The metamodel-based approach of the OCL 2.0 specification achieves a separation between concrete and abstract OCL syntax. Basically, one can now define an own notation (e.g., a 'visual OCL') and map this notation to the abstract OCL syntax. However, the adopted OCL 2.0 specification suggests a concrete syntax that is compliant with the current OCL standard [OMG03b, Chapter 9]. This concrete syntax is defined by an attributed grammar that provides a mapping onto the abstract syntax. The motivation for

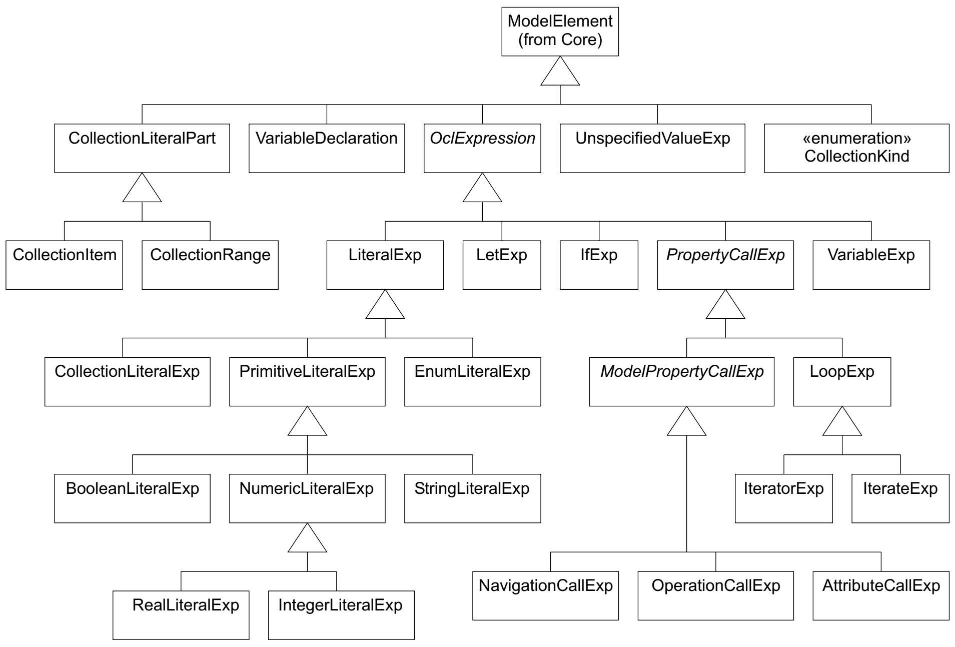


Figure 2.8: Types of the OCL Expressions Metamodel [OMG03b, Section 8.3.10, Figure 12]

taking an *attributed* grammar is '*the easiness of the construction and the clarity of this mapping*' [OMG03b, Section 9.1].

The attributed grammer comes with production rules in EBNF that are annotated with synthesised and inherited attributes as well as disambiguating rules. The attributes are necessary to provide rules for the purpose of well-formedness, in particular type checking. *Inherited attributes* are defined for elements on the right hand side of production rules. Their values are derived from attributes defined for the left hand side of the corresponding production rule. For instance, each production rule has an inherited attribute `env` (short for 'environment') that represents the rule's namespace. *Synthesised attributes* are used to keep results from evaluating the right hand sides of production rules. For instance, each production rule has a synthesised attribute `ast` (short for 'abstract syntax tree') that constitutes the formal mapping from concrete to abstract syntax. *Disambiguating rules* allow to uniquely determine a production rule if there are syntactically ambiguous production rules to choose from.

As an example, we consider some of those production rules in the attributed grammar that are defined for different kinds of operation calls in OCL. A production rule name is appended by `CS` to distinguish between concrete syntax element and its corresponding metaclass `OperationCallExp`.

```
-- Production rules for OperationCallExpCS:
[A] OperationCallExpCS ::= OclExpressionCS[1]
                           simpleNameCS OclExpressionCS[2]
...
[C] OperationCallExpCS ::= OclExpressionCS '.'
                            simpleNameCS '(' argumentsCS? ')'
...
[H] OperationCallExpCS ::= simpleNameCS OclExpressionCS
```

Option [A] is used for infix operations (e.g., `4 + 2`), option [H] is for unary prefix expressions (e.g., unary operation `not` applied to boolean expressions), while option [C] is the rule for a common operation call. All other options are omitted in this example. By providing additional disambiguating rules, it is guaranteed to be able to uniquely determine one of the possible optional production rules. In option [C], the context classifier is given by the result gained from applying production rule `OclExpressionCS`, i.e., after `OclExpressionCS` on the right-hand side is deduced, we know its type (on level M1) and may refer to it by `OclExpression.ast.type`. Furthermore, a 'simple' name of type `String` is gained from applying production rule `simpleNameCS`, and a list of arguments in form of a sequence of classifier instances is gained from production rule `argumentsCS`. In order to be able to correctly deduce the non-trivial non-terminals on the right-hand side, we provide them the current list of names that are visible from the current expression position. This information is kept in the inherited attribute called `env` of a special type called `Environment` with corresponding manipulation operations. In our example it is just necessary to forward the environment information:

```
[C] OclExpressionCS.env = OperationCallExpCS.env
[C] argumentsCS.env = OperationCallExpCS.env
```

A mapping from the concrete to the abstract syntax is defined by re-typing synthesized attribute `ast` in each production rule to the corresponding expression metaclass. In the case of `OperationCallExpCS`, we therefore define

```
OperationCallExpCS.ast : OperationCallExp
```

In order to store the information gained from applying a rule (e.g., option [C]), the abstract syntax tree variable has to be updated. This is done by applying values from evaluating the subexpressions of the right-hand side to features of the abstract syntax tree variable `ast`. In the case of operation calls, model elements of level M1 have to be searched and stored, i.e., a source classifier instance, an instance of an operation, and a sequence of argument classifier instances. For the source and the arguments, we simply pass the abstract syntax trees of `OclExpressionCS` and `argumentsCS` to the features `source` and `arguments` of `OperationCallExpCS.ast`, as shown below.

```
[C] OperationCallExpCS.ast.source    = OclExpressionCS.ast
    OperationCallExpCS.ast.arguments = argumentsCS.ast
    OperationCallExpCS.ast.referredOperation =
      OclExpressionCS.ast.type.lookupOperation(simpleNameCS.ast,
           if argumentsCS->notEmpty() then
                arguments.ast->collect(type)
           else Sequence{}
           endif)
```

For the operation name (that is of type `String` due to applying rule `simpleNameCS`), we have to determine a corresponding operation instance (on level M1). We here make use of operation

```
lookupOperation(name:String, paramTypes:Sequence(Classifier)) : Operation
```

This operation is one of the metalevel operations that are additionally defined for metaclass `Classifier` in the context of OCL. On a given instance `c` of metaclass `Classifier`, operation `lookupOperation(name,paramTypes)` returns an operation object (of level M1), if there is an operation defined for `c` with matching name and parameter types. If there is no such operation, `OclUndefined` is returned.

It is remarkable that there is a direct correspondence between the proposed concrete syntax by means of an attributed grammar and the abstract syntax defined for OCL types and expressions. The mapping is implicitly given by the attributes of the grammar. Note that OCL users will not be confronted with the abstract syntax of OCL expressions - they will rather formulate OCL constraints based upon a concrete language, preferably in the standard textual form as proposed in the official UML specification. But now, one can think of well-defined alternative styles, e.g., graphical representations of constraints as it was proposed in [BKPPT00]. Such approaches can now be established by simply providing a mapping to the abstract syntax of OCL.

2.3.3.2 OCL Semantics

At this point, readers should be able to interpret OCL constraints in an intuitive manner. Nevertheless, a semantics has to be defined in order to be able to precisely answer the question:

> Given the overall system state of a UML model (i.e., a *snapshot*), what is the actual result from evaluating an OCL expression over that snapshot?

In recent years, different formal semantics have been published that define (parts of) earlier versions of OCL, e.g., [RG98, CK01, BW02]. Interestingly, OCL 2.0 provides *two approaches* for the semantics of OCL. First a semantics definition is given by a set-theoretic mathematical approach, based on [Ric01]. This will be discussed in more detail in the next chapter. The other approach defines the semantics on the level of the UML metamodel, based on the report 'Unification of Static and Dynamic Semantics for UML' [KW01]. The structure is shown in Figure 2.9. OCL metatypes as introduced before (cf. Figure 2.7) are found in subpackage `Types` of package `Ocl-AbstractSyntax`.

Figure 2.9 shows how the packages relate to each other, and to the packages from the abstract syntax. It shows the following packages:

- The `Domain` package describes the values and evaluations. Note that this package resides on layer M1 of the 4-layer architecture, while the abstract syntax resides on layer M2. The package is divided into two subpackages:
  - The `Values` package describes the semantic domain, i.e., the set of possible values. It represents the values that OCL expressions may yield as result.
  - The `Evaluations` package describes the evaluations of OCL expressions. It contains the rules that determine the result value for a given expression.
- The `AS-Domain-Mapping` package describes the associations of the values and evaluations with elements from the abstract syntax, i.e., this package links the domain (on layer

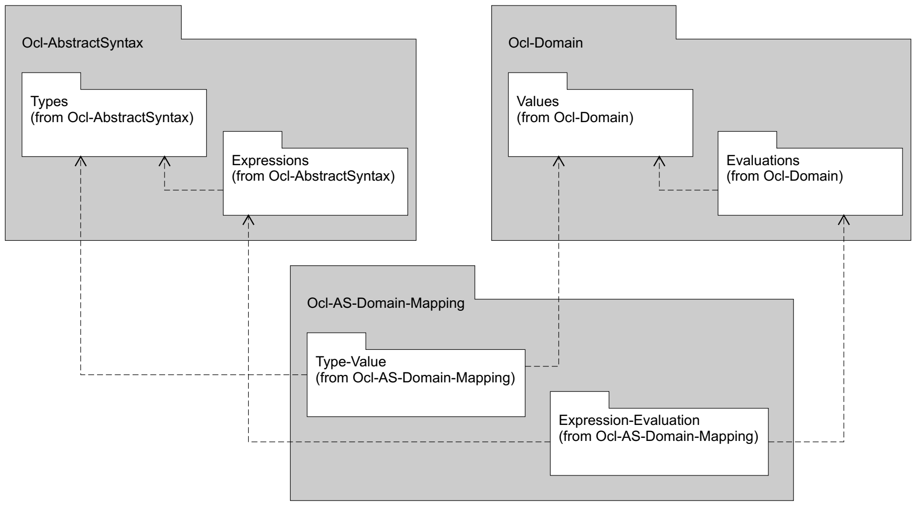

Ocl-AbstractSyntax
Types (from Ocl-AbstractSyntax)
Expressions (from Ocl-AbstractSyntax)
Ocl-Domain
Values (from Ocl-Domain)
Evaluations (from Ocl-Domain)
Ocl-AS-Domain-Mapping
Type-Value (from Ocl-AS-Domain-Mapping)
Expression-Evaluation (from Ocl-AS-Domain-Mapping)

Figure 2.9: OCL Semantics: Metamodel Packages [OMG03b, Section 10.1, Figure 14]

M1) with the abstract syntax (on layer M2). Note that the `AS-Domain-Mapping` package itself cannot be positioned in one of the layers. It is also divided into two subpackages:

- The `Type-Value` package contains the associations between the instances in the semantics domain and the types in the abstract syntax.
- The `Expression-Evaluation` package contains the associations between the evaluation classes and the expressions in the abstract syntax.

To summarize the approach, the semantics of an OCL expression is given by associating each value defined in the semantic domain with a type defined in the abstract syntax, and by associating each evaluation with an expression from the abstract syntax. In turn, the value yielded by an OCL expression based on a given snapshot of the UML model is the result value of its evaluation.

2.3.3.3 Discussion

In recent years, complaints about the concrete OCL syntax could be observed, e.g., [Pad00, Section 5]. For UML 2.0, the metamodel approach of OCL 2.0 might enable tool developers to overcome this problem in the future. Tools could employ their own constraint language in UML 2.0; they only have to provide a mapping to the OCL metamodel. Thus, a tool does not have to stick to the concrete OCL syntax provided in the adopted OCL 2.0 specification. For example, there is already work available on a visual OCL variant [BKPPT00, BKPPT01, KTW02]. Related work with so-called *constraint diagrams* and *constraint trees* is published in [GHK99, KH02].

Mandel and Cengarle have shown that OCL does not yet have the expressive power of a relational algebra [MC99]. A relational algebra is a collection of operations that take relations (or: sets of tuples) as operands and result in a relation. Relational algebra uses operations from mathematical set theory and specific operations developed for manipulation of data in the area of relational databases [EN00]. Fundamental operations are *select*, *project*, *difference*, *union*, and *(cartesian) product*, and several additional useful operations can be derived, e.g., *intersection*, *division*, and *join*. In standard OCL, operations *select*, *difference*, and *union* are already available, but as OCL currently does not have a notion of tuples, operations *project* and *product* are not supported. With the introduction of a `Tuple` type and corresponding operations on tuples [AB01] and the adoption in OCL 2.0 [OMG03b], OCL is getting *almost* the expressive power of a relational algebra and can be used as a query language similar to SQL. In this context, Balsters explains in [Bal03] that the relational *join* operation is still not suffiently supported in OCL 2.0.

Besides the issues addressed in the OMG OCL 2.0 RfP [OMG00a], there are a number of further unresolved issues in the OCL language definition, maybe the two most important and frequently discussed among these are

- the role of undefined values, in particular in combination with logical operations [Ric01, Section 4.2.5], and

- the interpretation of recursion in OCL constraints. Consider the operation postcondition

  ```
  context AClass:fac(n:Integer) : Boolean
    pre:  n >= 0
    post: result = if (n=0) then 1
                            else n * fac(n -1)
                  endif
  ```

  In UML 1.5 as well as in the OCL 2.0 specification it is allowed that the right-hand-side of such a definition may refer to the operation actually being defined (i.e., the definition may be recursive) *as long as the recursion is not infinite*. But this statement still does not solve what expressions like

  ```
  context AClass:op() : Boolean
    post: result = self.op()
  ```

  are resulting in. Brucker and Wolff show two possible interpretations [BW02]. Either such expressions are illegal, or the result is the undefined value `OclUndefined`. Unfortunately, the first solution requires a notion of well-formedness that is undecidable, i.e., not machine-checkable. The second is consistent with the *least fixpoint* semantics [Win93] and was proposed in the Amsterdam Manifesto on OCL [CKM$^+$99, CKM$^+$02].

From the OCL language concepts, the type system is by now in a considerable stable situation. Though, the role of type `OclType` is discussed controversially in different publications. In the sequel, we therefore consider once more the type system of the OCL 2.0 specification and review the definition and usage of type `OclType` within the OCL language definition and different approaches in literature.

The Role of OclType. In OCL expressions, it is sometimes necessary to access user-defined classes, e.g., to perform type casts, to check for a certain subtype, etc. There are different possibilities to provide access to such instances of metaclass `Classifier`:

1. The current standard as defined in UML 1.5 uses `OclType` and refers to is as a *metatype*, as it reads in (cf. [OMG03d, Section 6.8.1.1]):

   > *All types defined in a UML model, or pre-defined within OCL, have a type. This type is an instance of the OCL type called OclType. Access to this type allows the modeler limited access to the meta-level of the model. This can be useful for advanced modelers.*

   There are even pre-defined operations provided for `OclType` to further access metalevel features, e.g., operations to get the list of attribute names, association end names, and direct as well as indirect supertype names (surprisingly, operations to extract subtypes are missing).

   Modelers are thus able to access the metalevel (level M2). Note that this breaks up the 4-layer architecture underlying the UML modeling approach.

2. In contrast, Baar and Hähnle suggest to model `OclType` as a pure metatype [BH00] without access for the modeler (see Figure 2.10). Operations as mentioned above do not need to be defined on `OclType` in their approach, as the UML core metamodel already provides such means, either directly or indirectly (by navigation along associations in the UML core metamodel). I.e., that approach also allows direct metamodel access. It is worth noting in this context that nested collections are not possible in that metamodel.

3. Richters proposes in his OCL type metamodel a metatype `OclTypeType` with `OclType` as its only instance on level M1 [Ric01, Section 6.4]. Moreover, all classifiers of the referred UML user model are maintained by an additional `ObjectType`, i.e., a bijective function between instances of `ObjectType` and user-defined classes is established. Semantically, the domain of an object type is the set of object identifiers defined for the class and its children.

4. In the adopted OCL 2.0 specification, `OclType` is now residing on level M1 as a subtype of `OclAny`. It is regarded as an enumeration of the classifier names of the referred UML user model. Its corresponding metaclass is `OclModelElementType`, a subclass of `Classifier` (more precisely, `OclModelElementType` is a subtype of `Enumeration`). The operations previously defined on `OclType` are no longer available, as instances of `OclType` are only used as parameters in some operations of type `OclAny`, namely in operations

   ```
   1.  oclIsKindOf(t:OclType) : Boolean
   2.  oclIsTypeOf(t:OclType) : Boolean
   3.  oclAsType  (t:OclType) : instance of OclType -- Arguable! See below.
   ```

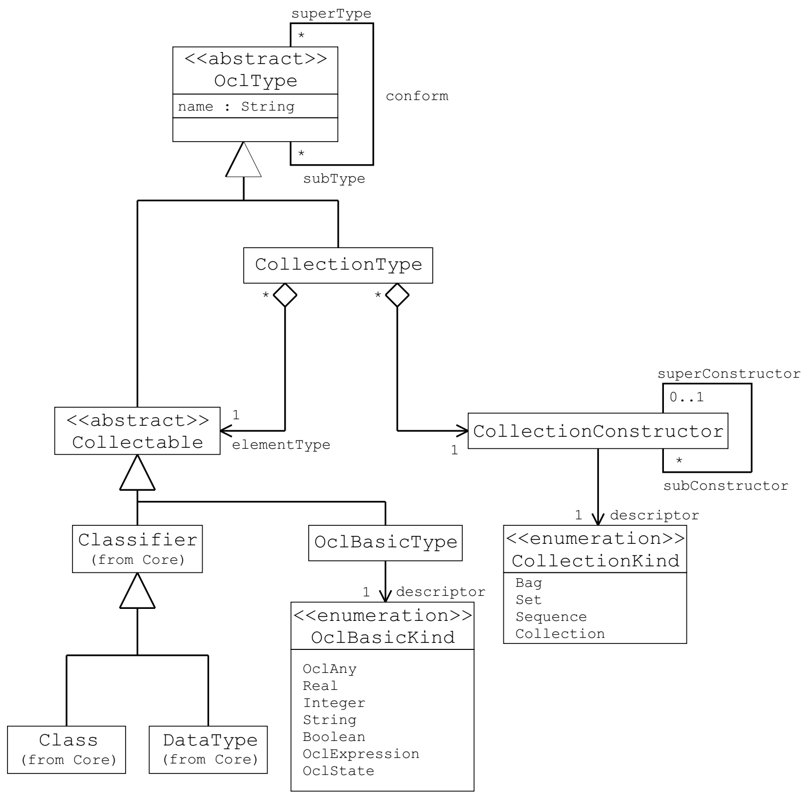


Figure 2.10: OCL Type Metamodel Proposal by Baar and Hähnle [BH00]

Note that `oclAsType()` actually does not return an instance of `OclType`, as this refers to an enumeration literal, e.g., `OclType::InputBuffer` when `OclType` is an enumeration type. That operation rather returns an object re-typed to the used-defined type *with the name* represented by parameter `t`. It is better to specify in this case:

```
obj.oclAsType(t:OclType) : OclAny

context OclAny::oclAsType(t : OclType) : OclAny
  post: if self.oclIsKindOf(t) then
           result = self and result.oclIsTypeOf(t)
        else
           result = OclUndefined and result.oclIsTypeOf(OclVoid)
        endif
```

Parameter `t` denotes by definition an instance of metaclass `Classifier`, as `OclType` is a powertype over classifiers. Only if `t` is a subtype of the current context determined by `self`, re-typing can take place. In all other cases, `OclUndefined` is returned.

With the latter approach in mind, it is better to model `OclType` as a powertype for `OclAny`, as illustrated in Figure 2.11. The powertype concept is a dependency relationship among a generalization and gives access to specialized types as instances. It is thus a more natural way to represent the elements of `OclType` that does not need to redefine classifiers in an additional enumeration.

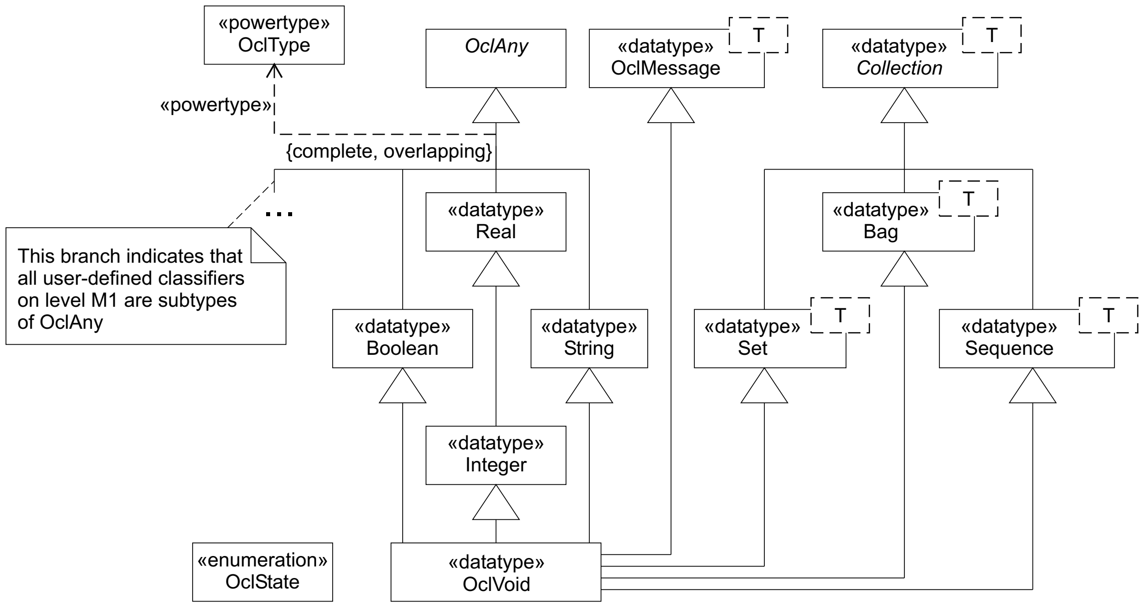


Figure 2.11: OCL Standard Library Types Proposal

OCL and State Diagrams. In UML 1.5 there is a specific new kind of invariant defined that has not yet received much attention; so-called *state invariants* can be formulated and attached to a specific state s in a State Diagram associated to class c [OMG03d, Section 2.5.2.13]. Basically, this is equivalent to a common OCL invariant of the form

```
context c inv:
  self.oclInState(s) implies <stateInvariantExpression>
```

Surprisingly, there is still no semantics definition for state invariants and state-related operation `oclInState(s:OclState)`, and even in the adopted OCL 2.0 specification this issue is missing. To overcome this, we are going to integrate a general State Diagram description to the current semantic model of OCL in Chapter 4.

2.3.4 UML Extension Mechanisms

Although UML is a general purpose modeling language, it has mechanisms that can be applied to tailor UML to specific domains, i. e., to perform a specialization by introducing restrictions on the general UML metamodel (layer M2 in Table 2.1). In terms of UML, the facilities for metamodel specializations are referred to as *UML extensibility mechanisms*. The standard UML extensibility mechanisms are stereotypes, (user-defined) tagged values, and constraints. These modeling elements are capable of adapting the UML semantics without changing the actual UML metamodel. They are often called *lightweight extensibility mechanisms*, in contrast to a direct manipulation of the UML metamodel, which can be seen as heavyweight extensibility mechanisms (e.g., adding new meta classes and associations).

Stereotypes are classifications of existing model elements. They are an alternative way to introduce specialized model elements without adding subclasses in the actual metamodel. It is possible to specify hierarchies among stereotypes, and besides the textual annotation in double square brackets that identifies a stereotype (e.g., ≪`metaclass`≫), it is also allowed to introduce a completely new notation (i. e., a graphical symbol) for a stereotype. In practise, stereotypes are mainly applied to model elements in order to embed their corresponding diagram type into the different stages of a software development process. E.g., in [OMG03d, Section 4.3.5] the stereotypes ≪`entity`≫, ≪`control`≫, and ≪`boundary`≫ are introduced to specialize the notion of classes. The stereotype semantics are informally described and a graphical notation is introduced for each stereotype. The stereotypes are used to model UML classes as passive entities without initiative to interact, control classes that manage interactions between objects, and 'peripheral' boundary classes that build an interface to actors outside of the regarded system. *Tagged values* are characteristics of UML metamodel elements or stereotypes that restrict the model elements by key/value pairs. Additionally, constraints can be defined to specify consistency rules of and between model elements, often also denoted as well-formedness rules. Constraints can be expressed in natural language or by use of the Object Constraint Language (OCL, see Section 2.3.3). Although there currently is no official specification that provides a standard for applying the presented extension mechanisms to specialize standard metamodels like the UML, a white paper on the UML profile concept has been released by the OMG Section [OMG99]. In that article, first the general requirements for adequate extension mechanisms are identified, before the notion of a profile is defined as follows:

> *A Profile is a specification that specializes one or several standard metamodels, called the reference metamodels. [...] A Profile is a consistent definition context for elements such as, but not limited to, well-formedness rules, tagged values, stereotypes, constraints, semantics expressed in natural language, extensions to the standard metamodel and transformation rules.*

Popular examples of proposed UML profiles are UML-RT [SR98], the UML Profile for CORBA [OMG00b], and the two UML Profiles for Business Modeling and for Software Development Processes published in the UML 1.5 specification [OMG03d, Chapter 4].

2.4 UML and Time

The UML standard already provides a variety of means to construct object-oriented models. However, several aspects of real-time systems development require additional constructs that are not directly covered in standard UML. UML can be naturally extended to better define and design real-time systems using the extension mechanisms described before in Section 2.3.4. Resulting models are then tailored more accurate towards the specific domain, and implementation can be performed in a more straightforward way.

In the following, we describe

1. language elements the UML itself provides to specify timing issues (Section 2.4.1),

2. approaches that extend UML to better model *architectural aspects* of real-time systems (Section 2.4.2), and

3. approaches that extend State Diagrams to capture *behavioral aspects* of real-time systems (Section 2.4.3).

2.4.1 Time and Timing Constraints in Standard UML

For modeling aspects of real-time, UML provides language elements for timing marks, time expressions, and timing constraints to be used in Sequence and Collaboration Diagrams. Figure 2.12 shows a Sequence Diagram with a phone call scenario [OMG03d, Section 3.60.3]. In that figure, the annotations in curly brackets are timing constraints composed by time expressions. On level M2, `TimeExpression` is seen as a statement to define the absolute or relative time of occurrence of an event. There is no particular format defined for time expressions and timing constraints in UML, and annotations as shown in Figure 2.12 must therefore be seen as suggestions for UML users. Labels a,b,c, and d in Figure 2.12 are timing marks that can be referred to in time expressions. An alternative notation is shown at the lower right bottom of Figure 2.12. A corresponding time expression could be `f.receiveTime - e.sendTime < 1 sec`, but note that the notation is ambiguous when the arrows are horizontal, because sending and receiving time cannot be distinguished.

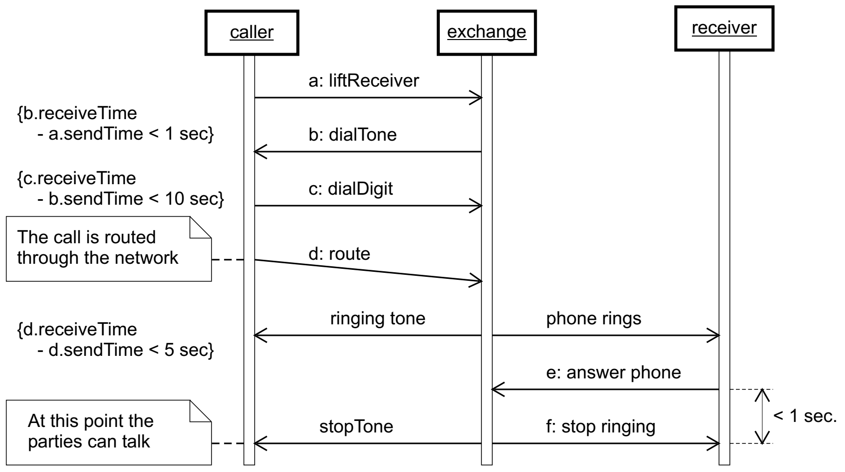

caller
exchange
receiver
a: liftReceiver
{b.receiveTime - a.sendTime < 1 sec}
b: dialTone
{c.receiveTime - b.sendTime < 10 sec}
c: dialDigit
The call is routed through the network
d: route
ringing tone
phone rings
{d.receiveTime - d.sendTime < 5 sec}
e: answer phone
At this point the parties can talk
stopTone
f: stop ringing
< 1 sec.

Figure 2.12: UML Sequence Diagram Example [OMG03d, Section 3.60.4, Figure 3-55]

As *message* is standing for a specification concept in UML, actual messages sent between objects are called *stimuli* in UML. Functions `sendTime()` (i.e., the time at which a message is sent by an object) and `receiveTime()` (the time at which a message is received by an object) are applied to stimuli names to yield a time. Unfortunately, there is no semantics description in the UML specification about those times. UML even suggests that users invent further timing functions when needed for particular domains or implementations, e.g., `executionTime()`. A semantics, however, is not in the scope of UML, and an appropriate mapping to a well-defined time expression has still to be provided.

The *UML Language User Guide* shows a way to apply time expressions to operations [BRJ99, page 324] with standard UML: A note with tagged value `semantics` is attached to an operation. In that note, a time expression then specifies the operation's time complexity.

This typically represents the minimal/maximal time of expected completion of an operation execution. Such specifications can be used in different ways. E.g., the resulting running system can be compared with the asserted times specified in the model. Alternatively, by adding up (asserted or actual) operation times, compound times of entire transactions can be calculated.

In the *UML 2.0 Superstructure Proposal* [OMG03f], Sequence Diagrams are now equipped with improved modeling elements for time bounds. Arcs that represent messages that are sent between objects can now be annotated by expressions that refer to

- duration observations (e.g., '`Code d = duration`'),
- duration constraints (e.g., '`{d..3*d}`'),
- time observations (e.g., '`t=now`'), and
- time constraints (e.g., '`{t..t+3}`').

Moreover, *Timing Diagrams* are one of the new kinds of diagrams in UML 2.0 [OMG03f, Section 14.4]. Timing diagrams model changes of object states over time along a linear time axis. Basically, modelers can specify conditions that imply object state changes as part of object lifelines. The behavior of objects as well as interactions among objects can thus be restricted. Though some examples are provided and a guideline for the basic graphical notation is given in the UML 2.0 superstructure proposal, the semantics description of timing diagram is still incomplete, e.g., the semantics of tick mark values and timing rulers is unclear.

2.4.2 Modeling Real-Time System Architectures with UML

To overcome the limited means for modeling time and related aspects in UML, several competing and complementary proposals to extend UML have been developed. These are specifically tailored to the domain of real-time systems, i.e., focusing on aspects like system architecture, communication mechanisms, and real-time constraints. In this context, the approach known as RT-UML by B.P. Douglass completely sticks to standard UML [Dou00]. Douglass restricts on using stereotypes to explain the purpose of model elements that commonly occur in the real-time domain. E.g., a variety of message stereotypes is presented to represent synchronization and message arrival. The CASE tool Rhapsody is an implementation of RT-UML[3].

Another widely recognized work of real-time systems modeling with UML is the UML-RT profile based on the ROOM methodology [SR98, Sel99, RS01] and implemented by Rational RoseRT[4]. The key concepts of ROOM are *capsules*, *ports*, *connectors*, *protocols*, and a specific variant of State Diagrams. Capsules represent components of real-time systems and have explicit external interfaces specified by ports. Connectors between ports represent (logical) communication between capsules. Protocols are defined independently of capsules and represent reusable interfaces for communication by specifying roles for participants with allowed incoming and outgoing messages. Ports implement these participants, i.e., two ports may communicate when their corresponding protocol roles are compatible.

[3]http://www.ilogix.com

[4]http://www.rational.com/products/rose

There are a number of other tools available that deal with development of real-time systems, e.g., the Telelogic TAU[5] suite uses SDL combined with UML and ARTiSAN offers a tool called Real-Time Studio[6]. An extensive list of UML tools together with their provided functionality is available online[7].

The fact that a number of different (tool-based) modeling techniques have emerged for developing real-time systems led to a variety of different notations and terminologies. The OMG therefore issued a Request for Proposals on a *UML Profile for Schedulability, Performance, and Time (SPT)* that shall provide a common framework by means of UML that one the one hand covers this diversity but one the other hand still offers flexibility for further specializations. An extensive submission on that Request for Proposals has been submitted by leading CASE tool vendors in the domain of real-time systems development (i.e., ARTiSAN, I-Logix, Rational, Telelogic, TimeSys, and Tri-Pacific). That submission is at time of writing still being reviewed by the OMG [OMG03c].

All UML tools and the UML profile for SPT have only limited support for temporal (constraint) specification. Temporal operators are basically timers with a fixed duration, e.g., `after(t)`. Currently, such properties can only be formulated in (stereotyped) notes without a standard semantics. It is thus not possible to verify whether a UML model satisfies temporal constraints. Different working groups therefore attempt to provide UML with a formal semantics, e.g., the pUML initiative (precise UML)[8] or the 2U consortium (unambiguous UML)[9].

2.4.3 Time-Annotated State Diagrams

While there are several publications available on introducing timing aspects in different behavior-related UML diagrams (e.g, [EW01] for Activity Diagrams), this Section focuses on approaches regarding timed variants of UML State Diagrams. As there are also a significant number of works on Harel Statecharts in this context published, we also briefly mention these, as they have heavily influenced more recent works on UML State Diagrams.

In 1987, Statecharts were introduced by Harel [Har87] to overcome the limitations of classical flat, unstructured state-transition diagrams. Statecharts basically introduce hierarchical states and additionally support parallel substates and broadcast communication. The operational semantics of Statecharts was first defined by Harel et al. in [HPSS87], providing a notion of steps and microsteps. That semantics bases on the so-called *synchrony hypothesis* that assumes that the system instantaneously reacts on inputs coming from the environment. The environment is seen as a discrete process sending inputs at successive points of time. It is assumed that the system's reactions on inputs at time t are completed before the inputs at time $t + 1$ occur.

However, this semantics showed some drawbacks, in particular, causal paradoxes could occur due to negated events in transitions. This problem was overcome by Pnueli and Shalev in [PS91] by a notion of globally consistent steps that guarantees the causality principle. Basically,

[5]http://www.telelogic.com
[6]http://www.artisansw.com
[7]http://www.jeckle.de/umltools.htm
[8]http://www.puml.org
[9]http://2uworks.org

they forbid that the cause attached to a transition may appear as a negated consequence in the same step. Several variants of Statecharts have also been published by other authors with different semantics definitions. But it is out of scope of this thesis to give a detailed overview of these works, and we refer to [Bee94, Lev97] for further readings.

In 1996, Harel and Naamad pointed out in their article on 'The STATEMATE Semantics of Statecharts' [HN96]:

> *Being an unofficial language, statecharts clearly have no official semantics, and researchers are free to propose semantics as they see fit. However, the only implemented and working semantics for statecharts has for many years been the one described here. [. . .]*
>
> *The main difference [. . .] was whether changes that occur in a given step (such as generated events or updates to the values of data items) should take effect in the current step or in the next one.*

Harel originally proposed to allow such effects within the same step [HPSS87], while in the STATEMATE semantics, generated events and value updates take effect in the next step (which, however, may happen still at the same point of time).

STATEMATE supports discrete simulation time with a global clock. In the asynchronous execution mode, the clock is only incremented at (super-)step boundaries to the next relevant clock tick, determined by counters handling time-outs and scheduled actions.

Statecharts became also prominent in the area of *object-oriented system modeling*, especially with the emergence of the first UML specifications after fall 1995. As the semantics of UML State Diagrams is only informally specified in the official OMG specifications, many researchers have published works on formalizations of (subsets of) UML State Diagrams in recent years, e.g., [BCR00, JEJ02, Kus01, Kwo00, LMM99b, LP99, RACH00, SKM01, Bee01].

Object-oriented UML State Diagrams exhibit a behavior different from structural Statecharts like Harel or STATEMATE Statecharts, the most important of which are

- point-to-point communication between objects instead of broadcast via channels,
- an implicit event queue storing incoming events instead of an event set,
- reaction on one event at a time by dispatching instead of reacting to all current input events,
- input events may exist for an arbitrary time instead of one time unit only,
- non-instantaneous communication instead of instantaneous communication,
- activities that take time in addition to instantaneous actions,
- distinction between types and instances instead of pure instance level modeling,
- encapsulation instead of separation of data and control.

Explicit Timing Mechanisms in State Diagrams. Formalizations that explicitly introduce *timing mechanisms* to Statecharts, let it be Harel's, STATEMATE, or UML State Diagrams. The following list provides an overview of existing work in this area without claiming that this list is complete.

- Kesten and Pnueli [KP92] introduce *Timed Statecharts* in which transitions are annotated by timing intervals, denoting the lower and upper bounds of a transition. A global clock is used for synchronous progress of time.

- Leung and Chan [LC96] introduce time-annotated transitions to Harel Statecharts, using *Duration Calculus* [CHR91] to define the underlying semantics. This implies some semantic changes w.r.t. the synchrony hypothesis as well as duration and consistency of actions and events.

- Maggiolo-Schettini and Peron [MSP96] use time durations associated to transitions. In this context, Levi's PhD thesis [Lev97] defines a compositional timed Statechart semantics by a translation to a process language called $\mathcal{TSP}$ with discrete time.

- Petersohn and Urbina [PU97] present a timed semantics of STATEMATE Statecharts by clocked transition systems with internal clocks and discrete time.

- Damm et al. [DJHP98] define a variant of STATEMATE Statecharts dedicated to perform compositional model checking. The semantics differs from the article by Harel and Naamad [HN96] in the sense that it provides a compositional semantics based upon synchronous transition systems.

  The article focuses on the asynchronous semantics (or super-step semantics) of the STATEMATE simulation tool, in which it is distinguished between internally and externally generated events. External events are only consulted at the first step and are communicated to the environment not until completion of a super-step, while internal events will be sensed already in the next step.

- Eshuis and Wieringa [EW00] propose a formal real-time semantics for UML State Diagrams at requirements level, adapting the STATEMATE semantics of [HN96]. Local variables, real-time, point-to-point communication, synchronous communication, and dynamic creation and deletion of objects are addressed.

- David, Möller, and Yi [DM01, DMY02] give a translation of restricted UML State Diagrams to flat UPPAAL timed automata. Among the preserved State Diagram concepts are state hierarchy, parallel composition, synchronization of remote parts, and history entries. Additionally, they equip State Diagrams with local clocks.

- Del Bianco, Lavazza, and Mauri [DLM02] define – based on previous own work [LQV01] – a timed UML State Diagram semantics by a translation to a first order temporal logic called TRIO. Though, several restrictions on UML State Diagrams are applied. Newly introduced specification means include concurrent events attached to transitions, guards with references to events to formulate *negated events*, and extension of the timeout mechanism `After(t)`, such that an interval `After(a,b)` may be specified, in which a state may be left. It is also possible to get the latest *occurrence time* of a transition.

- Knapp, Merz, and Rauh [KMR02] describe model checking of UML State Diagrams in a prototype implementation called HUGO/RT, in which each time-annotated State Diagram is translated to a flattened UPPAAL timed automata and an additional automata is generated for each event queue. The current status does not cover parameters attached to events, deferred events, history states, and more than one instance of a class.

- Burmester defines in his thesis a UML State Diagram variant called *Realtime Statechart* that extends state descriptions in various ways [Bur02]. One focus of that work is to generate real-time code (Realtime Java code).

  All standard features except elapsed time events and change events are kept in this approach. Additionally, Realtime Statecharts may be equipped with a number of local clocks. Each state may have a *timing invariant* that specifies the latest time it should be left again, similar to invariants of timed automata. *Clock resets* are attached to entry- and exit-operations to reset local clocks. Worst-case execution-times (wcet) are attached to operations that specify how long execution of an operation will take. A do-activity may have a specified period, as do-activities are interpreted as being operations that are periodically executed until a triggering event is dispatched. Transitions may additionally carry time guards, clock resets, a deadline, a wcet, a priority, and synchronization signals.

  Due to the number of introduced concepts, several methods are investigated to check temporal consistencies among a Realtime Statechart. A formal semantics of Realtime Statecharts is given by a mapping to *extended hierarchical timed automata*, that in turn are based on hierarchical timed automata as presented in [DM01].

2.5 Contributions of the Chapter

The contributions of this chapter can be summarized as follows.

- A review of selected parts of the UML is given, i.e., UML Class Diagrams, State Diagrams, the Object Constraint Language, and UML extension mechanisms.

- A proposal for a better representation of built-in type `OclType` within the OCL Standard Library is presented. Though access to the metamodel cannot be abolished, the proposed approach offers a way to access the metalevel M2 in a controlled way. As a consequence, a number of issues that are currently ill-defined in the OCL 2.0 specification could be resolved [Fla04], e.g., specifications for operations like `oclAsType()` can now be formulated by means of OCL postconditions.

- A review of timing issues within the different UML specifications and corresponding profiles is given, with an emphasis on timing annotations on behavioral elements such as messages sent and state transitions.

Chapter 3

Formal Verification

Beware of bugs in the above code;
I have only proved it correct,
not tried it.
– Donald E. Knuth[1], 1977

For certain computer systems – whether they are hardware, software, or a combination of both – it is desirable to guarantee their correct execution. The question is no longer only '*Do we build the right system (w.r.t. the client's requirements)?*', but rather '*Do we build the system right (w.r.t. the expected system behavior)?*'. This is especially important for *safety-critical systems*, i.e., those systems whose failure could result in loss of life, significant property damage, or damage to the environment. Typical examples of safety-critical systems are medical devices, aircraft flight control systems, and nuclear systems. But correct execution is also important for the large number of *commercially critical systems*, i.e., those systems whose failure would lead to enormous financial loss, e.g., in the domain of chip mass production.

Different approaches are used to analyze the correctness of a system. Frequently applied techniques are *testing* and *simulation*. These techniques can become very time consuming – especially in late phases of development – and do not consider all possible executions. Testing is performed with *test cases* that are identified in early phases of development and represent either typical or critical inputs. However, successful testing does only mean that the investigated test cases run correctly. Simulation only regards a limited number of executions and can thus only confirm that the investigated executions do not lead to erroneous situations.

Instead, formal verification techniques can investigate the complete state space of a system or, more precisely, a model of the system. Formal verification techniques comprise three different parts [HR00]:

- a *framework for modeling systems*, i.e., most typically a description language,
- a *specification language* for formulating the properties that should be fulfilled,
- a *verification method*, i.e., a formalism that defines when the description of a system satisfies the property specification.

[1]http://www-cs-faculty.stanford.edu/ knuth/faq.html

The main formal verification approaches in this context are theorem proving, equivalence checking, constraint solving, and model checking. They can be classified based on different criteria, like proof- or model-based verification, full- or property-based verification, the degree of automation, the domain of application, and the stage of usage within the development process [HR00].

It is almost impossible to list all individual existing formal specification and verification methods and corresponding tools. We will therefore focus on descriptions of those formal specification and verification techniques that are necessary to follow the remainder of this thesis.

We start with a brief introduction to *automata-based modeling approaches* which build an important foundation for the description techniques used in formal methods. Section 3.2 then focuses on *formal specification* techniques that are used to specify properties a system should fulfill. Section 3.3 outlines the formal verification technique called model checking. Model checking in combination with a notion of time is of particular interest for this thesis. Therefore, Section 3.4 gives an introduction to *real-time model checking* which particularly addresses formal verification of time-dependent properties w.r.t. a given model.

3.1 Automata-Based Modeling Approaches

In formal methods, it is necessary to describe an abstract version of the system by means of a *model* in a certain notation, i.e., a *modeling language*. These languages base upon different approaches. *Algebraic modeling languages* are used to model the behavior of concurrent communicating processes, e.g., CCS (Calculus of Communicating Systems) [Mil80] and CSP [Hoa78]. *Object-oriented modeling languages*, such as the graphical UML notation, structure the system under consideration into different views, e.g., static and dynamic parts. The third prominent approach in this context concerns *automata-based languages*.

The notion of a *finite automata* in language theory is also known as *finite state machine* (FSM), *Kripke Structure*, or *transition system* in other areas. Basically, FSMs have a finite set of states, a set of transitions between states, and labeling functions to define the output reaction on given inputs.

Let I and O be the input and the output alphabets of the FSM. Generally, a (deterministic) finite state machine is a tuple $\mathcal{M} = \langle I, O, S, S_0, next, out \rangle$ that has a finite set S of states, a set S_0 of initial states[2], a transition function $next : S \times I \rightarrow S$, and an output function out that is either defined by $out : S \rightarrow O$ or $out : S \times I \rightarrow O$.

In literature, it is commonly distinguished between Moore-type (or: state-based) and Mealy-type (or: input-based) automata. The difference lies in the definition of the output function out. For Moore-type, we have $out : S \rightarrow O$, i.e., an output symbol ω is assigned to each state $s \in S$. The symbol ω is outputted when the FSM is in state s. For Mealy-type, we have $out : S \times I \rightarrow O$, i.e., the output ω is depending on the current state $s \in S$ and an input symbol $\iota \in I$. The symbol ω is outputted when the FSM is in state s and the input symbol ι occurs.

Several variants and extensions of this basic notion of an FSM exist. For instance, it might be allowed to apply *sets of input and output symbols* for function out or introduce non-determinism by allowing a *transition relation* $next \subseteq S \times I \times S$.

[2]It is often required that $card(S_0) = 1$.

One important variant is the notion of a *Kripke Structure*. Kripke Structures are graphs that comprise of nodes representing the set of reachable states of a system and of edges representing the state transitions of the system. In the general Kripke Structure, it is often abstracted from the inputs. The transition relation is required to be *total* w.r.t. the first component, i.e., for each $s \in S$ there is at least one outgoing transition. As Kripke Structures are applied in the context of formal verification by model checking, we here define the general notion of a Kripke Structure in more detail by $\mathcal{K} = \langle Pr, S, S_0, T, l \rangle$, where Pr is a set of atomic propositions (comparable to the outputs of FSMs), S is the set of states, S_0 is the set of initial states, $T \subseteq S \times S$ is the (total) transition relation, and $l : S \rightarrow \mathcal{P}(Pr)$ is a state labeling function that shows which atomic propositions are valid in a given state s.

As modeling more complex systems with only one automata soon becomes quite cumbersome, modularity is often supported in automata-based modeling approaches. System components are modeled as separate automata that are equipped with some mechanisms to cooperate. This can be done, for example, by synchronous signals communicated between automata.

Semantics. The execution semantics of an FSM is defined by *execution paths*. An execution path is a (finite or infinite) sequence of states $\langle x_0, \ldots, x_i, \ldots \rangle$, where $\forall i \in \mathbb{N}_0 \; \exists \iota \in I : next(x_i, \iota) = x_{i+1}$. (In the case that $next$ is a relation, we require $\langle x_i, \iota, x_{i+1} \rangle \in next$.)

Starting in an initial state $s \in S_0$, the possible execution paths can be represented as an (infinite) tree of states. A state $s' \in S$ is called *reachable* iff it appears on one of the possible execution paths (or in the tree of states, respectively).

FSMs in other domains. Different other classes of automata and FSMs have emerged for particular application domains, e.g.,

- ω-automata that accept inputs of infinite length,
- communicating concurrent FSMs,
- hierarchical concurrent FSMs, in particular Harel Statecharts that are a graphical approach to represent complex control systems in a more compact form [Har87],
- dataflow graphs that are more suitable to describe computational intensive systems,
- timed automata that have an inherent notion of time,
- hybrid automata that comprise discrete and continuous time,
- and different kinds of combinations of the above, e.g., timed and hybrid Statecharts or FSMs with datapaths [Gaj97].

As an example for an extended automata model, Section 3.6.1.1 gives a complete formal definition of the syntax and semantics of *I/O-Interval Structures* that are equipped with synchronous signals to exchange information and a notion of discrete time.

3.2 Formal Specification

The notion of *formal specification* is used with various meanings in literature. The IEEE standard defines formal specification to be a specification written in a formal language, which in turn is based upon a rigorous mathematical model or just on a standardized programming or specification language [IEE87]. Van Lamsweerde provides in his roadmap of formal specification the following general definition [Lam00].

> [...] *a formal specification is the expression, in some formal language and at some level of abstraction, of a collection of properties some system should satisfy.*

Though this definition is kept very general, it also captures the *intention* of a specification and – in turn – a specification language; namely, a specification explicitly expresses the requirements (or: properties) a system under consideration should fulfill, and this has to be given in a formal, thus mathematical and unambiguous way. Depending on its intended application, a formal specification can be *executable* on a machine. But frequently, formal specifications are just *mentally executable* and used as a precise, unambiguous basis for discussion among members of a developer team. Formal specifications have been classified in literature with respect to different criteria, e.g., whether they contain graphical elements in their syntactical domain, whether they are executable, or whether they are tool supported.

Different kinds of logics build the foundation of formal specification languages, ranging from propositional and predicate logics over modal logics and temporal logics to process algebras. An overview of specification languages based on this classification is presented in [Mül96].

The most popular predicate logic-based languages are VDM [ISO96], Z [ISO02], B, and Larch. Prominent approaches for process algebraic specification are CCS (Calculus of Communicating Systems) [Mil80] and CSP (Communicating Sequential Processes) [Hoa78]. But we here focus on specification languages that are based on *temporal logics*, as these are of relevance for the remainder of this thesis.

3.2.1 Temporal Logics

Temporal logics are frequently applied to specify properties of state-transition systems or – more specifically – Kripke structures (see Section 3.1). They are used to describe required or illegal execution sequences.

In contrast to propositional and predicate logic, the validity of a temporal logic formula cannot be statically determined based on a single snapshot of a Kripke structure, i.e., an overall description of the current status of the model. Rather, a temporal logic formula concerns several (even infinitely many) snapshots. Temporal logics are therefore defined over *configuration sequences* or *runs* that represent possible executions of a model. In the following, we write $\pi = \langle x_0, x_1, \ldots \rangle$ for a path in a given Kripke structure $\mathcal{K} = \langle Pr, S, S_0, T, L \rangle$. We denote π^i for the suffix of π starting a position i of the path, i.e., $\pi^i = \langle x_i, x_{i+1}, \ldots \rangle$.

There is a variety of temporal logics described in literature that have been used in different domains. We can classify these by their view of evolving time. *Linear time*-based logics have a notion of execution sequences as a whole, while *branching time*-based logics work with alternative execution possibilities at each given point of time. This is especially useful when reasoning about non-deterministic models. We can also distinguish between temporal logics that work for *discrete* or for *continuous* time. While continuous time is frequently applied to reason about analogue systems, discrete time is often chosen to specify properties of concurrent synchronous models. A third characteristic that can be employed to classify temporal logics are whether they are future- or past-oriented.

We here describe two frequently applied discrete, future-oriented temporal logics called *Linear Temporal Logic* (LTL) and *Computation Tree Logic* (CTL), but note that there are other temporal logics available, e.g., the temporal logic of actions (TLA) by Lamport [Lam94] and the more general CTL* [CE81, CES86] that has LTL and CTL as sublanguages.

3.2.1.1 Linear Temporal Logic (LTL)

Formulas expressed in Linear Temporal Logic [Pnu80] are defined on individual computation paths π. The syntax of LTL formulas is defined by the following grammar in Backus Naur form.

Definition 3.1 *Let Pr be a set of atomic propositions, and let $p \in Pr$. The syntax of a linear temporal logic formula ϕ is recursively defined by*

$$\phi ::= \ p \ | \ \mathrm{true} \ | \ \mathrm{false} \ | \ \neg\phi \ | \ \phi \wedge \phi \ | \ \phi \vee \phi \ | \ \mathrm{X}\,\phi \ | \ \mathrm{F}\,\phi \ | \ \mathrm{G}\,\phi \ | \ \phi\,\mathrm{U}\,\phi\,.$$

The literals X, F, G, and U are *temporal operators* that describe properties of a path π.

- X is the *next* operator and requires that the following formula is true for the second – or more generally – next state of the path,
- F (often also denoted by <>) is the the *eventually* or *sometime in the future* operator and requires that the following formula is true in some subsequent state of the path,
- G (often also denoted by []) is the *globally* operator and requires that the following formula is true for all subsequent states of the path,
- U is the *until* operator and combines two formulas ϕ_1 and ϕ_2. The LTL formula ϕ_1 U ϕ_2 requires that – if there is a state on the path where ϕ_2 is true, on all preceding states ϕ_1 has to be true.

Sometimes additional useful operators are defined, e.g., the *release* operator R that is dual to the until operator [CGP99]. The formula ϕ_1 R ϕ_2 requires that ϕ_2 holds along the path up to an including the first state where ϕ_1 holds, but note that ϕ_1 does not have to hold at all. Logically, ϕ_1 R ϕ_2 is equivalent to $\neg(\neg\phi_1 \mathrm{~U~} \neg\phi_2)$.

An LTL formula is evaluated on an execution path or a set of execution paths. The regarded paths satisfy a formula ϕ if each of the regarded paths satisfies ϕ. Note that in [CGP99], LTL formulas are always considered for the set of *all* execution paths. We define a satisfaction relation for LTL formulas as follows.

Definition 3.2 *Let ϕ, ϕ_1, ϕ_2 be LTL formulas. We write $\mathcal{K}, \pi \models \phi$ to denote that the execution path π of the Kripke structure $\mathcal{K}$ satisfies the LTL formula ϕ. The satisfaction relation $\models$ for the semantics of LTL formulas over a path π in a Kripke structure $\mathcal{K}$ is inductively defined as follows.*

1. $\mathcal{K}, \pi \models \text{true}$
2. $\mathcal{K}, \pi \models p$ iff $p \in l(x_0)$, where $p \in Pr$ and x_0 is the first state of π
3. $\mathcal{K}, \pi \models \neg\phi$ iff not $\mathcal{K}, \pi \models \phi$
4. $\mathcal{K}, \pi \models \phi_1 \wedge \phi_2$ iff $\mathcal{K}, \pi \models \phi_1$ and $\mathcal{K}, \pi \models \phi_2$
5. $\mathcal{K}, \pi \models \phi_1 \vee \phi_2$ iff $\mathcal{K}, \pi \models \phi_1$ or $\mathcal{K}, \pi \models \phi_2$
6. $\mathcal{K}, \pi \models \mathrm{X}\, \phi$ iff $\mathcal{K}, \pi^1 \models \phi$
7. $\mathcal{K}, \pi \models \mathrm{F}\, \phi$ iff there exists an $i \geq 0$, such that $\mathcal{K}, \pi^i \models \phi$
8. $\mathcal{K}, \pi \models \mathrm{G}\, \phi$ iff for all $i \geq 0 : \mathcal{K}, \pi^i \models \phi$
9. $\mathcal{K}, \pi \models \phi_1 \,\mathrm{U}\, \phi_2$ iff there is an $i \geq 0$, such that $\mathcal{K}, \pi^i \models \phi_2$ and for all $j \in \{0, \dots, i-1\}$ holds $\mathcal{K}, \pi^j \models \phi_1$

It is easy to determine that operators $\vee$, $\neg$, X, and U are sufficient to express any LTL formula, as it holds

$$\begin{aligned} \phi_1 \wedge \phi_2 &\equiv \neg(\neg\phi_1 \vee \neg\phi_2), \\ \mathrm{F}\, \phi &\equiv \text{true}\, \mathrm{U}\, \phi, \\ \mathrm{G}\, \phi &\equiv \neg(\text{true}\, \mathrm{U}\, \neg\phi). \end{aligned}$$

3.2.1.2 Computation Tree Logic (CTL)

Computation Tree Logic is due to Clarke and Emerson [EC80, CE81]. It uses the notion of future branching executions, i.e., the execution of a model is regarded as a tree-like structure, starting with the current status of the model as the root.

CTL and LTL are closely related, as they basically work with the same temporal operators. However, in CTL there are additional *path quantifiers* attached to each temporal operator. Path quantifiers are denoted by E and A. The existential path quantifier E is used to require that the following subformula must hold for at least one of the possible future paths. The other path quantifier A is used to require that the following subformula must hold for all possible future execution paths. The syntax of CTL is recursively defined by the following grammar in Backus Naur form.

Definition 3.3 *Let Pr be a set of atomic propositions, and let $p \in Pr$. The syntax of a computation tree logic formula ϕ is recursively defined by*

$$\begin{aligned} \phi ::= {} & p \mid \text{true} \mid \text{false} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \\ & \mid \mathrm{EX}\, \phi \mid \mathrm{EF}\, \phi \mid \mathrm{EG}\, \phi \mid \mathrm{E}(\phi\, \mathrm{U}\, \phi) \\ & \mid \mathrm{AX}\, \phi \mid \mathrm{AF}\, \phi \mid \mathrm{AG}\, \phi \mid \mathrm{A}(\phi\, \mathrm{U}\, \phi)\ . \end{aligned}$$

Again, several other useful operators are defined for CTL, e.g., logical implication ($\phi_1 \rightarrow \phi_2$), a *weak until* operator W that does not require the second subformula to ever become true, or a *before* operator B that is dual to the until operator.

In contrast to LTL, the semantics definition in terms of a satisfaction relation is based on a single state $x \in S$ of the model and not on a run π. We therefore need a slightly different notation for execution paths π and denote now a path starting in state $x \in S$ by $\pi := \langle x_0, x_1, \ldots, x_i, \ldots \rangle$. Implicitly, this requires that there is a transition between each subsequent pair of states in π, i.e., $\forall i \in \mathbb{N}_0 : \langle x_i, x_{i+1} \rangle \in T$.

Definition 3.4 *Let ϕ, ϕ_1, ϕ_2 be CTL formulas. We write $\mathcal{K}, x_0 \models \phi$ to denote that the CTL formula ϕ is valid for state $x_0 \in S$ of the Kripke structure $\mathcal{K}$. The satisfaction relation $\models$ for the semantics of CTL formulas over a state x_0 in of the Kripke structure $\mathcal{K}$ is inductively defined as follows.*

1.	$K, x_0 \models \text{true}$	
2.	$K, x_0 \models p$	iff $p \in l(x_0)$, where $p \in Pr$
3.	$K, x_0 \models \neg\phi$	iff not $K, x_0 \models \phi$
4.	$K, x_0 \models \phi_1 \wedge \phi_2$	iff $K, x_0 \models \phi_1$ and $K, x_0 \models \phi_2$
5.	$K, x_0 \models \phi_1 \vee \phi_2$	iff $K, x_0 \models \phi_1$ or $K, x_0 \models \phi_2$
6.	$K, x_0 \models \mathrm{EX}\, \phi$	iff there exists a path $\langle x_0, x_1, \ldots \rangle$, such that $K, x_1 \models \phi$
7.	$K, x_0 \models \mathrm{EF}\, \phi$	iff there exists a path $\langle x_0, x_1, \ldots, x_i, \ldots \rangle$, such that there is an $i \in \mathbb{N}$ with $K, x_i \models \phi$
8.	$K, x_0 \models \mathrm{EG}\, \phi$	iff there exists a path $\langle x_0, x_1, \ldots \rangle$, such that for all $i \in \mathbb{N}_0 : K, x_i \models \phi$
9.	$K, x_0 \models \mathrm{E}(\phi_1 \,\mathrm{U}\, \phi_2)$	iff there exists a path $\langle x_0, x_1, \ldots, x_i, \ldots \rangle$, such that there is an $i \in \mathbb{N}_0$ with $K, x_i \models \phi_2$ and for all $j \in \mathbb{N}_0$ with $0 \leq j < i : K, x_j \models \phi_1$
6.	$K, x_0 \models \mathrm{AX}\, \phi$	iff for all paths $\langle x_0, x_1, \ldots \rangle$ holds that $K, x_1 \models \phi$
7.	$K, x_0 \models \mathrm{AF}\, \phi$	iff for all paths $\langle x_0, x_1, \ldots, x_i, \ldots \rangle$ holds that there is an $i \in \mathbb{N}$ with $K, x_i \models \phi$
8.	$K, x_0 \models \mathrm{AG}\, \phi$	iff for all paths $\langle x_0, x_1, \ldots \rangle$ holds that for all $i \in \mathbb{N}_0 : K, x_i \models \phi$
9.	$K, x_0 \models \mathrm{A}(\phi_1 \,\mathrm{U}\, \phi_2)$	iff for all paths $\langle x_0, x_1, \ldots, x_i, \ldots \rangle$ holds that there is an $i \in \mathbb{N}_0$ with $K, x_i \models \phi_2$ and for all $j \in \mathbb{N}_0$ with $0 \leq j < i : K, x_j \models \phi_1$

Again, it is easy to determine that all CTL formulas can be expressed by only using the logical operators $\vee$ and $\neg$, the temporal operators X and U, and the path quantifiers E and A. For more details about syntax and semantics of the temporal logics CTL and LTL and illustrative examples, we refer to [MP92, MP95, CGP99, HR00] where also several references to other literature concerning temporal logics are given.

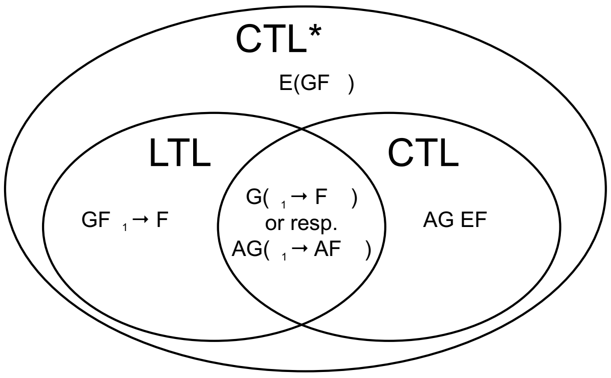


Figure 3.1: Expressive Power of LTL, CTL, and CTL*

Surprisingly, a comparison of CTL and LTL with respect to their expressiveness is difficult. It turns out that neither language is contained in the other. More specifically, the situation is as depicted in Figure 3.1. There are formulas in each of the two languages that are not expressible in the other, and CTL*, which has LTL and CTL as sublanguages, is more expressive than just the union of LTL and CTL. Corresponding sample formulas taken from [HR00] are provided in Figure 3.1.

3.2.2 Property Specification Patterns

It is interesting to see that only a limited sublanguage of complete temporal logics is relevant in practice. Certain *patterns* of specification can be identified. The first classification is due to Lamport [Lam77] and distinguishes between safety and liveness properties. Safety properties are also called invariants and are used to ensure that some erroneous situation will never be reached. Typically they start with the temporal operator AG, i.e., on all execution paths some property must globally be true. Liveness properties specify that some desired situation will eventually occur.

A more elaborated, hierarchical classification with the notion of safety and progress was suggested by Manna and Pnueli in [MP90, MP92]. They distinguish between six categories of properties, as it is illustrated in Figure 3.2.

Safety properties conform to the invariants identified by Lamport. Additionally, formulas of form $\mathrm{F}\ \phi$ in LTL (or $\mathrm{AF}\ \phi$ in CTL) build the class of *guarantee properties*. The disjunctive combination of formulas of these two base classes builds the class of *obligation properties*. Two more classes called *response* (or *recurrence*) and *persistence* are identified. Finally, the disjunctive combination of response and persistence properties builds the class of *reactive properties*.

Figure 3.2 is slightly modified compared to the original, as it only shows the *basic implications* between property classes. Here, an arrow connecting a property class with another class

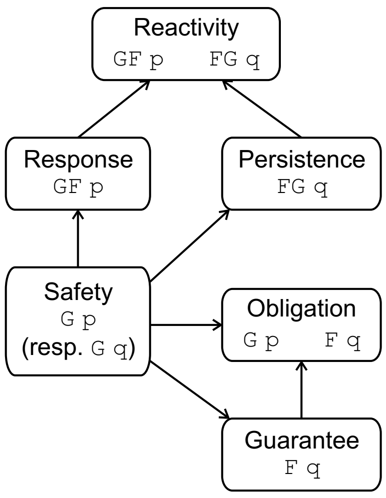


Figure 3.2: Property Classification based on Manna and Pnueli [MP92]

indicates an implication, e.g., a valid safety property $\mathrm{G}\ p$ implies that the response property $\mathrm{GF}\ p$ holds. Of course, additional implications can be derived, e.g., a valid obligation property $\mathrm{G}\ p \ \vee\ \mathrm{F}\ q$ implies that the response property $\mathrm{GF}\ p$ holds.

However, this classification is based upon the *syntax* (or *structure*) of the formulas. It might be more useful to have a *semantical* classification that better reflects the way of thinking when a requirement is to be specified. Such a classification might be more intuitive for non-experts to find the right formula. We will discuss such an approach in the next paragraphs.

In the recent two decades, a lot of experience in the domain of formal specification with temporal logics has been made. This is also due to the progress made in the area of formal verification by model checking (see Section 3.3). It turned out that the full power of temporal logics, which allow for arbitrarily nested formulae, is not needed in practice to formulate required properties. In this context, Dwyer et al. have developed a pattern system based upon more than 500 property specifications from different projects in the area of finite-state verification [DAC98b, DAC98a, DAC99]. That pattern system provides a structured set of commonly occurring property specifications and examples of how to translate these into different formal specification languages, such as LTL, CTL, or Graphical Interval Logic (GIL) [DKM+94].

The overall aim of the pattern approach is to support developers in a way that abstracts from the formal syntax of temporal logics.

3.2.2.1 Scopes

Dwyer et al. have identified different *scopes* applicable to a pattern. A scope is the part of the system execution path over which a pattern has to hold. Five basic kinds of scopes have been identified, as illustrated in Figure 3.3:

- Globally (i.e., the entire execution path),
- Before `R` (i.e., execution up to a state `R`),
- After `Q` (i.e., execution after a state `Q`),

- Between `Q` and `R` (i.e., all parts of the execution path from state `Q` to another state `R`), and
- After `Q` until `R` (i.e., all parts of the execution path from state `Q` to another state `R`, including those parts where `R` never occurs).

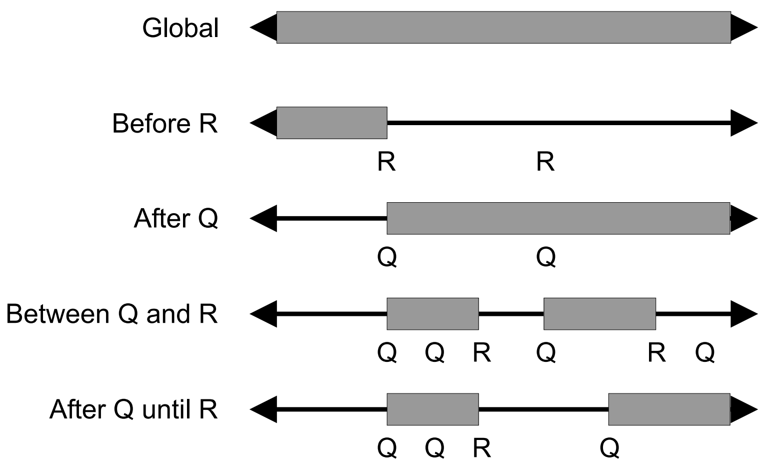


Figure 3.3: Specification Scopes of [DAC98a]

For state-delimited scopes with distinct delimiters `Q` and `R`, the interval in which the property is evaluated is closed at the left and open at the right end. Thus, the scope consists of all states beginning with the starting state and up to – but not including – the ending state. It is possible, however, to define scopes that are open-left and closed-right as well.

Note that most scopes may appear repetitively or with an unlimited future, as illustrated in Figure 3.3. These scopes are therefore embedded in *invariants*. Scope '`Before R`' is not an invariant, as we only investigate executions *up to the first occurrence* of `R` in this case. Patterns with that scope are only applied to paths starting at the initial state.

3.2.2.2 Patterns

The patterns themselves are hierarchically ordered as shown in Figure 3.4. In an online repository, for each pattern, each scope, and each formalism a corresponding formal description is provided [DAC98a]. To illustrate the approach, we take the absence pattern as an example. Table 3.1 shows corresponding CTL formulae for each scope in the context of the absence pattern.

The absence pattern describes a part of an execution path that is free of a certain state `P`. It is often also referred to as '*Never*'. We take a closer look at the pattern '`P is false before R`' in Table 3.1. A first intuitive attempt to specify a CTL formula for that case would be `A(!P W R)`. This formula makes use of the *weak until* operator `W` and means that along all possible execution paths `P` is not entered from the initial state until the first state in which `R` is true, *if any*. In particular, if `R` is never entered along an execution path, then `P` must also not be entered along that path.

Dwyer et al. always consider the case that scope delimiters `Q` and `R` might not appear on execution paths. Now consider the case that on every execution path `P` becomes eventually true, but `R` will afterwards never be entered. In this case, the property '`P is false before R`'

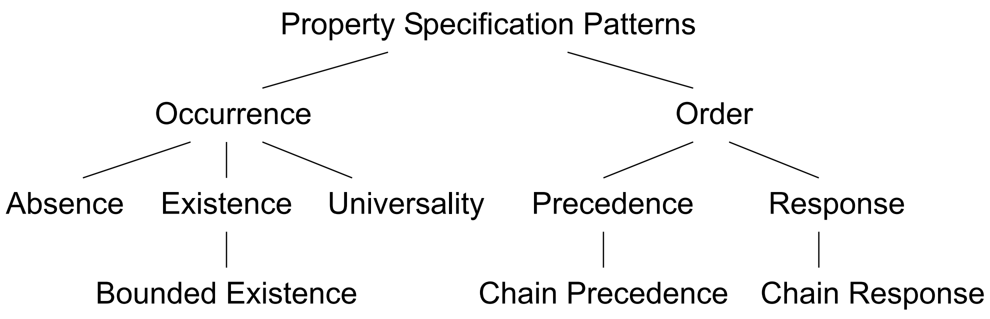


Figure 3.4: Property Specification Patterns

Table 3.1: CTL Formulae for Absence Pattern [DAC98a]

P is false …	
…globally	`AG(!P)`
…before `R`	`A((!P	AG(!R)) W R)`
…after `Q`	`AG(Q -> AG(!P))`
…between `Q` and `R`	`AG((Q & !R) -> A((!P |AG(!R)) W R))`
…after `Q` until `R`	`AG((Q & !R) -> A(!P W R))`

should also be true. However, our first formula `A(!P W R)` developed above does not cover this case and results in false in this case. One solution to resolve this issue is to add the sub-formula `(P & AG(!R))` as an alternative to sub-formula `!P`, resulting in

```
A( (!P | (P & AG(!R))) W R) .
```

As it holds $\neg a \vee (a \wedge b) \equiv \neg a \vee b$, we can simplify the latter formula to its final version as it appears in Table 3.1:

```
A((!P | AG(!R)) W R) .
```

As demonstrated, it always takes additional effort to include the case that scope delimiters `Q` and `R` might not appear at all on execution paths. Such assumptions unnecessarily complicate the resulting formulae. Instead, we propose a slightly different approach with inherent assumptions *requiring* that scope delimiters will eventually appear on all paths. Only if an assumption of such kind holds, a pattern can then be applied. Otherwise, a statement about validity cannot be given. By this approach, it is guaranteed that all possible executions really comply to the intended scope. Users of the pattern system need therefore pay less attention to whether delimiter states `Q` and `R` occur or not.

Moreover, mappings to respective temporal logic formulae, e.g., CTL, are significantly simplified and easier to adapt for further usage. As an example, Table 3.2 on the next page shows the absence pattern with additional assumptions and a simplified mapping to CTL formulae.

Assumptions can also be easily mapped to CTL and have to be checked separately. When an assumption is false over a given model, the actual property that is investigated cannot be validated.

Table 3.2: CTL Formulae for Absence Pattern with Additional Assumptions

Assumption	**Pattern**	**CTL Formula**
	`P` is globally false	`AG(!P)`
`R` becomes true on all paths [CTL: `AF(R)`]	`P` is false before `R`	`A(!P U R)`
`Q` becomes true on all paths [CTL: `AF(Q)`]	`P` is false after `Q`	`AG(Q -> AG(!P))`
`Q` and `R` always again become true on all paths [CTL: `AG AF(Q) & AG AF(R)`]	`P` is false between `Q` and `R`	`AG((Q & !R) -> A(!P U R))`
`Q` always again becomes true on all paths [CTL: `AG AF(Q)`]	`P` is false after `Q` until `R`	`AG((Q & !R) -> A(!P W R))`

While the patterns provided in the pattern system by Dwyer et al. already cover a broad range of requirements, it might still be necessary to adjust them for particular and more complex properties. There are a number of ways how this can be performed, e.g., by parameterization, logical combination, and changes in pattern scopes [DAC98a]. But note that users of the pattern system are usually not able to modify the temporal logic formulae without a concise understanding of the underlying semantics of the formal logics.

3.3 Symbolic Model Checking

In the terms of the classification for verification techniques, model checking is an automatic, model-based, property-verification formal method. Model Checking is intended to prove the correctness of safety-critical, concurrent, reactive systems. A *reactive system* is an event driven or control driven system that continuously has to react to external and/or internal stimuli. An additional specification by means of (temporal) properties represents the desired behavior of the system. From a logical viewpoint, the system is given as a Kripke Structure, and the properties are given by temporal logic formulae.

The general model checking approach is defined as follows.

Given a system that is modeled as a Kripke Structure $K = \langle Pr, S, S_0, R, L\rangle$ with a set Pr of atomic predicates, a set S of states, a total state transition relation $T \subseteq S \times S$, and a state labeling function $L : S \to \mathcal{P}(Pr)$. Let f be a temporal logic formula that specifies a desired behavioral property of K.

Table 3.3: Example Symbolic Representation

Atomic Proposition	**Symbolic Representation**
p	$\neg x_1 \wedge \neg x_2$
q	$\neg x_1 \wedge x_2 \wedge \neg x_3$
r	$x_1 \wedge \neg x_2 \wedge x_3$
s	$x_1 \wedge \neg x_2 \wedge \neg x_3$

The task is to find the set $C \subseteq S$ of states, for which f holds, i.e., $C \overset{def}{=} \{s \in S \mid K, s \perp f\}$.

Model checking Kripke Structures has limits due to the state space explosion problem, which results from the fact that the state space that has to be explored grows exponentially due to the cross product of concurrent modules or components in the model. A major enhancement therefore was to apply *binary decision diagrams* (BDDs) and a *symbolic representation* of the Kripke Structure under investigation [BCM$^+$90]. Symbolic model checking codes a Kripke Structure by using a vector of binary variables.

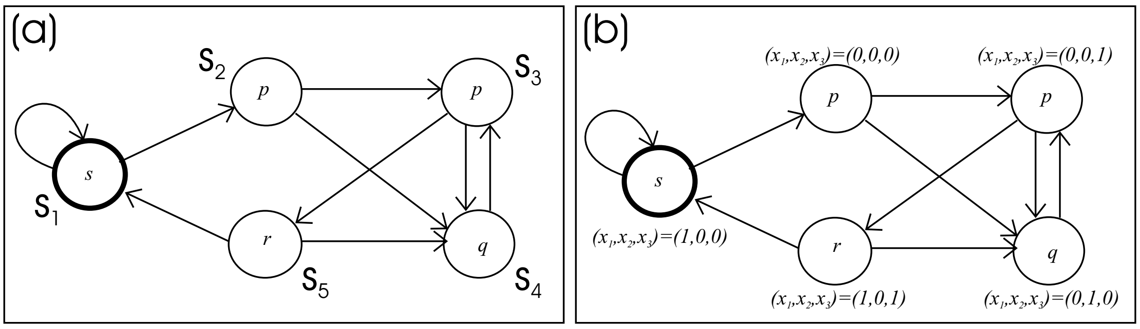


Figure 3.5: Sample Binary Coding of States

Example. Figure 3.5 provides an example. Given a Kripke Structure $\mathcal{K} = \langle Pr, S, S_0, T, L\rangle$ with $Pr = \{p, q, r, s\}$, $S = \{s_1, \ldots, s_5\}$, $S_0 = \{s_1\}$, and transitions and state labels as given in Figure 3.5(a). Symbolic model checking now transforms this model into another representation, i.e., the states of the Kripke Structure are coded by binary variables. In the example, the five states are represented by variables $x_1, x_2,$ and x_3 (Figure 3.5(b)). Atomic propositions and formula parts can then be described in terms of this coding. E.g., the set of states that hold s in the example is represented by $x_1 \wedge \neg x_2 \wedge \neg x_3$. More examples are listed in Table 3.3.

A corresponding (reduced ordered) binary decision diagram (ROBDD) for the representation of s is shown in Figure 3.6. Note that the order of variables influences the size of a BDD. Unfortunately, finding an optimal order is NP-complete. However, several approaches apply heuristics to find 'reasonable' orders of variables for BDDs.

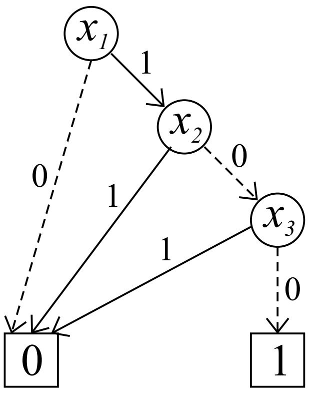


Figure 3.6: Binary Decision Diagram for $x_1 \wedge \neg x_2 \wedge \neg x_3$

BDDs can be used to code Kripke Structures, such that model checking algorithms can use this more compact structures for verification purposes. Basically, the transition relation is coded by the symbolic state representation. E.g., the transition from S_1 to S_2 in Figure 3.5 is coded by $x_1 \neg x_2 \neg x_3 \neg x'_1 \neg x'_2 \neg x'_3$. I.e., each BDD variable x is duplicated by x' to distinguish between transition source and target states. The complete Kripke Structure is then represented by disjunction of all transition formulas:

$$\begin{array}{llll} & x_1 \neg x_2 \neg x_3 x'_1 \neg x'_2 \neg x'_3 & \vee\, x_1 \neg x_2 \neg x_3 \neg x'_1 \neg x'_2 \neg x'_3 & \vee\, \neg x_1 \neg x_2 \neg x_3 \neg x'_1 \neg x'_2 x'_3 \\ \vee & \neg x_1 \neg x_2 \neg x_3 \neg x'_1 x'_2 \neg x'_3 & \vee\, \neg x_1 \neg x_2 x_3 \neg x'_1 x'_2 \neg x'_3 & \vee\, \neg x_1 \neg x_2 x_3 x'_1 x'_2 \neg x'_3 \\ \vee & \neg x_1 x_2 \neg x_3 \neg x'_1 \neg x'_2 x'_3 & \vee\, x_1 \neg x_2 x_3 \neg x'_1 x'_2 \neg x'_3 & \vee\, x_1 \neg x_2 x_3 x'_1 \neg x'_2 \neg x'_3 \end{array}$$

Model checking of higher level software models has already been successfully applied [CAB$^+$98]. However, the results are still limited due to the mentioned *state explosion problem*. This problem has led to a number of modular [GKC99] and compositional [MC81] approaches in model checking.

The two most popular model checking tools are SMV[3] and SPIN[4]. SMV accepts modular transition systems and CTL formulae as an input. There is a distinction between synchronous and interleaving execution of modules, i.e., in the interleaving mode, one execution step refers to one execution step of a single module only, while in the synchronous mode, all modules synchronously perform an execution step. SMV has a notion of *fairness*. Fairness properties are necessary to restrict the set of possible execution paths for verification of required behavioral properties. *Strong fairness* ensures that a module (or transition) that is infinitely often activated will also infinitely often be executed (or: fired). *Weak fairness* garantuees that modules progress independently from another in the interleaving mode.

SPIN uses a modeling language called PROMELA (PROcess MEta LAnguage) and supports verification of LTL formulae. As a remarkable feature, on-the-fly model checking techniques are employed that allow for verification of basic properties like safety and liveness properties. On-the-fly model checking has the advantage that there is no need to build a global state graph for the verification certain system properties.

[3]http://www-2.cs.cmu.edu/ modelcheck/smv.html
[4]http://spinroot.com/spin/whatispin.html

3.4 Real-Time Model Checking

Most of the currently existing works on model checking do not consider time-dependent behavior. But to be able to verify models with an inherent notion of time, some model checking approaches have been developed to also verify corresponding state transition systems.

Generally, adding a notion of time to state transition systems exacerbates the state explosion problem, especially if multiple timed transition systems are to be combined and (non-deterministic) timing intervals for transition times are allowed. Therefore, enhanced symbolic representations by extensions BDDs have been applied for timed transition systems which are more suitable for dedicated efficient verification algorithms.

On the one hand side, well-known CTL model checking techniques have been extended to cope with timing aspects. These approaches usually allow for labeling transitions with delay times (by means of natural numbers). Most notably, the underlying models assume a global clock with discrete time. For property specification, these approaches apply quantized timing parameters that are attached to temporal operators. Corresponding CTL extensions are, e.g., RTCTL (real time CTL, [EMSS92]), QCTL (quantized CTL, [FGK96]), and CCTL (clocked CTL [RK97]). Tools following this approach are Verus [CCM97] and RAVEN (**R**eal-time **A**nalyzing and **V**erification **EN**vironment) [Ruf01].

On the other side, a more general approach is based on *timed automata* by Alur et al. [ACD90]. In timed automata, time is represented by (an arbitrary number of) clocks carrying real numbers. The clocks are incremented synchronously. Each state is associated with an invariant that is a requirement on the values of the clocks. However, comparisons of clock values are only possible with constants. A transition is chosen based on input events and clock predicates. When firing a transition, the corresponding clocks are reset to zero.

In timed automata, time is passing in states, while firing a transition does not take time. As the clocks are real numbers, this results in an infinite state space. However, comparisons in invariants can only be made over integer values, such that two clocks with actual different (real) values might not be distinguished.

Properties over timed automata are expressed by Timed CTL (TCTL), an extension of CTL with dense-time semantics. It has been shown that model checking timed automata over TCTL is PSPACE complete [ACD90]. However, tools make use of enhanced techniques to build efficient data structures such that the symbolic representation of the model still becomes manageable.

Tools that base upon timed automata are Kronos[5] and UPPAAL[6]. Note that UPPAAL only supports limited subset of temporal logics for reachability analysis.

Another kind of model checking approach is considered by the HyTech tool. The underlying model are *hybrid systems* that comprise of continuous as well as discrete parts, i.e., a common finite state machine is equipped with continuous variables. States are associated with conditions over these variables and their first derivative (i.e., time). Transitions of a hybrid automata have a triggering condition, a number of assignment to variables, and might carry an annotation to synchronize with other automata. The HyTech model checker works for *linear* hybrid automata, i.e., all derivatives of variables are constants. Note that a hybrid automaton, in

[5]http://www-verimag.imag.fr/TEMPORISE/kronos

[6]http://www.uppaal.com

which all derivatives are 1 and all variables of the transitions are set to zero, is basically a timed automaton.

HyTech only supports a subset of TCTL, similar to UPPAAL. For hybrid automata, it has been shown that for a given set of states one can determine the set of next states. But it cannot be proved whether a given state will ever be reached. Thus, formal verification becomes semi-decidable. But for model checking it is of more practical relevance that the memory complexity and numeric precision can be handled.

3.5 Selection of a Real-Time Model Checking Tool

In this subsection, we review the four mentioned real-time model checkers Kronos, Verus, UPPAAL, and RAVEN in more detail. Basically, we can distinguish them by the following criteria:

- Underlying timing model (discrete or continuous),
- the number of clocks per module/component,
- the employed temporal logics (full TL or only partly supported),
- availability of a graphical user interface, and
- counter example representation.

Table 3.4 gives an overview about the model checkers. However, the performance of the model checkers cannot be directly compared in a fair way, as each of these tools has its advantages in a certain application domain. Only some comparisons are documented in literature, e.g., [Wit99, Ruf01].

We identify the following list of requirements that have to be met by a model checker to be suitable in the context of this thesis.

- The approach of this thesis aims to support verification of time-related properties in early stages of development and thus on a rather high level of abstraction. We therefore do not need to cope with subtle timing issues among system components (e.g., clock drifts). We can thus assume a *global notion of time* such that the system components synchronously perform execution steps at each tick of the global clock.
- We require synchronous as well as asynchronous communication among system components, and assume that messages are never lost.
- We also assume that we know of a minimal time unit for executions of actions and state transitions, which leads to a *discretization* of time.
- We need the ability to count the time elapsed since a state was entered. We therefore do not need the full power of timed automata, as we do not need multiple clocks in a systems component.

Table 3.4: Overview of Real-Time Model Checkers

Criteria	**UPPAAL**	**Kronos**	**Verus**	**RAVEN**	**HyTech**
Automata Model	Timed Automata	Timed Automata	Common Kripke Structures	Time-annotated Kripke Structures	Hybrid Automata
Time Model	Continuous	Continuous	Discrete unit-delay	Discrete multiple delay	Continuous
Number of Clocks	≥ 1	≥ 1	1 per module	1 per module	≥ 1
Internal State Representation	Explicit	Explicit	BDDs	Multi Terminal BDDs	Explicit
Temporal Logics	Parts of TCTL	TCTL	TCTL	CCTL	Parts of TCTL
GUI	Graphical Modeling Environment	None	None	Text input, GUI for Model Checking	Graphical Modeling Environment
Counter Examples	Graphical in GUI	Textual	Textual	External Waveform Browser	Textual

- We want to be able to *verify general properties* and can therefore not restrict on reachability analysis.

Discrete time, no need for multiple clocks, and support for general property specifications lead to either choosing Verus or RAVEN. Note that asynchronous communication can be programmed by hand using intermediate channel modules.

As RAVEN does not only show a better performance in some case studies [Ruf01], but also provides a user interface and additional timing analysis algorithms, this model checker was chosen for performing verification in the context of this thesis.

The timing and value analysis algorithms do not only return yes/no answers to inform whether properties hold or not. They determine minimal/maximal times that pass between two specified system states as well as minimal/maximal values of variables. This can be very helpful in the analysis phase.

3.6 RAVEN

In RAVEN, a model is given by a time-annotated state transition system, i.e., a set of so-called I/O-Interval Structures [Ruf01]. I/O-Interval Structures are based on Kripke Structures with [min,max]-time intervals at their state transitions and additional input and output signals to better support communication among Interval Structures. The specification language of RAVEN is called *Clocked CTL* (CCTL). Interval Structures, CCTL, and a corresponding input language are discussed in the following subsections. The definitions in Sections 3.6.1 and 3.6.1.1 are taken from the PhD thesis of Jürgen Ruf [Ruf00]

3.6.1 Interval Structures

Interval Structures are basically state-transition systems with time-annotated transitions. Each Interval Structure has exactly one clock that keeps track of elapsed time. The clock is reset to zero when – after taking a transition – the transition destination state is entered. A state *may* be left if the current clock value corresponds to a delay time specified by (at least) one of the outgoing transitions. The state *must* be left if the maximal delay time of all outgoing transitions is reached, as illustrated in Figure 3.7.

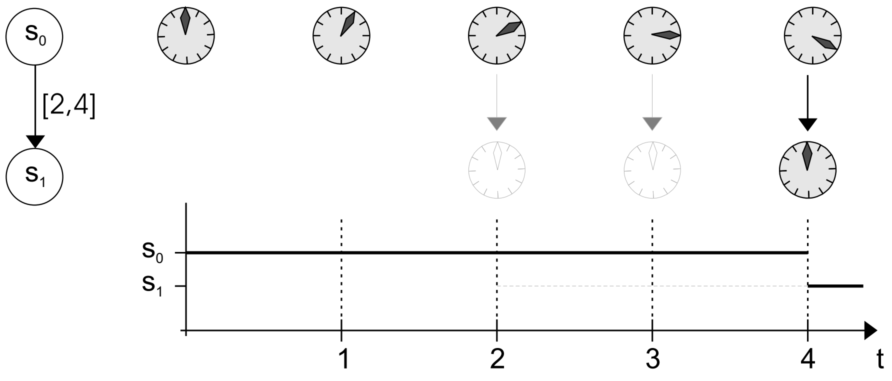


Figure 3.7: Example Interval Structure Transition [RK99]

Definition 3.5 *An Interval Structure $\Im$ is a tuple $\Im \overset{def}{=} \langle Pr, S, s_0, T, L, I \rangle$ with*

- *a set of atomic propositions Pr,*
- *a set of states S,*
- *an initial state $s_0 \in S$,*
- *a transition relation between the states $T \subseteq S \times S$, in which every state has at least one successor state, i.e., $\forall s \in S\ \exists s' \in S : \langle s, s' \rangle \in T$,*
- *a state labeling function $L : S \to \mathcal{P}(Pr)$,*
- *a transition time labeling function $I : T \to \mathcal{P}(\mathbb{N})$.*

Definition 3.6 *The maximal state time of a state $s \in S$ is the maximal delay time of all outgoing transitions of s. It is defined by*

$$MaxTime \overset{def}{=} \begin{cases} S \to \mathbb{N} \\ s \mapsto max\{\ v \mid \exists s' \in S : \langle s, s' \rangle \in T \ \wedge \ v \in I(s, s')\} \end{cases}$$

Every state s of an Interval Structure must be left after the maximal state time $MaxTime(s)$. Besides the states, we also have to consider the elapsed time since entering the current state to determine the transition behavior of the system. Thus, the actual state of an Interval Structure is given by a state together with the current clock value that represents the elapsed time. We call this tuple an *IS-configuration*.

Definition 3.7 *An IS-configuration $g \in S \times \mathbb{N}$ is a state $s \in S$ associated with a clock value $v \in \mathbb{N}_0$. The set of all IS-configurations in an Interval Structure $\Im = \langle Pr, S, s_0, T, L, I \rangle$ is given by:*

$$G \overset{def}{=} \{\langle s, v \rangle \mid s \in S \ \wedge \ v \in \mathbb{N}_0 \ \wedge \ v < MaxTime(s)\}$$

The dynamic semantics of Interval Structures is defined by *runs*, i.e., sequences of IS-configurations.

Definition 3.8 *Let $\Im$ be an Interval Structure, $\Im = \langle Pr, S, s_0, T, L, I \rangle$, and let g_0 be an initial IS-configuration. A* run *r is an (infinite) sequence of IS-configurations $(g_0, g_1, \ldots)$. For the IS-configurations $g_i = \langle s_i, v_i \rangle$ of such a sequence holds either*

- $s_i = s_{i+1} \ \wedge \ v_{i+1} = v_i + 1 \ \wedge \ v_{i+1} < MaxTime(s_i)$*, or*
- $\langle s_i, s_{i+1} \rangle \in T \ \wedge \ v_i + 1 \in I(s_i, s_{i+1}) \ \wedge \ v_{i+1} = 0.$

3.6.1.1 I/O-Interval Structures

To enable communication between a set of Interval Structures, an extension called *I/O-Interval Structures* has been proposed [RK99]. In these structures, input variables may be additionally specified. An *input label*, i.e., a Boolean formula over input variables, is attached to each transition. It is interpreted as an input condition that has to hold during the corresponding transition times. If no input label is explicitly specified for a transition, it is set to $true$ by default.

In the following definition, we formalize input labels with sets of valuations over the set Pr_{input} of input variables. An element of set $Inp \overset{def}{=} \mathcal{P}(Pr_{input})$ defines exactly one valuation of the input variables: the propositions contained in the set are true, all others are false. An element of set $\mathcal{P}(Inp)$ then defines all possible input valuations for one transition. E.g., given the input variables $Inp = \{a, b\}$, the boolean function

$$(a \ \wedge \ \neg b) \ \vee \ (a \ \wedge \ b) = a$$

is represented by set $\{\{a\}, \{a, b\}\} \in \mathcal{P}(Inp)$. This example shows that variable b has no effect on the valuation, i.e., a transition labeled with formula a may be taken independently of the input variable b.

Definition 3.9 *An I/O-Interval Structure is a tuple*

$$\Im_{I/O} \overset{def}{=} \langle Pr, Pr_{input}, S, s_0, T, L, I, I_{input} \rangle,$$

where

- *the components Pr, S, s_0, L, and I are defined analogously to Interval Structures,*
- *Pr_{input} is a finite set of atomic input propositions,*
- *the transition relation connects pairs of states and inputs: $T \subseteq S \times S \times Inp$.*
  Recall that Inp is the power set of input variables, $Inp = \mathcal{P}(Pr_{input})$.
- *$I_{input} : T \to \mathcal{P}(Inp)$ is a transition input labeling function.*

In T, the relevant input variables are defined for each transition, while in I_{input}, the valid valuations of input variables are defined that enable a transition to fire. For accessing the first component of a transition $t \in T$, we write $t[1]$. We require the following restriction on input labels:

$$\begin{aligned}\forall t_1, t_2 \in T :\ & (t_1[1] = t_2[1] \ \wedge\ t_1 \neq t_2) \\ & \Rightarrow (I_{input}(t_1) = I_{input}(t_2) \ \vee\ I_{input}(t_1) \cap I_{input}(t_2) = \varnothing).\end{aligned}$$

The restriction above ensures that if there are multiple transitions starting in the same state, their input restrictions are either equal or disjoint. With this restriction, the input valuations on transitions can be clustered as follows.

Definition 3.10 *The cluster function C computes all input valuations of a cluster represented by an arbitrary $i \in Inp$.*

$$C \overset{def}{=} \begin{cases} S \times Inp \ \to \mathcal{P}(Inp) \\ (s,i) \quad \mapsto \begin{cases} I_{input}(t) & if\ \exists s' \in S, \exists i' \in Inp : \\ & \quad t = \langle s, s', i' \rangle \in T \ \wedge\ i \in I_{input}(t) \\ \varnothing & otherwise \end{cases} \end{cases}$$

Because of the restriction on input labels in Definition 3.9, all clusters $C(s,i)$ of one state $s \in S$ are disjoint.

For a definition of the dynamic semantics of I/O-Interval Structures, the maximal state time has to be formalized.

Definition 3.11 *The maximal state time $MaxTime : S \times Inp \to \mathbb{N}$ is the maximal delay time of all outgoing transitions, i.e.,*

$$MaxTime \overset{def}{=} \begin{cases} S \times Inp \ \to \mathbb{N} \\ (s,i) \quad \mapsto max\{\ v \mid \exists s' \in S, \exists i' \in Inp : \\ \qquad\qquad t = \langle s, s', i' \rangle \in T \wedge \\ \qquad\qquad i \in I_{input}(t) \ \wedge v = max(I(t))\ \} \end{cases}$$

In addition to the current state and elapsed time, configurations of I/O-Interval Structures also have to consider the current inputs.

Definition 3.12 *An* I/O-IS-configuration $g = \langle s, i, v\rangle$ *is an IS-configuration* $\langle s, v\rangle$ *enriched by an input valuation* $i \in Inp$*. The set of all I/O-IS-configurations is given by*

$$G_{I/O} \overset{def}{=} \{\langle s, i, v\rangle \mid s \in S \ \wedge \ i \in \bigcup_{i' \in Inp} C(s, i') \ \wedge \ 0 \leq v \leq MaxTime(s, i)\}$$

We are now able to define the dynamic semantics of I/O Interval Structures by means of *runs*.

Definition 3.13 *Let* $\Im_{I/O} = \langle Pr, Pr_{input}, S, T, L, I, I_{input}\rangle$ *be an I/O-Interval Structure. A* run

$$r \overset{def}{=} (g_0, g_1, \ldots)$$

is a sequence of I/O-IS-configurations with $g_j = \langle s_j, i_j, v_j\rangle \in G_{I/O}$*, and for all* $j \in \mathbb{N}_0$ *holds either*

$$\begin{aligned} g_{j+1} &= \langle s_j, i_{j+1}, v_j + 1\rangle \\ & \quad with\; i_{j+1} \in C(s_j, i_j) \ \wedge \ v_j + 1 < MaxTime(s_j, i_j), \\ or & \\ g_{j+1} &= \langle s_{j+1}, i_{j+1}, 0\rangle \\ & \quad with\; t = \langle s_j, s_{j+1}, i_{j+1}\rangle \in T \ \wedge \ i_j \in I_{input}(t) \ \wedge \ v_j + 1 \in I(t). \end{aligned}$$

When we require that for every I/O-IS-configuration and for every input valuation there has to be a successor I/O-IS-configuration, we need to introduce a specific *failure state* for each of those transitions that have input restrictions and a delay time greater than 1. The failure state is entered if the current input valuation does not fulfill the input restriction any more. This leads to the following cases to be distinguished:

1. In the simplest case, a transition has no input restriction. Behavior is then as before in Interval Structures.

2. A unit-delay transition has an input restriction. Then it must be ensured that for all input valuations a successor state is specified.

3. A transition has an input restriction and a delay time $\delta > 1$. Then an additional transition with interval $[1, \delta - 1]$ and no input restriction has to connect the transition source state with a failure state.

To illustrate these cases, a graphical notation for I/O-Interval Structures is given in Figure 3.8. Input variables are denoted by $a_1, \ldots, a_n$, and f is a function with $f : (Pr_{input})^n \rightarrow \{true, false\}$.

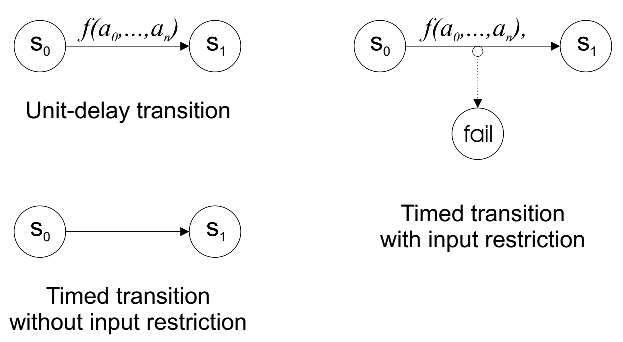


Figure 3.8: Graphical Notation for I/O-Interval Structures [RK99]

3.6.1.2 Extended I/O-Interval Structures

Finally, we extend I/O-Interval Structures by variables over finite value sets. Additionally, transitions can be attached by assignments and input conditions may carry complex boolean expressions over variables. These extensions are basically syntactical issues that can easily be mapped to and expressed by simple I/O-Interval Structures.

Definition 3.14 *An I/O-Interval Structure with variables and output signals is a tuple*

$$\Im_{I/O}^{Var} \overset{def}{=} \langle Q, Pr, Pr_{input}, Pr_{output}, S, s_0, Var_0, T, L, I, I_{input}, I_{output}, T_{assgn} \rangle \; .$$

- *The components Pr_{input}, S, T, L, and I are defined analogously to I/O-Interval Structures.*
- *$Q \overset{def}{=} \{var_1, \dots, var_n\}$ is a set of variables with finite value sets $Val(var_i)$. We require*

  $$\forall i \in \{1, \dots, n\} : \exists range_i \in \mathbb{N} : Val(var_i) = \{0, \dots range_i\},$$

  i.e., each variable is defined over a finite integer interval $[0, \dots, range_i]$.

  Alternatively, a variable may also be declared as an enumeration of identifiers over a given alphabet or as bitvectors $\{0, 1\}^n$. As these can easily be mapped to finite integer intervals as defined above, we do not explicitly consider these alternatives in the remainder.
- *The elements in Pr are atomic propositions over Q, i.e.,*

  $$Pr \overset{def}{=} \{(var_i \equiv val_{var_i}) \mid 1 \leq i \leq |Q| \wedge \\ var_i \in Q \;\wedge\; val_{var_i} \in Val(var_i)\}$$

  Semantically, we interpret $(var_i \equiv val_{var_i})$ to be true when variable var_i has the value val_{var_i} in the current configuration, and false otherwise.
- *The elements of Pr_{output} are boolean variables representing output signals visible in other I/O-Interval Structures. To avoid naming conflicts, we require $Q \cap Pr_{output} \cap Pr_{input} = \varnothing$.*

- *Function $I_{output} : Pr_{output} \to \mathcal{P}(Out)$ is an output signal labeling function, where*

$$Out \overset{def}{=} \mathcal{P}(Pr \cup Pr_{input} \cup Pr_{output}).$$

  An output label *$I_{output}(sig)$ for an output signal sig is a Boolean formula over variables of set $Pr \cup Pr_{input} \cup Pr_{output}$. Analogously to elements of Inp (i.e., valuations of input variables), an element lbl of Out defines exactly one valuation of all atomic propositions: the propositions contained in lbl are true, all others are false. An element of the set $\mathcal{P}(Out)$ then defines all possible valuations for an output signal.*

- *The transition input labeling function $I_{input} : T \to \mathcal{P}(Inp \cup Out)$ may now also carry boolean expressions over output signals Pr_{output}. We here leave out a more detailed re-definition.*

- *In addition to the initial state $s_0 \in S$, we have to set initial values for all variables:*

$$\begin{aligned} Var_0 = \{ \ & (var_1 \equiv val_1), \ldots, (\, var_n \equiv val_n) \ \mid \\ & n = |Q| \\ & \wedge \, \forall i \in \{1, \ldots, n\} : var_i \in Q \ \wedge \ val_i \in Val(var_i) \\ & \wedge \, \forall j \in \{1, \ldots, n\} : (i \neq j \Rightarrow var_i \neq var_j) \qquad \} \end{aligned}$$

- *We assume that there is a simple expression language $Assgn_Q$ available for specifying assignments to variables of set Q.*

- *The transition assignment labeling function $T_{assgn} : T \to \mathcal{P}(Assgn_Q)$ defines updates on the variables in Q. If for a transition $t \in T$, an explicit update on a variable $var \in Q$ is missing in $T_{assgn}(t)$, we have two choices:*

  1. *The value of var gets an arbitrary value when t is fired, or*
  2. *the value of var is kept unchanged when t is fired (this is also referred to as* automatic signal equivalence*).*

  We here take automatic signal equivalence semantics as default.

- *The execution semantics are the same as for I/O-Interval Structures. We assume that assigning new values to variables is executed without consuming time. Output signals are synchronously visible in all I/O-Interval Structures that are part of the model under consideration.*

3.6.2 Clocked Computation Tree Logic

Clocked CTL (CCTL) is a time-bounded temporal logic [RK97]. In contrast to classical CTL, the temporal operators **F** (i.e., eventually), **G** (globally), and **U** (until) are provided with interval time-bounds $[a, b]$, $a \in \mathbb{N}_0, b \in \mathbb{N}_0 \cup \{\infty\}$. The symbol ∞ is defined through: $\forall i \in \mathbb{N}_0 : i < \infty$, and it holds $i + \infty = \infty$ and $i - \infty = \infty$. These temporal operators can also have a single time-bound only. In this case the lower bound is set to zero by default. If no interval is specified, the lower bound is zero and the upper bound is infinity by default. The **X**-operator (i.e., next) can have a single time-bound $[a]$ only ($a \in \mathbb{N}$). If no time bound is specified, it is implicitly set to one.

The syntax of CCTL is recursively defined by the following grammar:

$$\begin{array}{rlll}
\phi ::= & p \mid \text{true} \mid \text{false} & \mid \phi \wedge \phi \mid \phi \vee \phi & \mid \neg\phi \\
& \mid \mathrm{EX}_{[a]}\ \phi & \mid \mathrm{EF}_{[a,b]}\ \phi & \mid \mathrm{EG}_{[a,b]}\ \phi \\
& \mid \mathrm{E}(\phi\ \underline{\mathrm{U}}_{[a,b]}\ \phi) & \mid \mathrm{E}(\phi\ \mathrm{U}_{[a,b]}\ \phi) & \\
& \mid \mathrm{E}(\phi\ \underline{\mathrm{S}}_{[a]}\ \phi) & \mid \mathrm{E}(\phi\ \mathrm{C}_{[a]}\ \phi) & \\
& \mid \mathrm{AX}_{[a]}\ \phi & \mid \mathrm{AF}_{[a,b]}\ \phi & \mid \mathrm{AG}_{[a,b]}\ \phi \\
& \mid \mathrm{A}(\phi\ \underline{\mathrm{U}}_{[a,b]}\ \phi) & \mid \mathrm{A}(\phi\ \mathrm{U}_{[a,b]}\ \phi) & \\
& \mid \mathrm{A}(\phi\ \underline{\mathrm{S}}_{[a]}\ \phi) & \mid \mathrm{A}(\phi\ \mathrm{C}_{[a]}\ \phi) &
\end{array}$$

where $p \in Pr$ is a proposition, $a \in \mathbb{N}_0$, and $b \in \mathbb{N}_0 \cup \{\infty\}$. For the symbol ∞, we define $\forall i \in \mathbb{N}_0 : i < \infty$.

The semantics of CCTL is defined as a validation relation ”$\models$”, using the notion of *runs*, which represent possible sequences of clocked states that occur during execution of $\Im$. Any arbitrary clocked state g_0 may be the starting point of a run. Table 3.5 shows some sample semi-formal descriptions of the validation relation for a given Interval Structure $\Im$ and a clocked state $g_0 = (s_0, v_0) \in G$. Note that ϕ and ψ denote arbitrary CCTL (sub)formulae.
The semantics for temporal operators with path quantifier **A** (i.e., regarding *all* possible runs) can easily be derived, e.g., $\mathbf{AX}_{[a]}\phi$ is equivalent to $\neg\mathbf{EX}_{[a]}\neg\phi$. Another example is $\mathbf{AF}_{[a,b]}\phi$, which is equivalent to $\neg\mathbf{EG}_{[a,b]}\phi$.

Extended CCTL Syntax. Analogously to the extension of I/O-Interval Structures, CCTL can also be extended to allow for variables over finite value sets. The syntax is extended by the following new rules for variable-based formulas.

$$\phi ::= (var = val) \mid (var > val) \mid (var >= val) \mid (var < val) \mid (var <= val)$$

where $var \in Q$ is a variable and $val \in Val(var)$ is a value of a given I/O Interval Structure $\Im_{I/O}^{Var}$ The semantics of CCTL are kept unchanged, as the new variable-based formulas still evaluate to a boolean value.

Example. For property specification, consider the following example. One requirement in our case study is that the input buffer of a station must not be blocked for too long in order to guarantee sufficient continuous workload, i.e., each accepted delivery request must be followed by actually loading an item at the input buffer within 100 time units after acceptance. Due to the dependency on other modules, in particular the AGVs, it is not obvious whether the model satisfies this property. Therefore, a corresponding CCTL formula has to be specified:

Table 3.5: Semi-formal Description of CCTL Operators

Formula	Denotation	Description
$g_0 \models p\ \ (p \in Pr)$	Proposition	g_0 is valid in p, if $p \in L(s_0)$
$g_0 \models \neg\phi$	Negation	g_0 is satisfied by $\neg\phi$ if $g_0 \models \phi$ is false.
$g_0 \models (\phi \wedge \psi)$	Conjunction	$g_0 \models \phi$ and $g_0 \models \psi$
$g_0 \models (\phi \vee \psi)$	Disjunction	$g_0 \models \phi$ or $g_0 \models \psi$
$g_0 \models \mathbf{EX}_{[a]}\ \phi$	Next	There exists a run $r = (g_0, \ldots)$ such that $g_a \models \phi$
$g_0 \models \mathbf{EF}_{[a,b]}\ \phi$	Eventually	There exists a run $r = (g_0, \ldots)$ and $a \leq i \leq b$ s.t. $g_i \models \phi$
$g_0 \models \mathbf{EG}_{[a,b]}\ \phi$	Globally	There exists a run $r = (g_0, \ldots)$ s.t. for all $a \leq i \leq b$ holds $g_i \models \phi$
$g_0 \models \mathbf{E}(\phi\ \underline{\mathbf{U}}_{[a,b]}\ \psi)$	Strong Until	There exists a run $r = (g_0, \ldots)$ and an $a \leq i \leq b$ s.t. $g_i \models \psi$ and for all $j < i$ holds $g_j \models \phi$
$g_0 \models \mathbf{E}(\phi\ \mathbf{U}_{[a,b]}\ \psi)$	Weak Until	There exists a run $r = (g_0, \ldots)$ and and either (a) there exists an $a \leq i \leq b$ s.t. $g_i \models \psi$ and for all $j < i$ holds $g_j \models \phi$, or (b) for all $i \leq b$ holds $g_i \models \phi$
$g_0 \models \mathbf{E}(\phi\ \mathbf{S}_{[a]}\ \psi)$	Successor	There exists a run $r = (g_0, \ldots)$ s.t. $g_a \models \psi$ and for all $i < a$ holds $g_i \models \phi$
$g_0 \models \mathbf{E}(\phi\ \mathbf{C}_{[a]}\ \psi)$	Conditional	There exists a run $r = (g_0, \ldots)$ for that holds: if $g_i \models \phi$ for all $i < a$, then $g_a \models \psi$

```
AG((acceptor.state = acceptor.accepting)
  -> AF[100]((loader.state = loader.waitingForDelivery)
             & AX(loader.state = loader.loading)
            )
  )
```

If RAVEN evaluates a CCTL formula to be incorrect, a counter example execution run can be generated. Execution runs are given by time-annotated sequences of state changes. RAVEN invokes a built-in waveform browser that lists all variables and their states over time.

3.6.3 RAVEN Input Language (RIL)

In the context of RAVEN, I/O-Interval Structures and a set of CCTL formulae are specified by means of the textual RAVEN Input Language (RIL). A RIL specification contains

(a) a set of global definitions, e.g., fixed time bounds or frequently used formulae,

(b) the specification of parallel running *modules*, i.e., a textual specification of I/O-Interval Structures, and

(c) a *specification* with a set of CCTL formulae, representing required properties of the model,

(d) an *analyze* section with a set of analysis formulae that extract times for minimal/maximal state transition times.

The following code is a fragment of the RIL model for the manufacturing case study.

```
MODULE agv1_negotiator
        // the internal states are declared in the SIGNALS compartment
SIGNALS
  state : { waitingForOrder computingBid waitingForAcknowledgement }
  dest  : { in ou ao pm mi mo am pd di do ad pw wi wo aw c1 c2 c3 c4 }
  order : BOOL
  currentItem : RANGE[0,20]

INPUTS                           // signals visible from other modules
  disp_requestTransport_inpStation := (inpStation_requestTransport_agv1.state = true)
  disp_requestTransport_mill       := (mill_requestTransport_agv1.state = true)
  disp_requestTransport_drill      := (drill_requestTransport_agv1.state = true)
  disp_requestTransport_wash       := (wash_requestTransport_agv1.state = true)

  ... // further inputs omitted

DEFINE                           // declaration of output signals
  bidding_inpStation   := (sendBidding_inpStation = true)
  bidding_mill         := (sendBidding_mill = true)
  bidding_drill        := (sendBidding_drill = true)
  bidding_wash         := (sendBidding_wash = true)

  ... // further output signals are omitted

INIT                             // the initial value of the internal states
  (state = waitingForOrder) & (order = false)

TRANS                            // transitions of the module
  |- state=waitingForOrder
       --  disp_requestTransport_inpStation & !order :1 --> dest:=in;
                                                            state:=computingBid
       --  disp_requestTransport_mill       & !order :1 --> dest:=mi;
                                                            state:=computingBid
       --  disp_requestTransport_drill      & !order :1 --> dest:=di;
                                                            state:=computingBid
       --  disp_requestTransport_wash       & !order :1 --> dest:=wi;
                                                            state:=computingBid
  ... // other transitions omitted
```

In RIL, we have to specify modules (i.e., I/O-Interval Structures) on the instance level, i.e., for each object, we have (at least one) module.[7] In the code shown above, we consider parts of the negotiation behavior of an AGV named `agv1`.

In the `SIGNALS` compartment, the variables (or: attributes) of an object are declared. By default, variable `state` comprises the states of the corresponding State Diagram part. As RAVEN

[7]This – among other restrictions – implies that we have to know in advance how many objects are created for a concrete system. Later, we define corresponding rules for our application domain that restrict UML models to be applicable to be mapped to I/O-Interval Structures.

only supports finite value sets, Integer values must be restricted to some finite value domain, e.g., here the interval `[0,20]` is chosen for attribute `currentItem`.

The set of dispatched events that trigger a transition is defined in the `INPUTS` compartment of input signals. In the code listed, the prefix `disp_` shall indicate that these input signals represent *dispatched events*. Event dispatching – in turn – is treated in separate modules and not shown here.

In the `DEFINE` compartment, signals visible to other modules are listed (so-called output signals). In the example, we use output signals to represent signals sent to other stations, e.g., to send bids in reply to requests for a transport.

In the `INIT` section, internal variables that have been defined in the `SIGNALS` compartment get an initial value. This is of particular relevance for the corresponding initial State Diagram state, i.e, we have to set `state = WaitingForOrder` for our example.

The `TRANS` compartment finally lists all transitions between the states. Conditions for taking a transition are prefixed by `--`. To build more complex conditions, the usual logical operators (&, |, ! for logical and, or, not) and relational operators can be applied.

An optional timing specification is prefixed by a colon. The timing specification can be a single value or an interval. If it is omitted, the time bound is set to [1,1] by default. When a transition is taken, the assignments following the arrow `-->` are executed. The assignments can affect all variables defined in the `SIGNALS` or `DEFINE` compartment. For more complex assignments, arithmetic operators ($+$ and $-$) can be applied. More details about the syntax of RIL can be found in [Ruf01].

Specification and Analysis. Property specifications by CCTL formulae are given in the `SPEC` compartment. Each CCTL formula gets a name to refer to. But basically, the syntax of CCTL directly corresponds to the syntax as given in Section 3.6.2.

We here therefore focus on the analysis compartment with additional analysis queries. They are of the following form.

```
analysisDeclaration ::= 'ANALYSIS' analysis
analysis            ::= analysisName ':=' anatype
anatype             ::= 'MIN STABLE TIME OF' '(' cctlFormula ')'
                        | 'MAX STABLE TIME OF' '(' cctlFormula ')'
                        | 'MIN TIME FROM' '(' cctlFormula ')' 'TO' '(' cctlFormula ')'
                        | 'MAX TIME FROM' '(' cctlFormula ')' 'TO' '(' cctlFormula ')'
```

The grammar for `cctlFormula` is basically the same as for ordinary CCTL formulas. Only some additional features are defined, e.g., the predefined variable `INIT` to refer to the overall initial state of the model.

Analysis queries allow to compute time delays, i.e., minimal and maximal reaction times or maximal wait times. A `MIN STABLE` query takes a CCTL formula as a parameter and computes the minimal time for which the formula is true during execution of the corresponding model. Analogously, a `MAX STABLE` query takes a CCTL formula and computes the maximal time for which the formula is true during execution of the model. For example,

```
agv1_remainingInWaitingState :=
  MAX STABLE TIME OF (agv1_negotiator.state=agv1_negotiator.waitingForOrder)
```

is a query to determine the maximal time in which `agv1` remains in state `waitingForOrder`.[8]

`MIN TIME` and `MAX TIME` require two CCTL formulae as parameters. They compute the minimal/maximal delay time between two conditions becoming true. The first parameter determines the *start configuration set*, i.e., those model states and time points, in which the first CCTL formula is true. The second CCTL formula then determines the *destination configuration set*, i.e., those model states and time points, in which the second CCTL formula is true. `MIN` and `MAX` compute the minimal and maximal time distances between these two configuration sets. For example,

```
agv1_maxTime_firstAccept :=
    MAX TIME FROM (INIT) TO ( agv1_negotiator.sendAcceptBid_inpStation
                             | agv1_negotiator.sendAcceptBid_mill
                             | agv1_negotiator.sendAcceptBid_drill
                             | agv1_negotiator.sendAcceptBid_wash )
```

is a query to determine the maximal time until the AGV accepts to take an order for the first time.

3.6.4 Graphical User Interface

While the RAVEN verification engine can be called directly from the console by a purely textual command, the graphical RAVEN user interface is of great help due to the large number of options that can be applied. We here only outline the main features of the GUI and refer to [Ruf01] for more details.

The RAVEN GUI consists of three parts, i.e., (1) status information at the top, (2) main buttons and different control sheets in the middle, and (3) log information at the bottom. The four different control sheets are of special interest and need further explanation. The *Composition* control sheet shown in Figure 3.9(a) on page 84 allows to select from different algorithms, optimizations, and heuristics to build the internal representation of a given RIL model. Compared to the standard expand-compose-reduce algorithm, other techniques may lead to faster compilations, e.g., the algorithm 'combined' performs the composition and reduction phases in one step. The RAVEN manual states that the latter algorithm is preferrably to be used for large models in the *multiple delay mode*. RAVEN basically distinguishes unit delay and multiple delay mode (see Figure 3.9(b)). In multiple delay mode, a dedicated representation called MTBDD (multi terminal binary decision diagram) is used that allows to use further optimization techniques for model checking (e.g., time prediction and time jumps). In unit delay mode, delay times of transitions are simply encoded by introducing additional stutter states. Other preferences regard checks for correct clustering (see Definition 3.10) and dead-/livelocks.

The *Model Checking and Analysis* control sheet (Figure 3.9(c)) lists the parsed CCTL specifications and timing analysis formulae. Again, different optimization techniques can be enabled (e.g., 'time jump' in multiple delay mode or 'abstraction' for untimed specifications). Enabling 'counter example' leads to the generation of an execution trace when a CCTL formula with

[8]Actually, the state formula becomes more complex when the the mapping of UML State Diagrams to I/O-Interval Structures presented in Section 5.4 is considered. See Section 7.3 for more details.

A-quantifiers is falsified. Pressing the button ’check’ starts the verification with the chosen options. With the buttons ’show’ and ’new’, one can edit and update specifications directly within the user interface. The ’reset’ button sets the proof state to unproved, such that a verification can be performed again with different options. The analysis compartment in the lower part of the control sheet shows the parsed timing analysis formulae. Its options and buttons can be used in a very similar way. The simulation compartment is for randomly generating an example trace of the model. Users can set the desired number of execution steps with a slider.

Finally, the *Recourses and Statistics* sheet shown in Figure 3.9(d) provides information about the run times that were necessary for the different tasks of the verification. In particular, the BDD-PACKAGE window gives detailed information about the generated internal representation of the investigated model.

3.7 Contributions of the Chapter

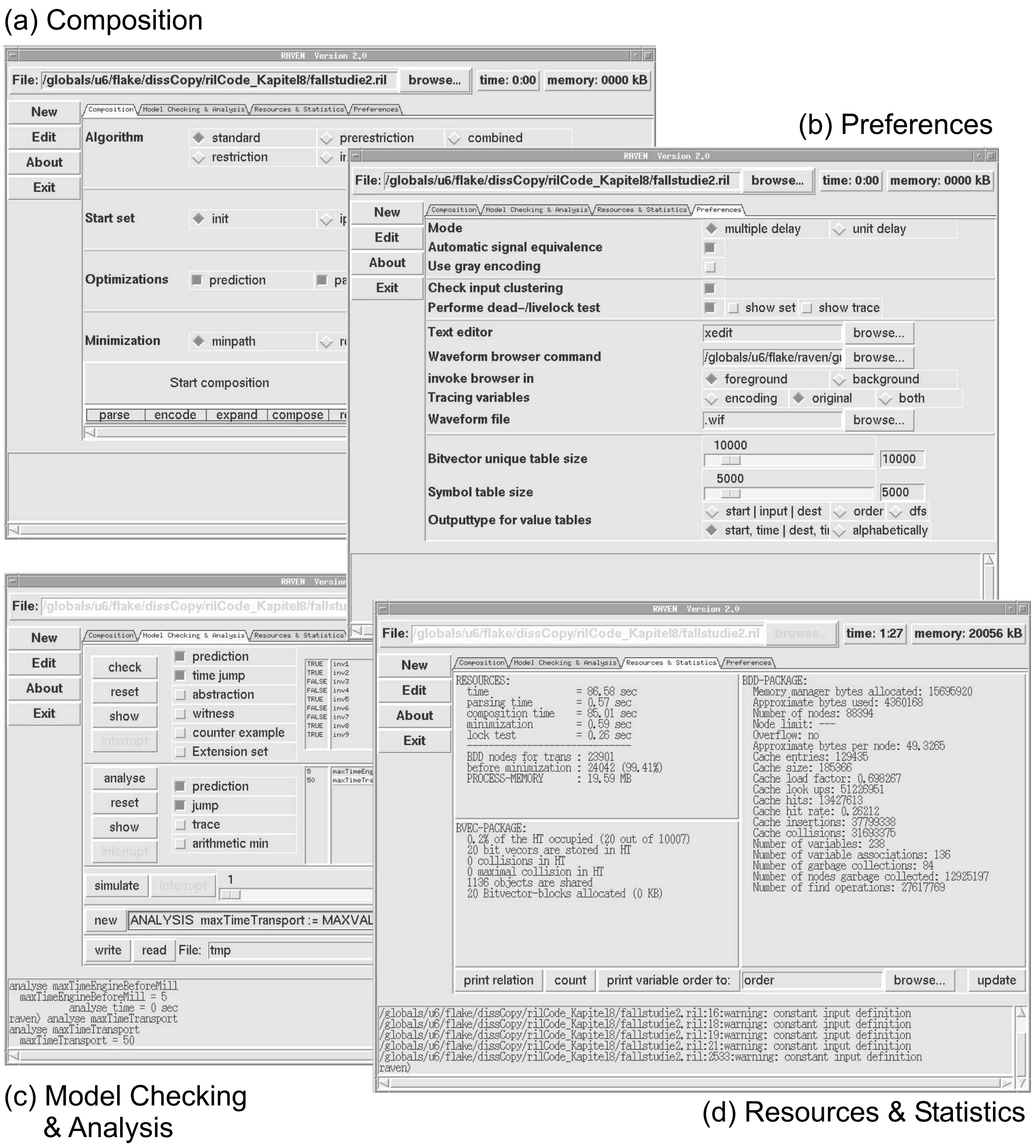
This chapter provides the following contributions:

- An overview of formal modeling and specification approaches w.r.t. the formal verification method of model checking, especially real-time model checking, is given. The property specification pattern system by Dwyer et al. is reviewed. We illustrated that the CTL temporal logic formulas of that pattern system could be simplified under certain reasonable assumptions.

- An evaluation of existing model checkers is made w.r.t. requirements relevant for the considered domain. The RAVEN model checker is chosen to be applied in the context of this thesis.

- I/O-Interval Structures are outlined. They build the underlying formal model of the RAVEN model checker. Note that Section 3.6.1.2 is a new contribution developed in the context of this thesis. In that section, the formal model of I/O-Interval Structures is extended by variables over finite value sets, transitions that can be annotated by assignments, and input conditions that may carry complex boolean expressions over variables.

- Correspondingly, the syntax of CCTL is extended to allow variables over finite value sets and boolean operators to compare their values.



File: /globals/u6/flake/dissCopy/rilCode_Kapitel8/fallstudie2.ril time: 0:00 memory: 0000 kB

Algorithm: [✓ standard] prerestriction combined restriction
Start set: [✓ init]
Optimizations: [✓ prediction]
Minimization: [✓ minpath]

Mode: [✓ multiple delay] unit delay
Automatic signal equivalence: [✓]
Use gray encoding: [ ]
Check input clustering: [✓]
Performe dead-/livelock test: [✓] show set show trace
Text editor: xedit
Waveform browser command: /globals/u6/flake/raven/g
invoke browser in: [✓ foreground] background
Tracing variables: encoding [✓ original] both
Waveform file: .wif
Bitvector unique table size: 10000
Symbol table size: 5000
Outputtype for value tables: start | input | dest order dfs [✓ start, time | dest, ti] alphabetically

Check: [✓ prediction] [✓ time jump] abstraction witness counter example Extension set
Analyse: [✓ prediction] [✓ jump] trace arithmetic min

```
TRUE   inv1
TRUE   inv2
FALSE  inv3
FALSE  inv4
TRUE   inv5
FALSE  inv6
FALSE  inv7
TRUE   inv8
TRUE   inv9
```

```
5    maxTimeEng
50   maxTimeTra
```

ANALYSIS maxTimeTransport := MAXVAL
File: tmp

```
analyse maxTimeEngineBeforeMill
  maxTimeEngineBeforeMill = 5
          analyse time = 0 sec
raven> analyse maxTimeTransport
analyse maxTimeTransport
  maxTimeTransport = 50
```

File: /globals/u6/flake/dissCopy/rilCode_Kapitel8/fallstudie2.ril time: 1:27 memory: 20056 kB

```
RESOURCES:
  time                = 86.58 sec
  parsing time        = 0.57 sec
  composition time    = 85.01 sec
  minimization        = 0.59 sec
  lock test           = 0.26 sec
  ------------------------------
  BDD nodes for trans : 23901
  before minimization : 24042 (99.41%)
  PROCESS-MEMORY      : 19.59 MB

BVEC-PACKAGE:
  0.2% of the HT occupied (20 out of 10007)
  20 bit vecors are stored in HT
  0 collisions in HT
  0 maximal collision in HT
  1136 objects are shared
  20 Bitvector-blocks allocated (0 KB)
```

```
BDD-PACKAGE:
  Memory manager bytes allocated: 15695920
  Approximate bytes used: 4360168
  Number of nodes: 88394
  Node limit: ---
  Overflow: no
  Approximate bytes per node: 49.3265
  Cache entries: 129435
  Cache size: 185366
  Cache load factor: 0.698267
  Cache look ups: 51226951
  Cache hits: 13427613
  Cache hit rate: 0.26212
  Cache insertions: 37799338
  Cache collisions: 31693375
  Number of variables: 238
  Number of variable associations: 136
  Number of garbage collections: 84
  Number of nodes garbage collected: 12925197
  Number of find operations: 27617769
```

print variable order to: order

```
/globals/u6/flake/dissCopy/rilCode_Kapitel8/fallstudie2.ril:16:warning: constant input definition
/globals/u6/flake/dissCopy/rilCode_Kapitel8/fallstudie2.ril:18:warning: constant input definition
/globals/u6/flake/dissCopy/rilCode_Kapitel8/fallstudie2.ril:19:warning: constant input definition
/globals/u6/flake/dissCopy/rilCode_Kapitel8/fallstudie2.ril:21:warning: constant input definition
/globals/u6/flake/dissCopy/rilCode_Kapitel8/fallstudie2.ril:2533:warning: constant input definition
raven>
```

Figure 3.9: RAVEN Graphical User Interface

Chapter 4

Extended Object Model

The sciences do not try to explain,
they hardly even try to interpret,
they mainly make models.
By a model is meant a mathematical construct which,
with the addition of certain verbal interpretations,
describes observed phenomena.
The justification of such a mathematical construct
is solely and precisely that it is expected to work.
– John Von Neumann (1903 – 1957)

This section formally defines the syntax and semantics of *extended object models* that take State Diagrams as a behavioral description of active classes into account. The notion of an extended object model is based upon a formalization of the object model as presented by Richteres in [Ric01]. Note that a number of definitions are adopted from that work, but the following concepts are additionally introduced in this chapter:

- Signals for classes together with well-formedness rules,
- generalization of signals,
- State Diagrams and their relation to classes,
- an extension of the formal descriptor of a class,
- an extension of the formal definition of a system state, and
- a formal definition of system state sequences.

In Section 4.1, the syntax of extended object models is formally defined. In particular, we define the syntax of (a significant part of) UML State Diagrams in Section 4.1.3. The semantics of extended object models is defined in Section 4.2. The formalization of State Diagram state configurations in Section 4.2.3 is of particular interest, as the deficiencies of the current informal notion of *active state configurations* for UML State Diagrams are identified and resolved. Based upon a definition of extended *overall system states* as snapshots of an executed model in Section

4.2.5, we define a high-level notion of *execution traces*, i.e., sequences of overall system states, that comprise all necessary information to evaluate OCL expressions also w.r.t. state-oriented operations. Section 4.3 then reviews the presented extension of the object model w.r.t. the completion of the formal semantics of OCL 2.0.

4.1 Syntax

An *Extended Object Model* is a tuple

$$\begin{aligned}\mathcal{M} \overset{def}{=} \langle\ & CLASS, ATT, OP, SIG, isQuery, paramKind, SC,\\ & ASSOC, \prec, \prec_{SIG}, associates, roles, multiplicities\ \rangle\end{aligned}$$

with

- a set $CLASS$ of classes, $CLASS \overset{def}{=} ACTIVE \cup PASSIVE$,
- a set ATT of attributes, $ATT \overset{def}{=} \bigcup_{c \in CLASS} ATT_c$,
- a set OP of operations, $OP \overset{def}{=} \bigcup_{c \in CLASS} OP_c$,
- a function $isQuery : CLASS \times OP \rightarrow Boolean$ that determines whether an operation is a query operation or not,
- a function $paramKind : CLASS \times OP \times \mathbb{N} \rightarrow \{in, inout, out\}$ that gives for each operation parameter its parameter kind,
- a set SIG of signals, $SIG \supseteq \bigcup_{c \in CLASS} SIG_c$,
- a set SC of State Diagrams, $SC \overset{def}{=} \bigcup_{c \in ACTIVE} SC_c$,
- a set $ASSOC$ of associations,
- generalization hierarchies $\prec$ for classes and $\prec_{SIG}$ for signals, and
- functions $associates$, $roles$, and $multiplicities$ that give for each association $as \in ASSOC$ its dedicated classes, their role names, and multiplicities, respectively.

In the following, each of the tuple elements is considered in detail. For element names in $\mathcal{M}$, let $\mathcal{A}$ be an alphabet and $\mathcal{N} \subseteq \mathcal{A}^+$ a set of finite, non-empty names.

4.1.1 Types

We assume that there is a set $\Sigma \overset{def}{=} (T, \Omega)$, where $T \subseteq \mathcal{P}(\mathcal{N})$ is a set of type names and Ω a set of operation signatures over types in T. In particular, $T \overset{def}{=} T_B \cup T_E \cup T_C \cup T_S$ comprises

- a set of basic standard library types T_B, i.e., $Integer$, $Real$, $Boolean$, and $String$,
- a set T_E of user-defined enumeration types,
- a set T_C of user-defined classes, and
- a set of special types $T_S \overset{def}{=} \{OclVoid, OclState, OclAny\}$.

The elements of the types $t \in T$ are kept in value sets $I_{TYPE}(t)$. $I_{TYPE}(t)$ (or simply $I(t)$ when the context is clear) is called the *type domain* of $t \in T$.

`OclVoid` is a subtype of any other type and allows to operate with undefined values. The only value of `OclVoid` is called `OclUndefined` and is denoted in the following by $\perp$. For convenience, we presume that $\perp$ is included in each type domain, such that we have, e.g.,

$$\begin{aligned}
I(OclVoid) &\overset{def}{=} \{\perp\},\\
I(Integer) &\overset{def}{=} \mathbb{Z} \cup \{\perp\},\\
I(Real) &\overset{def}{=} \mathbb{R} \cup \{\perp\},\\
I(Boolean) &\overset{def}{=} \{\text{true}, \text{false}\} \cup \{\perp\},\\
I(String) &\overset{def}{=} \mathcal{A}^* \cup \{\perp\},\\
I(OclState) &\overset{def}{=} \mathcal{N} \cup \{\perp\},\\
I(OclAny) &\overset{def}{=} \big(\textstyle\bigcup_{t \in T_B \cup T_E \cup T_C} I(t) \big) \cup I(OclState).
\end{aligned}$$

As the type domain $I(OclState)$ is actually determined by the states of the State Diagrams SC_c of the referred UML user model, a more elaborated definition of $I(OclState)$ is given in Section 4.2.6.

The domains of types in T_E are simply the enumeration literals as given by the enumeration types defined in the referred UML user model. For example, w.r.t. Figure 2.2 on page 21, we have

$$T_E = \{MachineKind, AcceptState, LoaderState, ItemState, ItemKind\}, \text{ and}$$

$$I(MachineKind) = \{Mill, Drill, Wash, \perp\}.$$

Note that in the concrete OCL syntax, enumeration literals are represented by double colon notation, e.g., `MachineKind::Mill`.

The domains of types in T_C are *object identifiers* that represent instances of class c. This issue will be further discussed in Section 4.2.1.

Operations in Ω include, e.g., the usual arithmetic operations +, -, *, / for `Integer` values. Moreover, *collection types* for sets, ordered sets, sequences, and bags are defined in Σ to manage collections of values, e.g., `Set(String)`, `Bag(Integer)`, and `Sequence(Real)`.

4.1.2 Classes and their Characteristics

A class is a description for a set of objects sharing the same characteristics, i.e., attributes, operations, signals, and associations.[1] In conformance with the semantics of the adopted OCL 2.0 specification, we here do not distinguish between the UML classifier concepts of classes and interfaces. OCL constraints are specified for instances of an *interface specification*[2]. Whether such an interface specification is given in the form of a UML class or interface definition does not make a difference in the context of OCL. We first focus on attributes, operations, and signals. Associations are separately defined in the set $ASSOC$ in Section 4.1.4.

Definition 4.1 *(Classes and Types)*
The set of classes $CLASS$ is a finite set of names, $CLASS \subseteq \mathcal{N}$. $CLASS$ is the union of two disjoint sets $ACTIVE$ and $PASSIVE$ of active and passive classes,

$$CLASS \overset{def}{=} ACTIVE \cup PASSIVE.$$

Active classes specify entities capable of dynamic behavior, which is specified by an associated State Diagram (see Definition 4.5).

Each class $c \in CLASS$ induces a type $t_c \in T_C \subset T$ having the same name as the class. A value $val \in I(t_c)$ of a type $t_c \in T_C$ refers to an object of the corresponding class $c \in CLASS$.

The difference between c and t_c is that the special value $\perp$ is additionally included in $I(t_c)$ for all $c \in CLASS$. In the remainder, let $c \in CLASS$ be a class and $t_c \in T_C$ be the type of the class c.

For example, the sets of active and passive classes w.r.t. the UML model in Figure 2.2 on page 21 is

$$\begin{aligned} ACTIVE &= \{ \; FactoryUnit, AGV, Station, InputStorage, OutputStorage, \\ &\qquad Machine, Buffer, InputBuffer, OutputBuffer \; \} \text{ and} \\ PASSIVE &= \{ \; Item, NegotiationParticipantTransport, \\ &\qquad NegotiationParticipantDestination, NegotiationManager \; \} \; . \end{aligned}$$

Attributes. Classes are associated with attributes that describe characteristics of their objects. An attribute has a name and a type that specifies the domain of attribute values.

Definition 4.2 *(Attributes)*
Let $c \in CLASS$ be a class and $t_c \in T$ be the type of class c. The set of attributes of c is defined by $ATT_c \overset{def}{=} \{\langle a, t_c, t\rangle \mid a \in \mathcal{N} \wedge t \in T\}$.

[1] In this thesis, we use the term *characteristics* to refer to the elements that are called *properties* in terms of UML, because we employ the notion of a *property* in a different context. We will use the term *property* to refer to a specification of a required *behavioral* or *dynamic* property of a given model.

[2] Here, the term *interface* is used in a general sense, i.e., we are not referring to the UML classifier concept `Interface`.

In the triple $\langle a, t_c, t\rangle$, a denotes the attribute name, t_c represents the type of c to which the attribute is applied, and t is the type of a. Attribute names must be pairwise distinct, i.e.,

$$\forall att, att' \in ATT_c \text{ with } att = \langle a, t_c, t\rangle, att' = \langle a', t_c, t'\rangle : \\ att \neq att' \Longrightarrow a \neq a'.$$

For example, the attributes of class `InputBuffer` are

$$\begin{aligned} ATT_{InputBuffer} = \{ \; & \langle acceptStatus, InputBuffer, AcceptState\rangle, \\ & \langle loaderStatus, InputBuffer, LoaderState\rangle, \\ & \langle announced, InputBuffer, Boolean\rangle \; \}. \end{aligned}$$

Though the attribute names of a class must be pairwise distinct, attributes with the same name may appear in several classes which are not related by generalization (cf. well-formedness rules in Section 4.1.5).

Operations. In addition to attributes, a class may be associated with a number of operations and signals. Operations are used to describe behavioral characteristics of objects. That behavior might be specified by an associated State Diagram, but we here only consider *operation signatures* that declare an interface of operations.

Definition 4.3 *(Operations)*
Let $c \in CLASS$ be a class and $t_c \in T$ be the type of class c. The operations of class c are defined by a set OP_c of operation signatures,

$$OP_c \overset{def}{=} \{(\omega : t_c \times t_1 \times \ldots \times t_n \to t) \mid \omega \in \mathcal{N}, n \in \mathbb{N}_0, \text{ and } t, t_1, \ldots t_n \in T\}.$$

Symbol ω determines the operation name, and the first parameter t_c denotes the type of c to which operation ω belongs.

For example, the operations of the abstract class `Buffer` are

$$\begin{aligned} OP_{Buffer} = \{ \; & \langle load : Buffer \times Item \to OclVoid\rangle, \\ & \langle unload : Buffer \times Item \to OclVoid\rangle \; \}. \end{aligned}$$

Function $isQuery : CLASS \times OP \to Boolean$ determines whether an operation is a query operation without side-effects on the current status of the executed model (cf. [OMG03d, Section 2.5.2.7]). Only operations op of a class c with $isQuery(c, op) = \text{ true}$ are allowed to be *called* when an OCL expression is evaluated, as the evaluation of OCL expressions must not have side effects on the actual status of the referred UML user model.

Though not explicitly shown in Figure 2.2 on page 21, the following operations are query operations:

$$\begin{aligned} & \langle getDistance : AGV \times FactoryUnit \to Position\rangle, \\ & \langle getParkPos : AGV \times Station \to Position\rangle, \\ & \langle getInputPos : AGV \times Station \to Position\rangle. \end{aligned}$$

Operation Parameter Kinds. Note that UML generally allows operation parameters to be of kind `in`, `out`, `inout`, or `result` [OMG03d, Sect. 2.5.2.31]. The current official OCL specification as well as the object model definition by Richters do not consider parameter types. However, the adopted OCL 2.0 specification now considers parameter kinds.

Therefore, we introduce function $paramKind : CLASS \times OP \times \mathbb{N} \rightarrow \{in, inout, out\}$ gives for each formal parameter its parameter kind [OMG03d, Section 2.5.2.31]. Set $\mathbb{N}$ is used to access individual parameters, i.e., for an operation signature $op = (\omega : t_c \times t_1 \times \ldots \times t_n \rightarrow t)$, position i, $1 \leq i \leq n$, refers to the parameter of type t_i.

Parameter kind in represents an input parameter that is not changed after operation execution. Parameters of kind out are output parameter that are unassigned at the time of operation call. They are assigned with a specific value when the operation call returns. Parameters of kind $inout$ are a combination of the two previous kinds, i.e., they provide an input value for the operation, and this value might be changed when the operation returns (this is also known as call-by-value-and-result).

OCL 2.0 assumes that at most one parameter of kind `result` is specified [OMG03b, Appendix A.2.1.2 and A.3.2]. If neither a result type nor any `inout` or `out` parameters are specified for an operation, we set the result type t to the predefined type `OclVoid`. If there are `inout` or `out` parameters specified, the operation result type is a tuple in which the relevant parameter values appear in their specified order, including the result value (if any) as the last element.

However, in this formalization we employ some simplifications without loss of generality. First, we simply always consider the complete tuple of operation parameters, i.e., parameters of kind `in` are always considered. All we have to require is that the values of input parameters must not change when the operation call returns. And then we also always include a return value, even if the operation return type is `OclVoid`. In this case, the return value is simply set to $\perp$.

Signals. Signals are an asynchronous communication mechanism of UML. When a signal is sent, the calling object simply continues its execution, while synchronous operation calls make the invoking operation wait for a return value. In contrast, an asynchronous operation call is like sending a signal, but note that a potential return value is simply discarded.

Richters has not considered asynchronous signals in his formal model. Reactions on signals received by an object obj are specified by a State Diagram associated with the class to which obj belongs. Consequently, when integrating State Diagrams into the formal object model, signals now also have to be regarded as well.

In UML, signals are classifiers, i.e., signals are generalizable model elements defined independently of the classes handling them. The set SIG in the model description defines all signals of a model. As we support generalization of signals, SIG is a superset of the individual signal sets SIG_c. The set SIG_c of signals that can be handled by objects of a class c is specified by so-called *receptions* [OMG03d, Sect. 3.26.6]. Note that signals can only be handled by instances of active classes, as passive classes do not have associated State Diagrams.

Definition 4.4 *(Signals)*
The signals that can be handled by instances of a class $c \in ACTIVE$ are defined by the set

SIG_c *of signal receptions,*

$$SIG_c \overset{def}{=} \{(\omega : t_c \times t_1 \times \ldots \times t_n) \,|\omega \in \mathcal{N}, n \in \mathbb{N}_0, \text{ and } t_1, \ldots, t_n \in T\}.$$

Symbol ω denotes the signal name, and t_c refers to the type of c to which signal ω is applied. As signals are asynchronous, no return value is expected, such that all signal parameters are all input parameters.

For example, the negotiation of transports to be performed is modeled by signal communications in the UML Class Diagram shown in Figure 2.2 on page 21.

Visibility of Attributes, Signals, and Operations. Though supported in UML Class Diagrams, visibility features such as `private`, `protected`, or `public` are not reflected in the formal object model. In the adopted OCL 2.0 specification, all model elements are considered visible [OMG03b, Section 9.2.2], although it is also mentioned that tools may employ UML visibility rules, i.e., only allow OCL expressions to be specified over model elements visible from the expression's context.

4.1.3 Abstract Syntax of State Diagrams

The UML 1.5 StateMachine package specifies concepts for modeling discrete behavior through finite state-transition systems [OMG03d, Section 2.12]. The provided state machine formalism is an object-based variant of Harel Statecharts [Har87]. Though state machines are applicable to various model elements within UML, the graphical form of *UML State Diagrams* is most frequently used to model the reactive behavior of class instances.

UML allows multiple State Diagrams to be applied to a single class. The reason for this is that it should be possible to associate different State Diagram to a class in different phases of development, e.g., in the analysis and in the design phase. However, we here require that there is one State Diagram SC_c for each $c \in ACTIVE$.

Note that UML *submachine states* and *stub states* do not appear in our general definition. Submachine states are a syntactical convenience to represent a 'call' to a another state machine as a 'subroutine', using stub states as entry and exit points. Thus, a submachine state is semantically equivalent to a composite state, and we can assume that all these states have explicitly been copied into SC, such that all submachine states and stub states are eliminated.

Definition 4.5 *(Abstract Syntax of State Diagrams)*
Let $c \in CLASS$ be a class. Each $c \in ACTIVE$ has an associated State Diagram SC_c representing the reactive behavior of instances of c.

$$SC_c \overset{def}{=} \begin{cases} \begin{array}{c} \langle \quad S_c, VARS_c, TR_c, EVTS_c, GUARDS_c, ACTS_c, \\ internalTrans_c, shallowHistory_c, deepHistory_c, \\ init_c, final_c, substates_c, entry_c, exit_c, \\ doActivity_c, deferrableEvents_c \, \rangle, \end{array} & \text{if } c \in ACTIVE \\ \\ \varnothing, & \text{if } c \in PASSIVE. \end{cases}$$

To keep the definition concise, we omit the class annotator c for State Diagram components in the following and provide the general definition of a State Diagram SC, i.e.,

$$\begin{aligned} SC \overset{def}{=} \langle\ & S, VARS, EVTS, GUARDS, ACTS, TR, internalTrans, \\ & shallowHistory, deepHistory, defaultHistory, init, final, \\ & substates, entry, exit, doActivity, deferrableEvents\ \rangle, \end{aligned}$$

where

1. $S \subseteq \mathcal{N}$ is a set of states. S is the union of the following disjoint sets.

   - Pseudo states $Pseudo$, consisting of the disjoint sets of (a) initial states $Init$, (b) merging states $Join$, (c) splitting states $Fork$, (d) static conditional branch states $Junction$, (e) dynamic conditional branch states $Choice$, and (f) history states $History \overset{def}{=} ShallowHistory \cup DeepHistory$,
   - synchronization states $Synch$,
   - simple states $Simple$,
   - composite states $Composite$, which in turn comprises the two disjoint sets of sequential composite states Xor and orthogonal composite states And, and
   - final states $Final$.

   For more details about these states, see [OMG03d, Section 2.12.2]. For convenience, we define
   $$Proper \overset{def}{=} And \cup Xor \cup Simple.$$

2. $VARS \subseteq \mathcal{N}$ is a set of local variables.

3. $EVTS \subseteq EXPR_{Evts}$ is a set of events. We assume that there is an expression language $EXPR_{Evts}$ available to formulate events such as operation calls, signals, timers, etc.

4. $GUARDS \subseteq EXPR_{Guards}$ is a set of conditions. We assume that there is a language $EXPR_{Guards}$ available to formulate boolean expressions.[3]

5. $ACTS \subseteq EXPR_{Acts}$ is a set of actions. We assume that there is an expression language $EXPR_{Acts}$ available to formulate actions such as assignments, operation calls, signals, etc.

6. $TR \subseteq (S \setminus Final) \times EVTS \times GUARDS \times ACTS \times (S \setminus Init)$ is a set of transitions. A transition connects a source state $s \in S \setminus Final$ and a destination state $s' \in S \setminus Init$,

[3]In this context, boolean OCL expressions are frequently applied.

may have a trigger event $e \in EVTS$, a guard condition $g \in GUARDS$, and an action expression $a \in ACTS$. In the following, the five convenience functions

$$\begin{aligned} tr_{src} &: TR \rightarrow S \setminus Final, \\ tr_{dst} &: TR \rightarrow S \setminus Init, \\ tr_{evt} &: TR \rightarrow EVTS, \\ tr_{grd} &: TR \rightarrow GUARDS, \\ tr_{act} &: TR \rightarrow ACTS \end{aligned}$$

are used to extract the source state, destination state, event, guard, and action of a given transition, respectively.

For a transition $t = \langle s, e, g, a, s' \rangle$, we use the notation $s \xrightarrow{e[g]/a} s'$ and omit e, g, or a when $tr_{evt}(e) = \varnothing$, $tr_{grd}(g) = \varnothing$, or $tr_{act}(a) = \varnothing$, respectively. Transitions $t \in TR$ with $tr_{src}(t) = tr_{dst}(t)$ are called ***self-transitions***.

7. Function $internalTrans : Proper \rightarrow \mathcal{P}(EVTS \times GUARDS \times ACTS)$ gives the set of ***internal transitions*** for a given state $s \in Proper$. Internal transitions semantically differ from self-transitions. When triggering an internal transition in a state s, the exit- and entry-actions of s are not executed.

8. Functions

$$\begin{aligned} shallowHistory &: Composite \rightarrow ShallowHistory, \\ deepHistory &: Composite \rightarrow DeepHistory, \text{ and} \\ defaultHistory &: Composite \rightarrow History \end{aligned}$$

determine for a given composite state $s \in Composite$ its (potential) shallow, deep, and default history state, respectively. The concept of history states has already been presented in Section 2.3.2.

Let $h \in History$ be a history state of a composite state s, i.e., $h = shallowHistory(s)$ or $h = deepHistory(s)$. We require that there is at most one transition $t \in TR$ with $tr_{src}(t) = h$. This transition leads to the ***default history state*** of s, i.e., $tr_{dst}(t) = defaultHistory(s)$. This default state is entered only when (a) the composite state s is entered via h and (b) the composite state s is entered for the first time.

9. Function $init : Composite \rightarrow Init$ gives for each composite state the unique initial (pseudo) state. For all $s \in Init$, there is no $tr \in TR$ with $tr_{dst}(tr) = s$, i.e., initial states do not have incoming transitions. Moreover, there is exactly one transition $tr \in TR$ with $tr_{src}(tr) = s$, i.e., each initial state has exactly one outgoing transition that leads to a corresponding proper state.

10. Function $final : Composite \rightarrow Final$ gives for each composite state the unique final state. There is no transition $tr \in TR$ with $tr_{src}(tr) \in Final$, i.e., final states do not have outgoing transitions.

11. $substates : Composite \rightarrow \mathcal{P}(S)$ gives all substates of a state, such that

(a) there is a unique state $top \in Composite$ with
$\forall s \in Composite : top \notin substates(s)$,

(b) $\forall s \in And : substates(s) \subseteq Composite$,[4]

(c) $\forall s \in Composite \setminus \{top\}$ there is exactly one path

$$\langle s_1, \ldots, s_n \rangle \in \underbrace{Composite \times \ldots \times Composite}_{n\ times,\ n \geq 2},$$

with $s_1 = top \ \wedge \ s_n = s \ \wedge \ s_{i+1} \in substates(s_i)$ for $1 \leq i \leq n - 1$.

12. Functions $entry, doActivity, exit : Proper \rightarrow ACTS$ give the actions to take when a state is entered, active, or left, respectively.

13. $deferrableEvents : Proper \rightarrow \mathcal{P}(EVTS)$ gives the set of events to be retained for later consumption.

Example. The AGV State Diagram SC_{AGV} shown in Figure 2.4 on page 28 is formally expressed as follows.

$$\begin{aligned}
Init_{AGV} &= \{\ init_1, init_2\ \},\\
And_{AGV} &= \{\ AGV\ \},\\
Xor_{AGV} &= \{\ Negotiator, Transport\ \},\\
Simple_{AGV} &= \{\ WaitingForOrder, ComputingBid,\\
&\quad WaitingForAcknowledgement, Idle, MovingToLoad, Loading,\\
&\quad MovingToUnload, Unloading, MovingToVacate\ \},\\
VARS_{AGV} &= \{\ currentItem, s1, s2\ \},\\
EVTS_{AGV} &= \{\ s1.\hat{}requestTransport(i), s2.\hat{}acceptBid(i), s2.\hat{}rejectBid(i),\\
&\quad s2.\hat{}requestTransport(i), agv.vacate(p), when(order = true)\ \},\\
GUARDS_{AGV} &= \{\ order = true, order = false, s2 = s1\ and\ i = currentItem,\\
&\quad p = self.pos\ \},\\
ACTS_{AGV} &= \{\ order := true, order := false, currentItem := i, pos := dest,\\
&\quad dest := getInputPos(s1), dest := self.getInputPos(s1),\\
&\quad dest := getParkPos(), dest := currentItem.nextDest(pos),\\
&\quad send\ s1.rejectRequest(i), send\ s1.bidding(currentItem, bid),\\
&\quad send\ s2.rejectRequest(i), computeBid(dest),\\
&\quad move(dest), load(currentItem), unload(currentItem)\ \},
\end{aligned}$$

[4]This is a well-formedness rule of the UML standard (see [OMG03d, Section 2.12.3.1]). In many alternative formal syntax definitions, even $s' \in Xor$ is required in this case, leading to a *normal form* of alternating Xor- and And-states in the state hierarchy.

The rich set TR_{AGV} of transitions is not listed for brevity reasons. To give an example, we here only provide the self-transition of state $WaitingForOrder$: The remaining transitions can easily be obtained from the State Diagram in Figure 2.4.

$$WaitingForOrder \xrightarrow{s1.requestTransport(i)[order=true]/send\ s1.rejectRequest(i)} WaitingForOrder.$$

For the states in S_{AGV}, the corresponding initial states and substates are defined as follows. Note that in this special case final states do not appear, as we assume the ideal case that the (physical) AGVs do not break down and their corresponding objects will never be destroyed.

$$\begin{aligned}
init_{AGV}(init_1) &= WaitingForOrder,\\
init_{AGV}(init_2) &= Idle,\\
substates_{AGV}(AGV) &= \{\ Negotiator, Transport\ \},\\
substates_{AGV}(Negotiator) &= \{\ WaitingForOrder, ComputingBid,\\
&\quad\ WaitingForAcknowledgement\ \},\\
substates_{AGV}(Transport) &= \{\ Idle, MovingToLoad, Loading, Unloading,\\
&\quad\ MovingToUnload, MovingToVacate\ \},
\end{aligned}$$

Finally, the entry, exit, and do-activities are defined.

$$\begin{aligned}
entry_{AGV}(ComputingBid) &= 'currentItem := i',\\
exit_{AGV}(MovingToLoad) &= 'pos := dest',\\
exit_{AGV}(Loading) &= 'dest := currentItem.nextDest(pos)',\\
exit_{AGV}(MovingToUnload) &= 'pos := dest',\\
exit_{AGV}(Loading) &= 'order := false',\\
exit_{AGV}(MovingToVacate) &= 'pos := dest',\\
doActivity_{AGV}(computingBid) &= 'computeBid(dest)',\\
doActivity_{AGV}(MovingToLoad) &= 'move(dest)',\\
doActivity_{AGV}(Loading) &= 'load(currentItem)',\\
doActivity_{AGV}(MovingToUnload) &= 'move(dest)',\\
doActivity_{AGV}(Unloading) &= 'unload(currentItem)',\\
doActivity_{AGV}(MovingToVacate) &= 'move(dest)',
\end{aligned}$$

All other components of SC_{AGV} are set to $\varnothing$.

Some Remarks on Definition 4.5. The set $Final$ of final states resembles a pseudo state a lot, e.g., a final state cannot have an entry action, activity, exit action, internal transitions, and deferrable events. Therefore, one might argue that final states belong to the set of pseudo states. But there is one significant semantic difference: Pseudo states are transient nodes, i.e., they are never part of an active state configuration. Final states, however, may appear in an active state configuration (see Definition 4.12 on page 103), even across an indefinite number of execution steps (i.e., run-to-completion steps, RTC-steps). This justifies to keep final states separated from pseudo states.[5]

Definition 4.5 covers most of the abstract State Diagram syntax of the official UML 1.5 specification [OMG03d, Section 2.12.2]. There are only a few details left out (e.g., the bound of synch states and some more syntactical restrictions [OMG03d, Section 2.12.3]), which can easily be added to the definition if necessary. But for our purposes, it is sufficient to regard the State Diagram components defined above.

There are several semantic issues arising when State Diagrams have to be considered along the generalization hierarchy of classes (see Section 4.1.5).

UML State Diagrams do not have an inherent explicit time model, although it is possible to syntactically specify time-related events. For example, a timeout that requires to leave a state after 10 seconds can be specified by an event `after(10 sec)` attached to a transition. Later, in Chapter 5, we formally define syntax and semantics of a *timed State Diagram variant*.

4.1.4 Associations

Associations are used to model structural relationships between classes. Though generally associations may connect an arbitrary number of classes, most frequently binary associations are applied. Moreover, there is no restriction on the number of associations a class may participate in.

Definition 4.6 *(Associations)*
The set ASSOC of associations is defined by

- *a finite set of names* $ASSOC \subseteq \mathcal{N}$,
- *a function* $associates : \begin{cases} ASSOC \to CLASS^{+} \\ as \mapsto \langle c_1, \ldots, c_n \rangle \text{ with } n \geq 2. \end{cases}$

Function $associates$ gives for each association $as \in ASSOC$ a tuple $\langle c_1, \ldots, c_n \rangle$ that represents the classes that participate in the association. Note that the elements of $\langle c_1, \ldots, c_n \rangle$ do not necessarily have to be distinct. In particular, binary associations with $associates(as) = \langle c, c \rangle$, i.e., both association-ends are attached to the same class c, are called self-associations or recursive associations. In general, a class may participate multiple times in a single association. In order to distinguish the role of each association end in such a case, unique role names are applied to be able to uniquely refer to a specific association end when navigating through the model.

Definition 4.7 *(Role Names)*
Let $as \in ASSOC$ *be an association with* $associates(as) = \langle c_1, \ldots, c_n \rangle$. *Role names for an association are defined by a function*

$$roles : \begin{cases} ASSOC \to \mathcal{N}^{+} \\ as \mapsto \langle r_1, \ldots, r_n \rangle \text{ with } n \geq 2, \end{cases}$$

[5]see UML RTF1.4, issue 3201, http://cgi.omg.org/issues/issue3201.txt

where all role names must be distinct, i.e.,

$$\forall i, j \in \{1, \ldots, n\} : i \neq j \implies r_i \neq r_j.$$

Function $roles(as) = \langle r_1, \ldots, r_n \rangle$ assigns each class c_i participating in the association as a unique role name r_i. If no role name is provided for an association end, the respective name of the class is taken by default, with the first letter in lower case. Note that for self-associations unique role names must be provided, as discussed above.

Moreover, an additional syntactical constraint is needed to guarantee unique role names, namely for the case that a class is part of multiple associations. Before we can formally express this constraint, we need to define two help functions, i.e., $participating$ and $navEnds$. First, function $participating$ gives the set of associations a class participates in.

$$participating : \begin{cases} CLASS \to \mathcal{P}(ASSOC) \\ c \mapsto \{\ as \mid\ as \in ASSOC \\ \qquad\qquad\qquad \wedge\ associates(as) = \langle c_1, \ldots, c_n \rangle \\ \qquad\qquad\qquad \wedge\ \exists i \in \{1, \ldots, n\} : c_i = c\}. \end{cases}$$

Second, function $navEnds$ gives the set of all role names that are reachable (or: navigable) from a class along a given association.

$$navEnds : \begin{cases} CLASS \times ASSOC \to \mathcal{P}(\mathcal{N}) \\ (c, as) \mapsto \{\ r \mid\ associates(as) = \langle c_1, \ldots, c_n \rangle \\ \qquad\qquad\qquad \wedge\ roles(as) = \langle r_1, \ldots, r_n \rangle \\ \qquad\qquad\qquad \wedge\ \exists i, j \in \{1, \ldots, n\} : i \neq j \wedge c_i = c \wedge r_j = r\}. \end{cases}$$

Informally speaking, we have to guarantee that navigation ends of the associations a given class c participates in are pairwise distinct. Otherwise, we might not be able to unambiguously navigate along associations and function $navigationEnds$ (see below) cannot correctly be built. We therefore require

$$\begin{gathered} \forall c \in CLASS, \forall as, as' \in participating(c) : \\ as \neq as' \implies navEnds(c, as) \cap navEnds(c, as') = \emptyset. \end{gathered}$$

We can now determine the set of role names that can be directly reached from a given class by navigating along the associations this class participates in by function $navigationEnds$, which is defined by

$$navigationEnds : \begin{cases} CLASS \to \mathcal{P}(\mathcal{N}) \\ c \mapsto \bigcup_{as \in participating(c)} navEnds(c, as). \end{cases}$$

An association specifies the possible existence of connections between objects. A connection between objects is also called a *link* in UML terminology (see Section 4.2.4). *Association multiplicities* specify the number of links that can be established on a given object.

Definition 4.8 *(Association Multiplicities)*
Let $as \in ASSOC$ be an association with $associates(as) = \langle c_1, \dots, c_n \rangle$. Function

$$multiplicities(as) = \langle M_1, \dots, M_n \rangle$$

assigns each class c_i participating in the association, $1 \leq i \leq n$, a non-empty set $M_i \subseteq \mathbb{N}_0$ with $M_i \neq \{0\}$.

For example, we require that an item is always associated with a factory unit. This is indicated in Figure 2.2 on page 21 by a number 1 attached to the association `is-at-unit` at the `FactoryUnit` end.

Aggregation and composition are special forms of associations representing part-whole relationships among classes. They are denoted by hollow and filled diamonds in the UML Class Diagram notation. Based on the observation that aggregations and compositions can be mapped to simple associations and additional OCL constraints [GR99], it is sufficient to regard only simple associations in this formal model.

4.1.5 Generalization

Generalization and specialization are taxonomic relationships between classes. Generalization refers to the bottom-up approach of setting up a more general class from one or more existing subclasses. Common features are adopted in the general class, while specific differences are restrained. By specialization, we refer to a relationship between classes, in which a general class is specialized in a top-down manner into one or more subclasses. Subclasses inherit features of their superclasses (e.g., attributes and operations).

Basically, specialization and generalization are different views of the same concept, and we will mainly use the term *generalization* in the following to refer to this concept. Thus, we may say that in a generalization relationship of *two* classes, we have a more general class (the parent) and a more specific class (the child) that is consistent with the parent and carries some additional information. Note that the notion of generalization is not only known for classes in UML. For example, generalization relationships are also applied to signals, packages, and use cases.

Definition 4.9 *(Generalization Hierarchy, Child and Parent Classes)*
A generalization hierarchy $\prec$ is an irreflexive partial order on $CLASS$, i.e., $\prec$ is an irreflexive, anti-symmetric, and transitive relation. Pairs in $\prec$ describe generalization relationships between two classes.

For $c_1, c_2 \in CLASS$ with $c_1 \prec c_2$, c_1 is called a child class of c_2, and c_2 is called a parent class of c_1.

A child class transitively inherits characteristics (i.e., attributes, operations, signals, and associations) of its parent classes.

Correspondingly, the generalization hierarchy $\prec_{SIG}$ defines an irreflexive partial order on SIG. As a signal can be specified as the child of another signal, reception of that child signal may also trigger any transition in a State Diagram that depends on any of its ancestor signals.

The set of characteristics defined for a class together with its inherited characteristics is called a *full descriptor of a class*. Before formalizing this issue, we define a function for collecting all transitive parents of a given class.

$$parents : \begin{cases} CLASS \to \mathcal{P}(CLASS) \\ c \mapsto \{c' \mid c' \in CLASS \wedge c \prec c'\}. \end{cases}$$

The complete set of attributes of c is the set ATT_c^* that contains all inherited and direct attributes.

$$ATT_c^* \overset{def}{=} ATT_c \cup \bigcup_{c' \in parents(c)} ATT_{c'}.$$

The complete set of user-defined operations is determined analogously.

$$OP_c^* \overset{def}{=} OP_c \cup \bigcup_{c' \in parents(c)} OP_{c'}.$$

The complete set of user-defined signals is given by

$$SIG_c^* \overset{def}{=} SIG_c \cup \bigcup_{c' \in parents(c)} SIG_{c'}.$$

Finally, the complete set of navigable role names for a class $c \in CLASS$ is given as follows.

$$navigationEnds^*(c) \overset{def}{=} navigationEnds(c) \cup \bigcup_{c' \in parents(c)} navigationEnds(c').$$

Definition 4.10 *(Full Descriptor of a Class)*
The full descriptor of a class $c \in CLASS$ is a tuple

$$FD_c \overset{def}{=} \langle ATT_c^*, OP_c^*, SIG_c^*, SC_c, navigationEnds^*(c) \rangle$$

containing all attributes, user-defined operations, signals, navigable role names, and the possibly associated State Diagram.

The UML standard requires that certain characteristics of a full descriptor must be distinct. For example, a class may not define an attribute that is already defined in one of its parent classes. These constraints are captured more precisely by the following well-formedness rules. Each constraint must hold for each $c \in CLASS$.

1. Attributes are defined in exactly one class.

$$\begin{gathered} \forall \langle a, t_c, t \rangle, \langle a', t_{c'}, t' \rangle \in ATT_c^* : \\ a = a' \implies t_c = t_{c'} \wedge t = t' \end{gathered}$$

2. An operation may only be defined once in a full class descriptor. The first parameter of an operation signature indicates the class in which the operation is defined. The following condition guarantees that each operation in a full class descriptor is defined in a single class.

$$\begin{gathered}\forall(\omega : t_c \times t_1 \times \ldots \times t_n \to t), (\omega' : t_{c'} \times t'_1 \times \ldots \times t'_n \to t') \in OP^*_c : \\ \omega = \omega' \wedge t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n \implies t_c = t_{c'}\end{gathered}$$

3. A signal may only be defined once in a full class descriptor. The first parameter of a signal signature indicates the class in which the signal is defined. The following condition guarantees that each signal in a full class descriptor is defined in a single class.

$$\begin{gathered}\forall(\omega : t_c \times t_1 \times \ldots \times t_n), (\omega' : t_{c'} \times t'_1 \times \ldots \times t'_n) \in SIG^*_c : \\ \omega = \omega' \wedge t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n \implies t_c = t_{c'}\end{gathered}$$

4. Role names are defined in exactly one class among the generalization hierarchy of a given class c.

$$\begin{gathered}\forall c_1, c_2 \in parents(c) \cup \{c\} : \\ c_1 \neq c_2 \implies navigationEnds(c_1) \cap navigationEnds(c_2) = \varnothing\end{gathered}$$

5. We have seen in Section 2.3.3.1 that OCL uses the same notation to refer to operations and signals to specify OCL messages. To uniquely identify an operation or signal, the operation and signal names of a class (in combination with the corresponding parameters) must be pairwise distinct.

$$\begin{gathered}\forall(\omega : t_c \times t_1 \times \ldots \times t_n \to t) \in ATT^*_c, \\ \forall(\omega' : t_{c'} \times t_1 \times \ldots \times t_n) \in SIG^*_c : \quad \omega \neq \omega'\end{gathered}$$

   Note that for ω', the types $t_1, \ldots, t_n$ are fixed by the parameter types of ω.

6. Similarly, OCL uses the same notation for accessing attributes and navigating by role name. Therefore, attribute names and role names must be pairwise distinct.

$$\forall \langle a, t_c, t \rangle \in ATT^*_c, \forall r \in navigationEnds^*(c) : \ a \neq r$$

Note that it is allowed for operations and signals to have the same name as attributes or role names, because the concrete syntax of OCL allows to distinguish between these cases.

4.2 Semantics

In the previous section, the syntax of extended object models has been defined. In this section, we now present a formal semantics of extended object models.

4.2.1 Objects

The domain of a class $c \in CLASS$ is the set of objects of this class and all of its child classes. Objects are referred to by object identifiers that are unique in the context of the whole executed model. In the remainder, no distinction will be made between objects and their identifiers, i.e., each object is uniquely determined by its identifier and vice versa.

Definition 4.11 *(Object Identifiers and Domain of a Class)*
The set of object identifiers of a class $c \in CLASS$ is defined by an infinite set

$$oid(c) \overset{def}{=} \{\underline{objId}_1, \underline{objId}_2, \ldots\}.$$

The domain of a class $c \in CLASS$ is defined as

$$I_{CLASS}(c) \overset{def}{=} \bigcup_{c' \in CLASS \text{ with } c' \prec c \,\vee\, c'=c} oid(c').$$

Correspondingly, the domain of type $t_c \in T_C$ that is induced by class c, is

$$I_{TYPE}(t_c) \overset{def}{=} I_{CLASS}(c) \cup \{\perp\}.$$

For brevity reasons, we omit the index of $I_{CLASS}(c)$ and $I_{TYPE}(t)$ when the context is clear.

4.2.2 A Note about State Diagram Inheritance

The problem of consistency among generalization of classes and inheritance of characteristics (i.e., attributes and operations) has been studied extensively for object-oriented languages, but *consistency among inheritance of behavior* in conceptual object-oriented design notations like UML has received less attention. Different notions for consistency of behavior have been identified in this context [EE94, SS00, SS02a]. Their definition makes use of the dynamic execution of State Diagrams by traces, which are execution runs through (the processes derived from) State Diagrams.

First, *weak invocation consistency* guarantees that each trace of the State Diagram for the superclass is also contained in the set of traces of the State Diagram for the subclass. With other words, a sequence of activities performable on instances of a superclass can also be performed on instances of a subclass. Second, *strong invocation consistency* guarantees the latter property even if activities added to the subclass have been inserted arbitrarily in that sequence. Finally, in *observation consistency*, the State Diagram of the superclass specifies an upper bound to the behavior of the subclasses. It is guaranteed that every trace of an instance of a subclass is observable as a trace of the superclass, when states, events, and activities added at the subclass are neglected.

Apart from that classification, UML 1.5 provides an informal description of three different inheritance policies for state machines [OMG03d, Section 2.12.5.3], which implicitly applies as well to State Diagrams: subtyping, strict inheritance, and general refinement, where ’refinement’ in this case is a synonym for inheritance.

Subtyping requires that a state in the subclass retains all its transitions. Transitions may lead to the same state or a new substate of that state (i.e., strengthening of the transition post-condition), and guard conditions may be weakened by adding disjunctions (i.e., weakening of transition preconditions). This corresponds to *weak invocation consistency* and complies to the substitutability principle.

The other two policies provided by UML 1.5 support neither observation nor invocation consistency and are instead oriented towards coding and inheritance issues. *Strict inheritance* is intended to encourage reuse of implementation rather than preserving behavior. This kind of policy results from the fact that in many programming languages features cannot be deleted in subclasses once defined in a superclass. Thus, it is not allowed to remove outgoing transitions in subclasses and to apply a different source state to an existing transition. Nevertheless, new states and transitions can be added, and guard conditions, transition target states, and incoming transitions may be altered without any further restrictions. Finally, *general refinement* basically places no restrictions on State Diagram inheritance.

Note that if a class c has multiple superclasses, the default State Diagram for c consists of all the State Diagrams of its superclasses as orthogonal regions. This may be overridden through a kind of State Diagram inheritance if required.

4.2.3 State Configurations

As a result from the discussion in the previous section we assume in the following that an extended object model $\mathcal{M}$ under consideration complies to a predefined policy of State Diagram inheritance. This means that for each active class $c \in ACTIVE$ there is a State Diagram specification SC_c which is consistent with the State Diagrams of the superclasses of c.

In a State Diagram with composite and concurrent states, the term 'current state' cannot be applied without causing confusion, as more than one state can be active at the same time. Consequently, UML 1.5 provides the notion of *active state configurations* [OMG03d, Section 2.12.4.3].

If the State Diagram is in a simple state that is contained in a composite state, then all the composite states that (transitively) contain the simple state are also active. Furthermore, as composite states in the state hierarchy may be concurrent, the currently active states are actually represented by a tree of states starting with the single state top_c at the root down to individual simple states $s_i \in Simple_c$ at the leaves. Such a state tree is in UML 1.5 referred to as a state configuration. In the following definition 4.12, we give a corresponding formal definition of state configurations. But first, we define a convenience function $superstate_c$ that gives the direct superstate of a state $s \in S_c$:

$$superstate_c : \begin{cases} S_c \to Composite_c \\ s \mapsto \begin{cases} s', & \text{if } \exists s' \in Composite_c \text{ with } s \in substates_c(s') \\ \varnothing, & else. \end{cases} \end{cases}$$

UML 1.5 does not consider final states in state configurations. In contrast, we include final states in the following definition for state configurations, as they might be active after an RTC-step. However, a final state that is a direct child state of top_c is not part of any configuration, since entering that state is equivalent to termination (or: destruction) of the corresponding object. Additionally, we explicitly exclude *immediate states*. Immediate states are proper states that are directly run through in an RTC-step, as they do not have outgoing transitions that have to wait for a triggering event. Consequently, they can never be part of an active state configuration after completion of an RTC-step. We here leave out a formal definition and simply refer to set $Immediate_c$ to denote the set of all immediate proper states of a State Diagram SC_c.

Furthermore, we make use of the following help sets for classes $c \in ACTIVE$:

$$\begin{aligned} ProperStay_c &\overset{def}{=} Proper_c \setminus Immediate_c, \\ Stay_c &\overset{def}{=} ProperStay_c \cup \{f \in Final_c \mid f \notin substates_c(top_c)\}, \\ Basic_c &\overset{def}{=} (Simple_c \setminus Immediate_c) \cup \{f \in Final_c \mid f \notin substates_c(top_c)\}. \end{aligned}$$

Definition 4.12 *(State Configurations with respect to state s)*
Let $c \in ACTIVE$ and SC_c be the State Diagram for c. A state configuration $\mathcal{C}$ with respect to a state s *is a maximal set of states that the State Diagram can be simultaneously in, taking state s as the root. Function cfg_c that maps a state $s \in ProperStay_c$ to the set of configurations $\mathcal{C}$ with respect to s is defined by*

$$cfg_c : \begin{cases} ProperStay_c \rightarrow \mathcal{P}(\mathcal{P}(Stay_c)) \\ s \mapsto \{\mathcal{C} \in \mathcal{P}(Stay_c) \mid s \in \mathcal{C} \\ \qquad \wedge \forall s' \in \mathcal{C} \cap And_c : substates_c(s') \subseteq \mathcal{C} \\ \qquad \wedge \forall s' \in \mathcal{C} \cap Xor_c : |substates_c(s') \cap \mathcal{C}| = 1 \\ \qquad \wedge \forall s' \in \mathcal{C} \setminus \{s\} : superstate_c(s') \in \mathcal{C} \;\}. \end{cases}$$

Definition 4.13 *(State Configuration)*
The set $I_{SC}(c)$ of overall state configurations *for a class $c \in ACTIVE$, which are state configurations with respect to the top state top_c, is determined by $cfg_c(top_c)$.*
For convenience, we define $I_{SC}(c)$ for all $c \in CLASS$ by

$$I_{SC}(c) \overset{def}{=} \begin{cases} cfg_c(top_c), & \text{if } c \in ACTIVE, \\ \varnothing, & \text{if } c \in PASSIVE. \end{cases}$$

By definition, each state configuration induces a state tree. But to uniquely determine a state configuration, it is sufficient to have information about terminal states, i.e., the simple and final states.

Definition 4.14 *(Basic State Configurations)*
Let $c \in ACTIVE$ and SC_c be the State Diagram for c. Let $s \in ProperStay_c$ be a state and let $\mathcal{C} \in cfg_c(s)$ be a state configuration with respect to s. The set

$$B_{\mathcal{C}} \overset{def}{=} \mathcal{C} \cap Basic_c$$

is called a basic state configuration *(with respect to $\mathcal{C}$). The set B_s of all basic configurations with respect to s is then defined by*

$$B_s \overset{def}{=} \{B_{\mathcal{C}} \mid \mathcal{C} \in cfg_c(s)\} \subseteq \mathcal{P}(\mathcal{P}(Basic_c)) .$$

Note that the following condition holds (cf. [PU97, Lemma 1]):

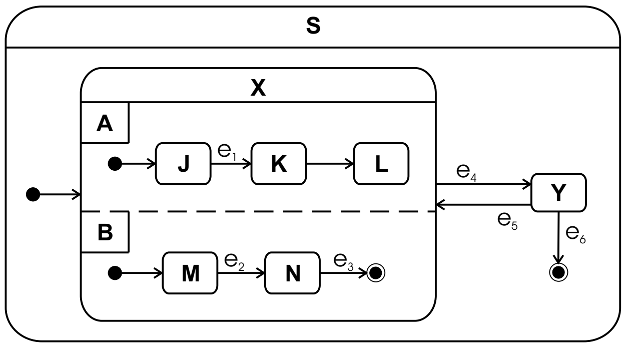
$$\forall s \in ProperStay_c, \forall B_{\mathcal{C}} \in B_s :$$
$$superstate^*(B_{\mathcal{C}}) \cap substates^*(s) = \mathcal{C} \ .$$

In other words, given a basic state configuration $B_{\mathcal{C}}$, we can uniquely determine the state configuration $\mathcal{C} = cfg_c(s)$ with respect to a state s.

We here employed function

$$superstate^* : \mathcal{P}(S_c) \rightarrow \mathcal{P}(ProperStay_c).$$

Basically, that function gives the set of transitive superstates on a given set of states (including that given set of states). Function $substates^* : ProperStay_c \rightarrow \mathcal{P}(Stay_c)$ in turn gives the set of transitive substates on a given state (including this state).



```
Proper States:        { S,X,Y,A,B,J,K,L,M,N }
Final States:         { S::FinalState,B::FinalState }
Immediate States:     { K }
State Configurations: { {S,X,A,B,J,M},{S,X,A,B,J,N},
                        {S,X,A,B,J,B::FinalState},
                        {S,X,A,B,L,M},{S,X,A,B,L,N},
                        {S,X,A,B,L,B::FinalState},
                        {S,Y} }
Basic Configurations: { {J,M},{J,N},{J,B::FinalState},
                        {L,M},{L,N},{L,B::FinalState},
                        {Y} }
```

Figure 4.1: State Diagram Example

Figure 4.1 gives a State Diagram example with corresponding basic state configurations. All proper states except immediate state `K` have an outgoing transition with a specified event e_i, $1 \leq i \leq 6$. As UML does not provide a textual equivalent for final states, we use the parent state name, double colons, and the keyword `FinalState` to syntactically refer to final states. Note that `S::FinalState` is not part of the configuration set (cf. usage of set $Stay_c$ in Definition 4.12).

4.2.4 Links

An association describes possible connections between objects (i.e., links) of the classes participating in the association. Semantically, an association is a relation that describes the set of all possible connections between objects of the associated classes (more precisely, classes and their children).

Definition 4.15 *(Links)*
Each association $as \in ASSOC$ *with* $associates(as) = \langle c_1, \ldots, c_n \rangle$ *is interpreted as the product of the sets of object identifiers of the participating classes:*

$$I_{ASSOC}(as) \overset{def}{=} I_{CLASS}(c_1) \times \ldots \times I_{CLASS}(c_n).$$

A link is an element $l_{as} \in I_{ASSOC}(as)$.

4.2.5 System State

In the following, we call a particular instantiation of an extended object model a *system*. A system is in different states as it changes over time, i.e., the (number of) objects, their attribute values, State Diagram configurations, and other characteristics change when actually executing the system. But it still has to be defined what a single system state exactly consists of. It is important to point out here that different notions of a system state are generally possible, depending on the scope of model analysis one wants to perform. In the original work on object models [Ric01], a system state is a tuple consisting of three parts:

- the current set of objects,
- their attribute values, and
- the current links that connect the objects.

A semantics of a large part of standard OCL expressions is defined over such systems states in [Ric01, Sect. 5.2]. However, as State Diagrams are not considered in that work, state-related operations such as `oclInState(statename:OclState)` could not be handled so far.

In our approach, we additionally investigate *sequences* of system states, i.e., we are going to perform an analysis over possible future system states and thus reason about evolution of State Diagram states. For this, we need a concise notion of *system state sequences* that also covers State Diagram configurations. In order to be able to formally define such sequences, we need to define which operations are to be executed next (for operation preconditions) and which operations terminate later (for operation postconditions). In this context, we adopt ideas of [ZG02, ZG03] to formalize currently executed operations and define additional functions to capture that information.

Definition 4.16 *(System State)*
A system state for an extended object model $\mathcal{M}$ *is a tuple*

$$\sigma(\mathcal{M}) \overset{def}{=} \langle \Sigma_{CLASS}, \Sigma_{ATT}, \Sigma_{ASSOC}, \Sigma_{CONF}, \Sigma_{currentOp}, \Sigma_{currentOpParam} \rangle$$

where

1. $\Sigma_{CLASS} \overset{def}{=} \bigcup_{c \in CLASS} \Sigma_{CLASS,c}$.
   The finite sets $\Sigma_{CLASS,c}$ contain all objects of a class $c \in CLASS$ existing in the system state, i.e.,
   $$\Sigma_{CLASS,c} \subseteq oid(c) \subseteq I_{CLASS}(c).$$

For further application, we define $\Sigma_{ACTIVE,c}$ for active and $\Sigma_{PASSIVE,c}$ for passive classes correspondingly.

2. The current attribute values are kept in set Σ_{ATT}. It is the union of functions $\sigma_{ATT,a} : \Sigma_{CLASS,c} \rightarrow I(t)$, where $a \in ATT^*_c$. Each function $\sigma_{ATT,a}$ assigns a value to a certain attribute of each object of a given class $c \in CLASS$.

3. $\Sigma_{ASSOC} \overset{def}{=} \bigcup_{as \in ASSOC} \Sigma_{ASSOC,as}$ comprises the finite sets $\Sigma_{ASSOC,as}$ that contain links that connect objects, where

$$\forall as \in ASSOC : \Sigma_{ASSOC,as} \subseteq I_{ASSOC}(as).$$

We refer to [Ric01] for detailed information about links, i.e., elements of $I_{ASSOC}(as)$, and formalization of multiplicity specifications.

4. The current State Diagram configurations are kept by

$$\sigma_{CONF} \overset{def}{=} \bigcup_{c \in ACTIVE} \{\sigma_{CONF,c} : \Sigma_{ACTIVE,c} \rightarrow I_{SC}(c)\}.$$

Each function $\sigma_{CONF,c}$ assigns a state configuration with respect to the corresponding top state top_c to each object of a given class $c \in ACTIVE$.

5. Let $\mathcal{ID}$ be an infinite enumerable set, e.g., $\mathcal{ID} = \mathbb{N}$. The set of currently executed operations is denoted by

$$\Sigma_{currentOp} \overset{def}{=} \bigcup_{c \in CLASS} \{\sigma_{currentOp,c} : \Sigma_{CLASS,c} \times OP^*_c \rightarrow \mathcal{P}(\mathcal{ID})\}.$$

Each function $\sigma_{currentOp,c}$ gives a set of unique identifiers $\in \mathcal{ID}$ that represents all currently executed operations for a given object $\underline{oid}$ and operation signature op. At the starting point of an operation execution, a unique identifier $\in \mathcal{ID}$ is associated with that operation execution. We require that the associated identifier must not change until the execution of that operation terminates.

6. $\Sigma_{currentOpParam} \overset{def}{=}$

$$\bigcup_{c \in CLASS} \{\sigma_{currentOpParam,c} : \Sigma_{CLASS,c} \times OP^*_c \times \mathcal{ID} \rightarrow I(t_1) \times \ldots \times I(t_n) \times I(t)\ \}$$

is a set of functions that gives the parameter values of each of the currently executed operations. For each $c \in CLASS$, we define $\sigma_{currentOpParam,c}$ as follows, where $op = (\omega : t_c \times t_1 \times \ldots \times t_n \rightarrow t) \in OP_c$:

$$\sigma_{currentOpParam,c}(\underline{oid}, op, id) \mapsto \begin{cases} \langle val_1, \ldots, val_n, returnVal \rangle, & \text{if } id \in \sigma_{currentOp,c}(\underline{oid}, op) \\ \varnothing, & \text{otherwise.} \end{cases}$$

In the definition above, $val_j \in I(t_j)$ denotes an arbitrary value defined for type $t_j \in T$, $1 \leq j \leq n$. The same holds for $returnVal \in I(t)$. If an operation is not returning a result, the result type t of operation op is `OclVoid`. In that case, we set $returnVal = \perp$.

Of course there are additional State Diagram characteristics that could also be taken into account to be part of a system state, e.g., event queues and changes occurring to them or additional information required for re-entering composite states via history states. However, while this can make sense in some specific approaches, the definition above is sufficient for reasoning about currently activated states and executed operations.

4.2.6 Semantics of Operation oclInState(statename:OclState)

The notion of a system state with integrated active State Diagram state configurations enables us to define a semantics of operation `oclInState(statename:OclState)`. This issue is still missing in the semantics of the adopted OCL 2.0 specification.

According to the OCL 2.0 specification, the operation signature of `oclInState(statename:OclState)` is defined by

$$oclInState : OclAny \times OclState \rightarrow Boolean,$$

where the domain of $OclState$ is formally defined by

$$I(OclState) = (\bigcup_{c \in ACTIVE} Stay_c) \cup \{\bot\}.$$

For an operation $op = (\omega : t_c \times t_1 \times \ldots \times t_n \rightarrow t) \in OP_c$, a semantics is generally defined by a total function with signature

$$I[[op]] : I(t_c) \times I(t_1) \times \ldots \times I(t_n) \rightarrow I(t),$$

where we implicitly assume a given system state.[6]

Correspondingly, we define the semantics of operation `oclInState(statename:OclState)` on a given system state $\sigma(\mathcal{M})$, a given object $\underline{oid} \in \Sigma_{CLASS,c}$, and a state name $s \in I(OclState)$ by

$$I[[oclInState : OclAny \times OclState \rightarrow Boolean]](\underline{oid}, s) \overset{def}{=}$$
$$\begin{cases} true, & \text{if } \underline{oid} \in \Sigma_{ACTIVE,c} \land s \in Stay_c \\ & \quad \land\, s \in \sigma_{CONF,c}(\underline{oid}), \\ false, & \text{if } \underline{oid} \in \Sigma_{ACTIVE,c} \land s \in Stay_c \\ & \quad \land\, s \notin \sigma_{CONF,c}(\underline{oid}), \\ \bot, & \text{if } \underline{oid} \notin \Sigma_{ACTIVE,c} \\ & \quad \lor\, (\underline{oid} \in \Sigma_{ACTIVE,c} \land s \notin Stay_c \cup \{\bot\}) \\ & \quad \lor\, s = \bot\ . \end{cases}$$

[6]More precisely, the additional variable assignment β has also to be considered. Function β determines values for OCL-specific variables, such as iterator variables and local variables of so-called `let`-expressions [OMG03b, Section A.3.1.2].

Note here that `oclInState(statename:OclState)` returns $\perp$ when $\underline{oid}$ is a passive object or when state s is not element of set $Stay_c$ of an active class $c \in ACTIVE_c$. Alternatively, we could have chosen to return $false$ instead in these cases. Neither the UML 1.5 standard nor the adopted OCL 2.0 specification give any information about this issue.

4.2.7 Traces

So far, it is not defined how a system state is actually built. The OCL 2.0 semantics simply assumes that a system state $\langle \Sigma_{CLASS}, \Sigma_{ATT}, \Sigma_{ASSOC} \rangle$ is given by a set Σ_{CLASS} of currently existing objects, a set Σ_{ATT} of attribute values for the objects, and a set Σ_{ASSOC} of currently established links that connect the objects. While this structure is easy to obtain from a concrete (implementation of a) running system, the situation becomes more complicated when also State Diagram states are considered.

We therefore have to define *traces*, i.e., sequences of system states that keep track of all 'noteworthy changes' within a running system. In the context of checking OCL constraints, we are, for instance, not interested in every single attribute value change that occurs during execution of an operation. Instead, we are interested in those system states in which something happens that is of relevance for evaluation of OCL expressions, e.g., when an operation has been completed and a corresponding postcondition should be checked.

In the simplest case, e.g., when (an implementation of) the system is executed on a single CPU, there is a clear temporal order on the system execution. But when (the implementation of) the system is distributed, we have a partial order among the system executions. This problem can be treated in an ideal case by introducing a *global clock* that allows for a global view on the system. We here follow the idea of a global view on the system.

Definition 4.17 *(Trace)*
A well-defined system state sequence called trace *for an instantiation of an extended object model $\mathcal{M}$ is an (infinite) sequence of system states,*

$$trace(\mathcal{M}) \overset{def}{=} \langle\langle\, \sigma(\mathcal{M})_{[0]}, \sigma(\mathcal{M})_{[1]}, \ldots, \sigma(\mathcal{M})_{[i]}, \ldots \rangle\rangle.$$

The first trace element $\sigma(\mathcal{M})_{[0]}$ denotes the initial system state. Given a system state $\sigma(\mathcal{M})_{[i]}$, $i \in \mathbb{N}_0$, the next system state $\sigma(\mathcal{M})_{[i+1]}$ is added to the trace when for at least one object a noteworthy change *occurs.*

Noteworthy Changes. Let $inv(c)$ denote the set of invariants of a class $c \in CLASS$. Let $inv^*(c)$ denote the full set of invariants for a class c, i.e.,

$$inv^*(c) = inv(c) \;\cup \bigcup_{c' \in parents(c)} inv(c').$$

Similarly, let $pre(op, c)$ and $post(op, c)$ denote the pre- and postconditions of an operation $op \in OP_c^*$. Recall that we assume that there is an *inheritance policy* for State Diagrams that guarantees that state-related OCL expressions are well-defined over inheritance relationships among active classes with associated State Diagrams.

We identified the following kinds of *noteworthy changes* relevant for evaluation of OCL expressions. In each case, we give a corresponding rule for updating the current system state $\sigma(\mathcal{M})_{[i]}$ and indicate whether OCL constraints have to be checked. Note that different kinds of noteworthy changes might occur in parallel at the same instant of time, such that more than one rule might have to be applied on a given system state $\sigma(\mathcal{M})_{[i]}$. For example, a number of objects can be created at the same time on different nodes in a distributed system, and in addition one or more links can be established.

Although we abstract from an explicit notion of time here, we have to assume a global view on the system to determine the set of noteworthy changes on the whole (in particular distributed) system at each instant of time.

In the following rules, we are using the $[i]$-annotation also for the components and functions defined in $\sigma(\mathcal{M})_{[i]}$, $i \in \mathbb{N}_0$.

1. Let $\underline{oid}_1, \ldots, \underline{oid}_n$ be the **objects** of classes $c_j \in CLASS$, $1 \leq j \leq n$, that are newly **created**.

$$\begin{aligned} &\forall j \in \{1, \ldots, n\} : \\ &\Sigma_{CLASS,c_j[i+1]} = \Sigma_{CLASS,c_j[i]} \cup \{\underline{oid}_j\} \end{aligned}$$

   Task: Check invariants $inv^*(c_j)$ for all objects $\underline{oid}_j$ on system state $\sigma(\mathcal{M})_{[i+1]}$.

2. Let $\underline{oid}_1, \ldots, \underline{oid}_m$ be the **objects** of classes $c_j \in CLASS$, $1 \leq j \leq m$, that are **destroyed**.

$$\begin{aligned} &\forall j \in \{1, \ldots, m\} : \\ &\Sigma_{CLASS,c_j[i+1]} = \Sigma_{CLASS,c_j[i]} \setminus \{\underline{oid}_j\} \end{aligned}$$

3. Let l_{as_j}, $1 \leq j \leq r$, be the **links** of associations $as_j \in ASSOC$ that are newly **established**.

$$\begin{aligned} &\forall j \in \{1, \ldots, r\} : \\ &\Sigma_{ASSOC,as_j[i+1]} = \Sigma_{ASSOC,as_j[i]} \cup \{l_{as_j}\} \end{aligned}$$

4. Let l_{as_j}, $1 \leq j \leq p$, be the **links** for associations $as_j \in ASSOC$ that are **removed**.

$$\begin{aligned} &\forall j \in \{1, \ldots, p\} : \\ &\Sigma_{ASSOC,as_j[i+1]} = \Sigma_{ASSOC,as_j[i]} \setminus \{l_{as_j}\} \end{aligned}$$

5. Let cfg_j, $1 \leq j \leq q$, be the **new state configurations** that are **reached** for objects $\underline{oid}_j$ of active classes c_j.

$$\begin{aligned} &\forall j \in \{1, \ldots, q\} : \\ &\sigma_{CONF,c_j}(\underline{oid}_j)_{[i+1]} = cfg_j \end{aligned}$$

   Task: Check $inv^*(c_j)$ for object $\underline{oid}_j$ on system state $\sigma(\mathcal{M})_{[i+1]}$.

6. Let $op_j = (\omega_j : t_{c_j} \times t_{j,1} \times ... \times t_{j,n} \to t_j)$, $1 \leq j \leq x$, be the **waiting operation calls** that are **started** to be executed by objects $\underline{oid}_j$ of classes $c_j \in CLASS$.

$$\begin{aligned}&\forall j \in \{1, \ldots, x\} :\\ &\quad \sigma_{currentOp,c_j}(\underline{oid}_j, op_j)_{[i+1]} =\\ &\qquad \sigma_{currentOp,c_j}(\underline{oid}_j, op_j)_{[i]} \cup \{newId_j\}\\ &\qquad \text{where } newId_j \in \mathcal{ID} \text{ is a unique identifier for } op_j\\ &\quad \text{and}\\ &\quad \sigma_{currentOpParam,c_j}(\underline{oid}_j, op_j, newId_j)_{[i+1]} = \langle v_{j,1}, \ldots, v_{j,n_j}, returnVal_j\rangle,\\ &\quad \text{where } \forall k \in \{1, \ldots, n_j\} :\ v_{j,k} \in I(t_k)\\ &\qquad\qquad\qquad \wedge\ \big(\, paramKind(c_j, op_j, k) = out\ \Rightarrow\ v_{j,k} = \bot\big)\\ &\quad \text{and } returnVal_j = \bot\ .\end{aligned}$$

We require that the tuples $\langle v_{j,1}, \ldots, v_{j,n_j}, returnVal_j\rangle$ remain unchanged until the corresponding operation execution terminates. By default, we set $returnVal_j = \bot$ to indicate that the return value is currently undetermined. Correspondingly, output parameters are also set to $\bot$. The values of parameters of kind in and $inout$ are determined by the given values of the referred operation call.

Task: For all $j \in \{1, ..., x\}$, check $pre(op_j, c_j)$ of operation op_j with identifier $newId_j$ on system state $\sigma(\mathcal{M})_{[i+1]}$.

7. Let $op_j = (\omega_j : t_{c_j} \times t_{j,1} \times ... \times t_{j,n} \to t_j)$ with $opId_j$, $1 \leq j \leq y$, be the **executing operations** of objects $\underline{oid}_j$ that **terminate**.

Note that it is not the scope of the semantics of OCL to perform updates on the parameter values when an operation terminates, just as it is not the task of OCL to update attribute values. We therefore assume that the system performs the necessary updates on the actual parameter values $v_{j,1}, ..., v_{j,n_j}$ and the return value $returnVal_j$ of the terminating operations identified by $opId_j$. Thus, all parameters of kind in are still unchanged and the parameters of kind $inout$ and out are already updated in system state $\sigma(\mathcal{M})_{[i]}$. We can therefore simply assign the updated parameter values as follows.

$$\begin{aligned}&\forall j \in \{1, \ldots, y\} :\\ &\quad \sigma_{currentOpParam,c_j}(\underline{oid}_j, op_j, opId_j)_{[i+1]} =\\ &\quad \langle v_{j,1}, ..., v_{j,n_j}, returnVal_j\rangle\end{aligned}$$

Furthermore, in the next but one step $i + 2$, the operation identifiers $opId_j$ must be eliminated from $\sigma_{currentOp,c_j}(\underline{oid}_j, op_j)$. The corresponding tuple of parameter values is eliminated in state $i + 2$ as well, as it is no longer needed.

$$\begin{aligned}&\sigma_{currentOp,c_j}(\underline{oid}_j, op_j)_{[i+2]} = \sigma_{currentOp,c_j}(\underline{oid}_j, op_j)_{[i+1]} \setminus \{opId_j\}\\ \text{and }\ &\sigma_{currentOpParam,c_j}(\underline{oid}_j, op_j, opId_j)_{[i+2]} = \varnothing\end{aligned}$$

Task: For all $j \in \{1, ..., y\}$, check $post(op_j, c_j)$ of operation op_j with identifier $opId_j$ on system state $\sigma(\mathcal{M})_{[i+1]}$. For passive objects $\underline{oid}_j$, we here also check the invariants $inv^*(c)$ on system state $\sigma(\mathcal{M})_{[i+1]}$. In contrast, invariants of active objects are only checked after completion of RTC-steps, which is covered by noteworthy change (5).

All components of system state $\sigma(\mathcal{M})_{[i]}$ except set ATT and the ones that are explicitly mentioned above remain unchanged for the subsequent system state $\sigma(\mathcal{M})_{[i+1]}$.

Restrictions on Traces. The following additional restrictions apply to traces:

1. Two subsequent sequence elements may differ in at most one operation call per object. In order to formally define this, we denote the overall number of operation executions for an object $\underline{oid}$ of class c in system state $\sigma(\mathcal{M})_{[i]}$ by

$$\psi(\underline{oid})_{[i]} \overset{def}{=} \sum_{op \in OP_c} |\sigma_{currentOp,c}(\underline{oid}, op)_{[i]}|.$$

   Using this definition, for each pair of adjacent system states $\sigma(\mathcal{M})_{[i]}$ and $\sigma(\mathcal{M})_{[i+1]}$, $i \in \mathbb{N}_0$, in $trace(\mathcal{M})$, it holds:

$$\forall c \in CLASS, \forall \underline{oid} \in \Sigma_{CLASS,c[i]} :$$
$$\underline{oid} \in \Sigma_{CLASS,c[i+1]} \Rightarrow abs(\psi(\underline{oid})_{[i]} - \psi(\underline{oid})_{[i+1]}) \leq 1$$

2. An object must not be destroyed when one of its operations is still executed. In turn, each executed operation occurring in the sequence must eventually be terminated, i.e., for all operation signatures $op \in OP_c$ of an object $\underline{oid}$ of a class c in a system state $\sigma(\mathcal{M})_{[i]}$, $i \in \mathbb{N}_0$, it must hold:

$$\forall execOp \in \sigma_{currentOp,c}(\underline{oid}, op)_{[i]}, \exists j \in \mathbb{N}, j > i :$$
$$\forall i \leq k \leq j : \underline{oid} \in \Sigma_{CLASS,c[k]} \wedge execOp \notin \sigma_{currentOp,c}(\underline{oid}, op)_{[j]}$$

Our definition of a trace neither makes any assumptions about concurrency among objects nor considers explicit time that has passed between two subsequent sequence elements. By documenting system states with the tuple components as defined above, this work can be seen as a general approach to capture those parts of the system runtime information that is necessary to reason about all relevant system states. In particular, a trace consists of a sequence of system states, documenting all situations

- immediately before any operation is executed, and
- immediately after any operation is terminated, and
- immediately after a new State Diagram configuration is reached.

These system states are sufficient to check OCL invariants as well as operation pre- and post-conditions that make use of state-oriented operations.

4.3 Discussion

With the syntax and formal semantics developed in this chapter, the formal semantics of OCL 2.0 is almost complete. Remaining issues not tackled here concern

1. a formalization of OCL messages,
2. a formal definition of global variable definitions within OCL constraints (so-called def-clauses), and
3. formal descriptions of operations on collection type `OrderedSet`.

Based on the extension of object models as presented in this chapter, a formalization of OCL messages with a semantics of message operators and operations has already been published in [FM04]. The necessary steps to integrate OCL messages are as follows.

- System states have to be further extended to keep the sequence of messages sent.
- Type domains $I(t)$ for $t \in T$ have to be extended by a special symbol denoted by ? to represent the *undetermined* or *unspecified* status of a variable. Note that this symbol is different from the String literal '?'.
- Object identifiers have to be used in a different way, as it might be necessary to refer to an object that is no longer existing at the time of postcondition evaluation.

  Assume that during execution of operation `announceOrder(i)` of an output buffer object $buffer$, messages `requestTransport(i)` have been sent to all associated AGV objects. Consider the following postcondition and assume that one of the associated AGV objects, say $\underline{agvObj}$, is for some reason destroyed while operation `announceOrder(i)` is still executing.

  ```
  context OutputBuffer::announceOrder(i:Item)
  post: machine.transporters@pre->forAll(vehicle:AGV |
                                  vehicle^^requestTransport(i:Item)->size() = 1)
  ```

  Then, at the time of evaluating this postcondition (i.e., at a later point of time), the object $\underline{agvObj}$ does no longer exist. But still, we need to have a valid reference to it in order to evaluate the subexpression `vehicle2equestTransport(i:Item)`.

  This is the reason why we have to distinguish between 'real' objects $\underline{agvObj} \in \Sigma_{CLASS,c}$ that are currently existing and object identifiers $agvObj \in oid(c)$ that just refer to a unique value.

The two other issues are quite easy to resolve; operations defined for ordered sets are basically the same as for sequences, and def-clauses can directly be mapped to so-called `OclHelper` variables and operations. `OclHelper` variables and operations, in turn, are stereotyped attributes and operations of classifiers. Such variables and operations can be used in OCL expressions just like common attributes and operations. Thus, it only has to be ensured that no naming conflicts occur, while additional semantic issues do not occur.

One important remaining task is to complete the metamodel-based OCL semantics. First of all, State Diagram states are still not considered at all in the metamodel-based OCL semantics. But also consistency among the two semantics should be reviewed.

4.4 Contributions of the Chapter

This chapter provides the following contributions:

- The formal model of the *object model* for OCL has been extended by several components. In particular, an abstract syntax of UML State Diagrams has been developed.
- Correspondingly, the semantics of object models has been extended. We provided a formal notion of state configurations that overcomes the deficiencies of the informal notion of *active state configurations* in the official UML specification.
- Moreover, a high-level dynamic semantics of traces, i.e., sequences of system states, is defined. Traces are built based on a set of noteworthy changes that were identifed as being sufficient to document all changes of a running system that are needed to evaluate OCL constraints.
- Together with the definition of traces, we provided rules that determine when OCL invariants as well as pre- and postconditions have to be checked.

Chapter 5

A Timed UML State Diagram Variant

Quid est ergo tempus?
Si nemo ex me quaerat,
scio;
si quaerendi explicare velim,
nescio.
– Aurelius Augustinus

In this chapter, we define a timed State Diagram formalism we will later apply to model real-time behavior in the context of modeling manufacturing systems. On the one hand, we only allow a limited subset of standard UML State Diagram model elements in our approach – presented in Section 5.1. On the other hand, we extend UML by allowing specification of time-annotated operations in Class Diagrams, i.e., an operation is associated with a timing interval that specifies the required (or estimated) [min,max] execution time. Furthermore, we introduce transition priorities to UML State Diagrams to overcome potential conflicts for firing state transitions. The complete syntax of our timed State Diagram variant is given in Section 5.2. The execution semantics of our timed State Diagram variant is then first informally described in Section 5.3, while a corresponding formal translation to I/O-Interval Structures in Section 5.4 completes the formalization.

The current UML standard does not have an inherent notion of time. This has been investigated by different groups for the design of time-dependent systems, e.g., RT-UML [Dou00], UML-RT based on ROOM [SR98], and *UML Profile for Scheduling, Performance and Time* [OMG03c]. Furthermore, UML (on purpose) leaves several issues open that inhibit a unique formal definition for the dynamic semantics of UML State Diagrams. E.g., there is no particular dispatching policy defined, and it is not clear which event is selected next from the event queue to trigger the next run-to-completion step (RTC-step) within the State Diagram.

While this approach might make sense for the intended general purpose of a modeling language standard, it is essential to have a mathematically precise formal semantics of State Diagrams for formal analysis purposes. Studying the numerous publications on formal semantics of UML State Diagrams, it can be observed that none of these covers all concepts of the extensive syntax of UML State Diagrams. An overview is, for instance, given in [Bee02]. Nevertheless, it is often *not* necessary to regard the whole syntax of UML State Diagrams in a specific modeling approach. However, the chosen sublanguage and the dynamic semantics for the specific context must be clearly identified. Thus, a precise modeling approach that makes use of UML State Diagrams must still additionally define a formal dynamic semantics, either by referring to an existing one or defining a new one.

Motivation and General Strategy of the Approach. As the focus of this thesis is on specification of state-oriented real-time constraints with (an extension of) the Object Constraint Language OCL, we have to build upon State Diagrams that are equipped with a notion of time. On the one hand, we want to adopt a large number of UML State Diagram concepts in order to support the rich set of modeling means. On the other hand, we have to define a new semantics with an inherent notion of evolving time. This is simply because we cannot formally relate time-bounded OCL constraints to an untimed referred UML user model.

Of course, we could make use of an existing timed variant of State Diagrams as described in Subsection 2.4.3. But none of the existing approaches considers the combination of concepts we want to address here (e.g., support of elapsed time events). Thus, we have to define our own formal semantics for a new (sub)set of UML State Diagram concepts. First, several design choices have to be made w.r.t. compositionality of State Diagrams, negated triggering events, priority schemes for transitions and events, etc. All these issues will be further addressed in the remainder of this chapter.

One of the most important design choices is the restriction to discrete time. This is due to the considered domain of manufacturing systems (cf. Chapter 6) – as message exchange is performed based on discrete events, we can assume discrete time. However, note that many existing timed State Diagram semantics employ continuous time.

With this decision, it is possible to consider state-transition systems with discrete time as an underlying formal basis for defining the execution semantics of timed UML State Diagrams. In the context of this thesis, we choose I/O-Interval Structures as the target language. With a corresponding mapping of state-oriented real-time OCL constraints to CCTL formulae (cf. Chapter 7), a formal relation between timed State Diagrams and real-time OCL constraints is then automatically established by the semantics of CCTL over I/O-Interval Structures (cf. Section 3.6.2).

Again, note that one can of course apply other timed UML State Diagram variants that also have a well-defined semantics, e.g., hierarchical timed automata [DMY02]. But then a different – yet similar – translation of real-time OCL constraints to a corresponding other temporal logics, e.g., timed CTL (TCTL) with continuous time has to be applied (cf. Chapter 7).

Relation to Synchronous Languages. In accordance with UML, we support point-to-point communication – however, signal broadcasts are considered to be part of UML 2.0. Actually, it turns out to be very easy to also integrate signal broadcasts into the formalization presented here, as the target language of I/O-Interval Structures already supports global visibility of signals.

But in the context of this thesis, we focus on point-to-point communication with asynchronous signals and synchronous operation calls. Synchronous operation calls block the sending object until a result is received. This causes the calling object to *wait*, which will take some time in a running system. One can abstract from this issue and assume that the system is *fast enough* to reply without a *notable time*. This assumption refers to the *synchrony hypothesis* [Ber89]. Basically, the synchrony hypothesis says that a reaction on input events does not take a notable time. If a system can thus react on all input events without loss of a received event and if this property can be actually checked, modeling under the synchrony hypothesis is as good as modeling with an explicit physical timing model.

But this is somehow in contrast to the *run-to-completion steps* (RTC-steps) defined in UML. UML employs event queues, from which one event is dispatched at a time to perform a so-called RTC-step, leading from one stable *active state configuration* to another stable active state configuration. Basically, this refers to a *superstep* known from other Statechart approaches. When considering evolving time, it is obvious that no two different stable active state configurations can exist at the same point of time, thus one has to assume a minimal elapsed time of *one time unit* between two stable active state configurations. This refers to a *unit delay structure*, similar to μ-charts [PS97], a Statechart variant in which transitions take place in exactly one time unit. μ-charts, however, have a different communication scheme with *instantaneous feedback* that makes a direct comparison with our approach difficult. In the μ-chart communication scheme, the employed unit delay structure complies to the synchrony hypothesis, whereas in our approach, a Statechart might react on an input event not instantaneously, based on the dispatching mechanism that takes only one event at a time out of the event queue of waiting events.

Supported UML State Diagram Concepts. In our timed variant of syntactically restricted UML State Diagrams, we neglect some rarely used pseudo-state concepts (e.g., synch states) and do not allow transitions to cross borders of And-states (as such transitions lead to subtle side effects and determination of the next active state configuration is not possible without further conventions, see Section 5.1). Furthermore, we annotate operations in Class Diagrams by operation times. This is possible with standard UML means by a stereotyped note attached to an operation (cf. the *UML Language User Guide* [BRJ99, p. 324]). Another, though non-standard, way is to simply attach a time or timing interval directly to the operation as shown in Figure 5.1.

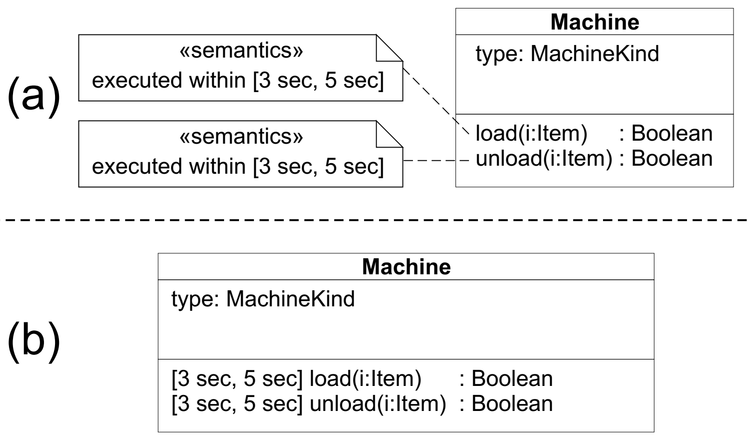

(a)
«semantics»
executed within [3 sec, 5 sec]
«semantics»
executed within [3 sec, 5 sec]
Machine
type: MachineKind
load(i:Item) : Boolean
unload(i:Item) : Boolean
(b)
Machine
type: MachineKind
[3 sec, 5 sec] load(i:Item) : Boolean
[3 sec, 5 sec] unload(i:Item) : Boolean

Figure 5.1: Operations Specified with Execution Times, (a) in standard UML Notation Using Structured Text, and (b) Our Shorthand Notation

A time expression attached to an operation in such a way specifies the operation's time complexity, typically the minimal/maximal time of *expected* completion of an operation execution. Such specifications can be used in different ways, e.g., the resulting running system can be compared with the asserted times specified in the model. Alternatively, by adding up (asserted or actual) operation times, compound times of entire transactions can be computed.

5.1 Syntactical Restrictions

Among the different state notions within the UML, only composite states, simple states, final states, and initial pseudo states are considered. In particular, we do not regard the following UML State Diagram modeling elements for states in our Timed State Diagram notation:

- **StubState and SubmachineState.** Submachine and stub states are used for syntactical convenience and can be substituted by actual composite states.

  In UML, submachine states are a syntactical convenience to represent a 'call' to a another state machine as a 'subroutine', using stub states as entry and exit points. Thus, a submachine state is semantically equivalent to a composite state, and we can assume that all these states have explicitly been copied into SC, such that all submachine states and stub states are eliminated.

- **SynchState.** Synch States in UML State Diagram Diagrams are used to model synchronizations among orthogonal regions. The firing of outgoing transitions from a synch state can be limited by a bound on the difference between the number of times outgoing and incoming transitions have fired. Synch states can be simulated by additional internal signals.

- **Junction PseudoState.** These are splits (also called forks or static conditional branches) or merges (also called joins) of transitions. They are for syntactical convenience and can simply be replaced by specifying corresponding simple state-to-state transitions.

- **Choice PseudoState.** These are dynamic conditional branches that can be simulated by adding intermediate states, provided that visiting these intermediate states does not take any additional time (as it is possible in the run-to-completion step semantics). In our approach, though, transition to another state consumes at least one time unit, and therefore these states cannot be directly simulated here.

With a translation to state-transition systems in mind, there are some more concepts in UML State Diagrams that we do not need or explicitly want to abstract from. In particular, the following UML State Diagram concepts are not regarded for our Timed State Diagram variant.

- **Internal Transitions.** Internal Transitions do not trigger entry- and exit-actions and are sometimes even seen as unnecessary [Sim00], as internal transitions are actually modeling behavior that belongs to a substate.

- **History States.** Shallow and deep history are a convenient modeling elements when recently exited substates should be re-entered.

  In our mapping to I/O-Interval Structures, exiting a composite state x is performed by setting an internal variable *activated* to false, while the most recent activated substate, say s, is retained in another internal variable $state$. When re-entering composite state x via a history state, variable *activated* has simply to be set to true, and the right substate s is then automatically 're-entered'.

But although it is generally possible to construct a translation of this concept and even no additional states are necessary, several additional cases have to be distinguished for the already quite complex transition mapping that is illustrated in Section 5.4.2. Therefore a formalization of this concept is left out in this thesis.

- **Object Creation/Deletion.** Actions for dynamic creation/deletion of objects are not supported, as we need to know all participating objects in advance to be able to instantiate a corresponding system with Kripke Structures. This is also a requirement when real-time model checking as described in Section 3.4 is to be performed.
- **Event Parameters.** Though standard UML allows parameters not only for operation calls but also for asynchronous signals, we do not regard event parameters. Note that these can be simulated by specifying a set of parameterless events, built based upon the cross product of the parameters' value sets.
- **Deferred Events.** Deferred events can be simulated by regenerating them as often as they are to be deferred.
- **Interlevel Transitions.** We are going to give well-formedness rules for interlevel transitions that restrict the set of possible interlevel transitions. In particular interlevel transitions that cross the border of composite states are critical w.r.t. the affected orthogonal regions.
- **Local Variables.** Local Variables in State Diagrams can be simulated by attributes defined in the class the State Diagram belongs to.

What we preserve from UML State Diagrams are hierarchical states, interlevel transitions to/from concurrent composite states, and synchronous and asynchronous event communication.

5.2 Syntax

We first formally define the supported subset of UML State Diagrams. Let $\mathcal{N}$ be a set of names.

Definition 5.1 *A timed UML-like State Diagram SC is a tuple*

$$\begin{aligned} SC \overset{def}{=} \langle\ & S, init, final, EVTS, GUARDS, ACTS, TR, \\ & substates, entry, exit, doActivity\ \rangle, \end{aligned}$$

where

1. $S \subseteq \mathcal{N}$ is a set of states. S is the union of the following disjoint sets.
    - Simple (or: basic) states, denoted by $Simple$,

- Composite states – denoted by $Composite$ – consisting of the two disjoint sets of sequential composite states Xor and orthogonal composite states And.

For convenience, we make use of function

$$type : S \rightarrow \{Simple, And, Xor\}$$

to assign a type to each proper state.

Note that initial and final states are now handled in separate sets and functions.

2. Function $init : Composite \rightarrow \mathcal{P}(S)$ defines the initial state(s) of a composite state.

   For each state $s \in Xor$, $init(s)$ contains exactly one direct substate x of s, which is the default state that is entered (or: activated), when s is entered. In this case, we might just write $init(s) = x$ instead of the formally correct version $init(s) = \{x\}$. For each state $s \in And$, we always have $init(s) = substates(s)$. Note here that it holds $\forall s \in And : init(s) \subseteq Xor$.

3. Final states may appear as children of Xor states. Though it is generally allowed in UML to draw several final states within a single Xor-state, we formally define that there is at most one final state as a child of each state $s \in Xor$.

   Function $final : Xor \rightarrow Final$ returns the final state of a given Xor-state s, if existent. If no final state is specified, $final(s) = \varnothing$. In the remainder, we may use $final(Xor)$ to denote the set of all final states.

   For dealing with (implicitly generated) completion events, we additionally define function $\tau : Conf \times Composite \rightarrow Bool$ that returns true, iff composite state s is terminated in configuration c.

$$\tau(c, s) \overset{def}{=} \begin{cases} false & \text{if } type(s) = Xor \ \wedge \ final(s) \notin c \\ true & \text{if } type(s) = Xor \ \wedge \ final(s) \in c \\ \bigwedge_{x \in substates(s)} \tau(c, x) & \text{if } type(s) = And \end{cases}$$

4. $EVTS \subseteq EXPR_{Evts}$ is a set of events. We assume that there is an expression language $EXPR_{Evts}$ available to formulate events such as operation calls, signals, timers, etc. We propose the following basic syntax:

```
^s     // asynchronous call event, s is the name of a signal received
op()   // synchronous call event, op is the name of an operation call
```

   An obvious extension to that proposed action syntax would be to allow *event parameters* as specified in standard UML. With the proposed basic syntax, event parameters can be simulated by a set of parameterless events, i. e., one event for each element of the cross product of the parameter value sets.

For reasons of formality, let $\xi_{evt} \in EXPR_{Evts}$ be the null event which is later used to denote that no triggering event is associated with a transition. As mentioned above, we do not regard event parameters.

Elapsed time events. A relative time expression denoting an *elapsed time event* in $EXPR_{Evts}$ is usually specified by ”`after tm`”, with `tm` $\in \mathbb{N}$. Semantically, `tm` is relative to the assumed minimal time unit. E.g., if the minimal time unit is 1ms, ”`after 1000`” is equivalent to ”`after 1 sec`”. However, we assume in the remainder that all specified timers `tm` directly refer to the assumed minimal time unit.

5. $GUARDS \subseteq EXPR_{Guards}$ is a set of conditions. We assume that there is a language $EXPR_{Guards}$ available to formulate boolean expressions (e.g., standard OCL). We use $true \in EXPR_{Guards}$ to refer to the guard that is always valid.

6. $ACTS \subseteq EXPR_{Acts}$ is a set of actions. We assume that there is an expression language $EXPR_{Acts}$ available to formulate actions such as assignments, operation calls, signals, etc. For convenience, we here define a very basic action syntax with

```
v := expr     // assignment, v is an attribute, expr is an expression
              // that evaluates to a value of the type of v
objId.^s      // asynchronous call action,
              // s is a signal sent to object objId
send objId.s  // alternative syntax for signals sent
              //
objId.op()    // synchronous call action,
              // op is an operation defined for object objId
```

This syntax can be easily extended if necessary (e.g., by event parameters similar as in $ACTS$), but note that the effects from executing the actions must be formalized in the semantics definition.

For reasons of formality, let $\xi_{act} \in EXPR_{Act}$ be the empty action that is associated with no action when evaluated.

7. $TR \subseteq S \times EVTS \times GUARDS \times ACTS \times (S \cup final(Xor))$ is a set of transitions. A transition connects a source state $s \in Proper$ with a destination state $s' \in S \cup final(Xor)$, may have a trigger event $e \in EVTS$, a guard condition $g \in GUARDS$, and an action expression $a \in ACTS$.

UML does not allow transitions to cross borders of an And-state with a source outside of that And-state. In addition, we also do not allow a transition to start in a substate of an And-state a leading to a state outside of a (see Figure 5.2). We are going to formalize this well-formedness constraint at the end of this section.

Analogously to Definition 4.5, the five convenience functions $tr_{src} : TR \rightarrow S, tr_{dst} : TR \rightarrow S \cup final(Xor), tr_{evt} : TR \rightarrow EVTS, tr_{grd} : TR \rightarrow GUARDS, tr_{act} : TR \rightarrow ACTS$ are used to extract the source state, destination state, triggering event, guard, and action of a given transition, respectively.

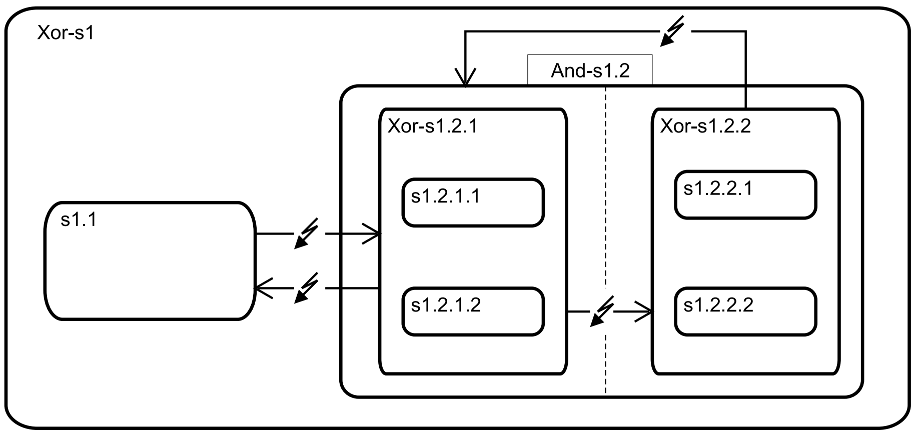


Figure 5.2: Invalid Transitions Crossing Boundaries of And-States

8. Funtion $substates : Composite \to \mathcal{P}(S)$ gives all immediate substates of a state, such that

   (a) there is a unique state $top \in Composite$ with $\forall s \in Composite : top \notin substates(s)$,

   (b) $\forall s \in And : substates(s) \subseteq Xor \ \wedge \ |substates(s)| > 1$, [1]

   (c) $\forall s \in Composite \setminus \{top\}$ there is exactly one path

   $$\langle s_1, \ldots, s_n \rangle \in \underbrace{Composite \times \ldots \times Composite}_{n\ times,\ n \geq 2},$$

   with $s_1 = top \ \wedge \ s_n = s \ \wedge \ s_{i+1} \in substates(s_i)$ for $1 \leq i \leq n-1$.

   Additionally, function $substates^+ : Composite \to \mathcal{P}(S)$ is used to get all (transitive) substates of a state, i. e.,

   $$substates^+(s) = substates(s) \ \cup \bigcup_{x \in substates(s)} substates^+(x).$$

   We also make use of $substates^*(s) = substates^+(s) \cup \{s\}$.

   Function $substates^+$ defines an irreflexive partial ordering among states (denoted by $<$), i. e., we write $s' < s$, iff $s' \in substates^+(s)$. It holds

   $$s'' < s' \ \wedge \ s' < s \ \Rightarrow \ s'' < s \qquad \text{(Transitivity)}$$

   If $s' < s$, we call s an *ancestor* of s'. If either $s' < s$ or $s < s'$, we say that s and s' are *ancestrally related*. We also make use of the reflexive partial ordering (denoted by $\leq$), i. e., $s' \leq s$, iff $s' \in substates^*(s)$.

[1]The first condition is a restricted variant of a well-formedness rule in the UML standard [OMG03d, Section 2.12.3.1], which also allows And-states as substates of an And-state. Our proposed definition ensures that all direct substates of an And-state are Xor-states.

The *least common ancestor* c for a set of states $X \subseteq \mathcal{P}(S)$ is defined by function

$$lca : \begin{cases} \mathcal{P}(S) \;\rightarrow\; Composite & \\ \quad X \;\mapsto\; c & \text{where } \forall x \in X : x \leq c \land \\ & \forall y \in S : (\forall z \in X : z \leq y) \Rightarrow c \leq y \end{cases}$$

Two states s and s' are *orthogonal* (denoted by $s \perp s'$), iff (1) $s \neq s'$, (2) s and s' are not ancestrally related, and (3) $type(lca(\{s, s'\})) = And$.

If $s' \in substates(s)$, we call s the *parent* of s' and define the three functions $parent$, $parent^{+}$, and $parent^{*} : S \rightarrow Composite$ according to function $substates$. We set $parent(top) \overset{def}{=} \varnothing$.

A set of states $X \subseteq S$ is *consistent*, iff for every distinct pair $x, y \in X$ either x and y are ancestrally related or $x \perp y$.

A *configuration* for a state s is a maximal consistent set $C_s \subseteq substates(s)$. Let $Conf$ be the set of all overall configurations, i.e., $Conf$ denotes all configurations for state top.

9. Functions $entry, exit : Proper \rightarrow ACTS$ give the actions to take when a state is entered or left, respectively.

10. Function $doActivity : Proper \rightarrow ACTS$ gives the activity to take when a state is activated. We allow that the activity has a specified timing interval that denotes how long the activity might take at least and at most. Note that this issue is not standard UML, but an extension we make use of for our analysis of timed State Diagram execution.

    UML does not make syntactical restrictions for states with a specified do-activity and different interpretations are possible for these activities, e.g., single execution or periodic execution until some outgoing transition is fired, and in this context preemption of activities, etc.

    To keep concise, we here further restrict states with do-activities that denote local operation calls with a specified execution time greater than 1. We require that those states are *simple states* that have a *unique triggerless outgoing transition* to another (not necessarily simple) state. The reason for these requirements is that it is not possible to perform a transition in one time step if activities were to be executed as part of the chain of actions that is determined by interlevel transitions. However, we allow entry- and exit-actions and an action associated with the outgoing transition for simple states with do-activities.

    As an effect, a state with such a do-activity will remain in that state for the specified (interval) amount of time and then change to the target state of the outgoing transition. Basically, this kind of behavior refers to that of UML Activity Diagrams.

Note that we do not allow transitions to cross boundaries of And-states, as illustrated in Figure 5.2. In Figure 5.3, all kinds of legal transitions between different kinds of states (on possibly different levels of hierarchy) are listed, provided that the transition conforms to the following property:

$$\begin{gathered}\forall tr \in TR : \\ \forall x \in substates^{*}(lca(\{tr_{src}(tr), tr_{dst}(tr)\})) \cap parent^{+}(tr_{src}(tr)) : \\ type(x) \neq And \wedge \\ \forall x \in substates^{*}(lca(\{tr_{src}(tr), tr_{dst}(tr)\})) \cap parent^{+}(tr_{dst}(tr)) : \\ type(x) \neq And\end{gathered}$$

to / from	lower level				same level				upper level		
	Simple	Xor	And	History	Simple	Xor	And	final	Simple	Xor	And
s is initial (pseudo) state	---	---	---	✓	✓	✓	✓	---	---	---	---
type(s) = Simple	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
type(s) = Or	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
type(s) = And	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Figure 5.3: Legal Transitions

5.3 Semantics

The official UML Specification does not make specific timing assumptions for execution of State Diagrams, e.g., it is possible for transitions to both be instantaneous or to take time [OMG03d, Section 3.75.1]. Most-often, it is assumed that transitions take no time. The same holds for states; they can be instantaneous as well as having a notable duration, e.g., when an activity is specified for a state.

For a formal analysis of timed UML State Diagrams, we have to provide a precise execution semantics w.r.t. evolving time. So, we have to decide for which operational parts time is evolving and how much time is needed for execution. We decide that

- A run-to-completion step takes one time unit. Basically, the system will change from one stable source configuration denoting the current status of the object before commencing the RTC-step to another destination configuration denoting the subsequent stable state configuration right after the RTC-step. In other words, when considering evolving time as an inherent characteristic of the system, two subsequent stable state configurations cannot occur at the same time instance. Thus, we assume at least a minimal expired time of one time unit between two stable state configurations to be able to distinguish these two configurations, no matter if any actions or activities are to be executed or not.

  If actions (i.e., exit, transition, or entry actions) are to be executed, we assume the following timing scheme:

  - assignments take no additional time (w.r.t. the run-to-completion step),
  - asynchronous signal calls take no additional time (w.r.t. the run-to-completion step),
  - a synchronous operation call blocks the corresponding region until a corresponding response is received. Note that a synchronous operation call blocks the State Diagram region where the call is sent from, and in the meantime, time is of course evolving.

- The dispatching mechanism dispatches an event as soon as the previous RTC-step is completed right at the next instance of time. In order to be able to proceed in one time step, transition priorities have to be specified for conflicting transitions (cf. Section 5.4.2.3).

- We abstract from communication times, as we are primarily interested in duration of object activities. The time an event needs when it is sent from one object to another is one time unit by default, i.e., a signal sent as a reaction to some dispatched event is visible after one time unit at the target object. Other approaches allow to specify delay times for events [KMR02]. Note that this can easily be integrated into our approach as well (see Section 5.4.1.3).

- In a timed model, synchronous operation call events sent to *other objects* cannot be seen as actions ”with negligible time” as described in the UML specification, as time evolves when waiting for an response on that operation call. Thus, we do not allow such operation calls as exit-, transition-, or entry-actions of states. Furthermore, as we extend UML by timing annotations on operations, synchronous operations invoked as exit-, transition-, or entry-actions for the object itself must not have a specified execution time larger than 1.

- We restrict on clock-synchronous semantics, i.e., an object dispatches a new event from its event queue to be processed by the corresponding State Diagram only at the tick of the (global) clock, in the moment when the previous RTC-step is completed. In the official UML specification, this issue is left open, such that alternatively, a clock-asynchronous semantics can be assumed, i.e., an object dispatches a new event as soon as it can.

- We allow synchronous as well as asynchronous communication, i.e., synchronous operation calls block the sender until a returning answer is received, and asynchronous signals sent appear as incoming signal events on the receiver side, but do not block the sender. However, we restrict the positions where synchronous operation calls can be made; these are only allowed as activities, as they block the State Diagram until an answer is received, and that will take some time.

- We assume a perfect underlying communication technology, i.e., none of the communicated events will be lost. We also abstract from the time needed for an event sent from the sending object to the receiving object. An event is visible on the receiver’s side at the next instance of time, i.e., the event will immediately be inserted into the event queue of State Diagram associated to the receiving object. (This is different to, e.g., my-Charts, in which the synchrony hypothesis is preserved, i.e., in μ-Charts an action and the event causing this action occur at the same instant of time.)

A possible extension is to attach a time (or timing interval) to activities to denote the assumed time needed to terminate a certain task that is associated with the state. In some approaches, that time specification may even be infinite, leading to situation in which a sent call is never getting to its receiving object. In this case, model analysis has to take into account additional so-called fairness properties, i.e., only those executions are investigated that exclude such situations. As we assume a perfect technology, we can abstract from this issue.

- As standard UML assumes, no compound triggers (i.e., at most one triggering event for a transition), no negated trigger events, and only a single entry and exit action attached to each state and a single action attached to each transition are allowed. Among the three actions that are to be executed when a transition is taken (exit-, entry-, and transition action), we do not allow mutual dependencies.[2] E.g., we do not allow a transition that connects states with exit action `x := 1` and entry action `x := x + 2`.

- Priority preorder is UML-conform, i.e., transitions on a lower level have precedence on higher level transitions.

- Instantaneous states are not possible due to our time semantics. The State Diagram remains for at least one time unit in each entered non-pseudo state.

- The causality principle (cf. Section 2.4.3 is preserved due to our time semantics.

- We limit the length of event queues by the following assumptions. The most important assumption is, that a limited, in advance well-known maximal number of objects of each class is instantiated. Then, for each State Diagram (that is instantiated for an object), the number of concurrently occurring synchronous call events is limited by the number of parallel substates of all potential calling objects. We still have to cope with a potential infinite number of received asynchronous signal events. This is limited by suppressing a signal (named sig) to be sent from object obj_1 to obj_2, if there is already such a signal sig waiting in the event queue of obj_2 that has been sent from obj_1 beforehand. Thus, a maximal number of concurrently events in the event queue can be determined based on the number of objects, and the event queue can be modeled by a finite number of states that represent the received events.

- Elapsed time events w.r.t. absolute time, e.g., `when(time=15:00:00h)`, are not supported.

In Table 5.1, we present a comparison of the main differences between standard UML State Diagrams and our Timed State Diagram variant.

5.4 Translation to I/O-Interval Structures

The problem of flattening hierarchical State Diagrams into flat FSMs has already been addressed in several publications (cf. Section 2.4.3). We here give a constructive translation of

[2]This is a restriction due to the translation to the execution semantics of I/O-Interval Structures in Section 5.4.

Table 5.1: Comparison of Standard UML State Diagrams and Our Timed State Diagram Variant

Concept	UML State Diagrams	Our Timed State Diagram
Incoming Event	Events are stored in an unlimited event queue and subsequently dispatched (one per RTC-step).	A limited event queue can store incoming events, i.e., one operation event and one signal event per sending object can be queued.
Event Dispatching	It is assumed that a reaction to all incoming events is eventually been carried out, but no specific dispatching mechanism is provided.	Non-deterministic choice.
Deferred Event	Events that are dispatched can be deferred for later consumption.	Not supported.
Transition Priority	Innermost transition has higher priority.	Innermost transition has higher priority. Additional priorities for conflicting transitions.
Interlevel Transitions	No interlevel transitions crossing boundaries of And-states in inbound direction.	In addition, also no interlevel transitions crossing boundaries of And-states in outbound direction.
Transitions Time	No standard semantics about time consumption, but most commonly, it is assumed that transitions take no time.	Getting from the source state to the destination state takes one time unit.
Actions	Actions have a negligible duration and can be specified with transitions and entering/exiting of states.	Actions have a negligible duration and can be specified with transitions and entering/exiting of states. Some restrictions on interdependencies among actions.
Activities	Ongoing activities can be specified in states (do/-activity).	Activities can be modeled by those operations that have a specified timing interval. Default is [1,1].
Communication	Non-instantaneous, but no assumptions or specification means for how long it takes to send a message.	Sent messages visible after one time unit. If required, an interval can be specified to simulate a delay.
Connectivity	No assumptions whether messages may be lost or not.	Communication is always correctly performed, no message will be lost.
Clock	No assumptions about clocks.	One local clock per State Diagram to measure elapsed time since last entering of a state. Processing based upon a global clock tick.

our restricted State Diagram approach to I/O-Interval Structures. We make use of both, the formal mathematical set-theoretic definition of I/O-Interval Structures as defined in Definition 3.14 (cf. Section 3.6.1.1), and the more convenient representation of I/O-Interval Structures in RIL (cf. Section 3.6.3).

Note that this translation does *not* claim to result in an efficient structure w.r.t. the target language of I/O Interval Structures. In particular, hierarchical states with interlevel transitions induce quite complex structures for flat FSMs, and additional signals have to be introduced to manage the 'synchronization' among the different composite state levels.

Given a State Diagram SC as defined in Definition 5.1, we first determine the sets of exited and entered states for each transition $tr \in TR$. We define the least common ancestor for a transition $tr \in TR$ by function

$$lca \overset{def}{=} \begin{cases} TR & \rightarrow \; Composite \\ tr & \mapsto \; lca(\{tr_{src}(tr), tr_{dst}(tr)\}) \end{cases}$$

For each $s \in Composite$, let $InitConf_s$ be the set of default initial substates relative to s, i.e.,

$$\begin{aligned} InitConf_s \overset{def}{=} \{x \in S \mid\; & x \in substates^*(s) \;\wedge \\ & (init(parent(x)) = x \;\vee\; x = top) \;\wedge \\ & \forall y \in substates^*(s) \cap parent^+(x) : \\ & init(parent(y)) = y\} \end{aligned}$$

In particular, $InitConf_{top}$ denotes the initial overall configuration of SC, where top is the outermost composite state. The set $ExitStates_{tr}$ of exited states and the set $EnterStates_{tr}$ of entered states for a transition $tr \in TR$ are given by

$$\begin{aligned} ExitStates_{tr} &\overset{def}{=} \{x \in S \mid x \in parent^*(tr_{src}(tr)) \cap substates^+(lca(tr))\} \\ &\quad \cup \{x \in S \mid x \in substates^+(tr_{src}(tr))\} \\ EnterStates_{tr} &\overset{def}{=} \{x \in S \mid x \in parent^*(tr_{dst}(tr)) \cap substates^+(lca(tr))\} \\ &\quad \cup \{x \in S \mid x \in InitConf_y, \text{with } y = tr_{dst}(tr)\} \end{aligned}$$

When a transition tr is taken from a source state $s_1 = tr_{src}(tr)$, state s_1 and all parent states of s_1 up to but excluding the least common ancestor $lca(tr)$ as well as all substates of s_1 are exited. Analogously, the set of entered states is determined by state $s_2 = tr_{dst}(tr)$, its parent states up to but excluding the least common ancestor $lca(tr)$, and those substates of s_2 that build the initial configuration $InitConf_{s_2}$. By definition, at most one substate of an Xor-state is in $EnterStates_{tr}$, i.e., for all $tr \in TR$ and $x \in Xor$, it holds

$$0 \leq |substates(x) \cap EnterStates_{tr}| \leq 1.$$

With these preliminaries, we are now going to successively build I/O-Interval Structures for a given State Diagram SC. We start with generating *rudimentary* I/O-Interval Structures that comprise all necessary variables and local input and output signals, but do not include transitions yet (Subsection 5.4.1). The set of transitions for I/O-Interval Structures is then determined in Subsection 5.4.2. Conflicting transitions are tackled in a separate subsection at the end of this chapter.

5.4.1 Generating I/O-Interval Structures

We here assume that we have an instantiated UML model (as in Definition 4.1 on page 86) with an initial system state (as in Definition 4.16 on page 105). We further assume that *all* objects necessary to perform the execution are already instantiated at the initial state of the system; recall that we do not support dynamic creation and deletion of objects. While this issue cannot generally be employed, it is acceptable for the considered domain of manufacturing systems, in which active objects represent the production processes that transform production items (cf. Chapter 6).

For each active object $objId$ that is associated via its class c with a State Diagram SC, the following I/O-Interval Structures will be constructed.

1. For each composite state s of SC, we define a separate I/O-Interval Structure $IS_{objId,s}$. These will first be only rudimentary, but will be completed step by step in subsequent sections, considering different cases, e.g., transitions with/without an elapsed time event or intralevel/interlevel transitions. As interlevel transitions have an effect on several other I/O-Interval Structures $IS_{objId,s'}$, we have to synchronize all affected I/O-Interval Structures by additional signals.

2. A separate I/O-Interval Structure $VarIS_{objId}$ is responsible for keeping attribute values and executing actions. Firing a transition tr in an I/O-Interval Structure $IS_{objId,s}$ causes $VarIS_{objId}$ to execute the corresponding actions associated with that transition. Note that it can happen that in orthogonal regions, transitions can be executed in parallel due to the same specified triggering event. In that case, execution of actions must not have side-effects.

3. The event queue for SC is modeled by a set of I/O-Interval Structures over the cross product of objects that can send events to $objId$ and the particular (synchronous and asynchronous) events on which SC reacts.

4. The event dispatcher $DispatchIS_{objId}$ is an I/O-Interval Structure that non-deterministically selects queued events from the event queue. This part can be replaced by different priority schemes if necessary.

5.4.1.1 Rudimentary I/O-Interval Structures for Composite States

Let ATT_c, OP_c, and SIG_c be the attributes, operations, and signals defined in class c, respectively.

Assuming that we have a fixed number of objects, i.e., we know in advance which objects will be part of the system, we can statically determine the objects (object identifiers) that exchange messages by analyzing the State Diagrams associated with these classes. Thus, for each object $objId \in \Sigma_{CLASS,c}$, we define the set of objects that may send a particular operation (signal) call event to $objId$ by functions

$$opSenders : \begin{cases} I_{CLASS,c} \times OP_c & \rightarrow \mathcal{P}(\Sigma_{CLASS}) \\ \langle objId, op \rangle & \mapsto \{ \Sigma_{CLASS,c'} \mid c' \in CLASS \land \\ & \qquad \exists evt \in EVTS_{c'} : evt = op \} \end{cases}$$

$$sigSenders : \begin{cases} I_{CLASS,c} \times SIG_c & \rightarrow \mathcal{P}(\Sigma_{CLASS}) \\ \langle objId, sig \rangle & \mapsto \{ \Sigma_{CLASS,c'} \mid c' \in Class \land \\ & \qquad \exists evt \in EVTS_{c'} : evt = sig \} \end{cases}$$

We now generate a rudimentary I/O-Interval Structure $IS_{objId,s}$ for each state $s \in Composite$ as it is shown in Figure 5.4 on page 131.

For clarification purposes, some comments on Figure 5.4 are necessary here. First of all, we make use of double square brackets as used in denotational semantics – $[[expr]]$ – to explicitly refer to the *evaluation value* of an expression $expr$, e.g., in $(state \equiv [[init(s)]])$, the subexpression $[[init(s)]]$ actually refers to the name of the initial state of composite state s. For transitions $tr \in TR$, we assume a naming scheme that allows to uniquely determine $[[tr]]$, e.g., a simple numbering `t1`, `t2`, etc.

The actual *states* of I/O-Interval Structure IS_{objId} are of course the (direct) substates of s. But additionally, we explicitly model whether s is activated or not, using a boolean state called $activated$. Furthermore, for all relevant transitions tr (i.e., those transitions with $s = tr_{src}(tr)$), we define a boolean variable $done\_[[tr]]$ for synchronization with other I/O-Interval Structures after completion of transition tr.

Several *input variables* are defined in Figure 5.4 to access (boolean) variable values from other I/O-Interval Structures. Variables $input\_[[evt]]$ are needed to determine whether an operation or signal call event evt is selected to be dispatched. Variables $grdValue\_[[tr]]$ are used to check whether a transition condition $tr_{grd}(tr)$ is true. Note that the actual condition expression $tr_{grd}(tr)$ is evaluated remotely in an I/O-Interval Structure called $VarIS_{objId}$ (see Figure 5.5). Finally, input variables $fire\_[[tr]]$ combine event triggers and guards for each transition tr in the set TR_s of transitions that affect a composite state s.

The initial configuration s_0 of $IS_{objId,s}$ is constructed by assigning initial values to all elements of set Q, depending on the type of $s \in Composite$ ($s \in And$ or $s \in Xor$) and on the initial overall configuration defined by State Diagram SC.

Note that this construction is by now only resulting in a set of rudimentary, incomplete I/O-Interval Structures. The set T of transitions and functions I, I_{input}, and T_{assgn} are still to be defined in each case. This will be discussed in Section 5.4.2.

5.4.1.2 I/O-Interval Structures for Variables and Assignments

A separate I/O-Interval Structure $VarIS_{objId}$ is responsible for keeping attribute values and executing actions. Firing a transition tr in an I/O-Interval Structure $IS_{objId,s}$ causes $VarIS_{objId}$ to execute the corresponding actions associated with that transition. See Figure 5.5 for the definition of $VarIS_{objId}$.

Given a state $s \in Composite$.
Let $state$ be a variable with $Value(state) = \{x \mid x \in substates(s)\}$.
Let $activated$ be a variable with $Value(activated) = \{true, false\}$.
Let $TR_s = \{tr \in TR \mid s \in ExitStates_{tr} \vee s \in EnterStates_{tr}\}$ be the set of transitions tr that affect s.
Let $parentTR_s = \{tr \in TR \mid s = tr_{src}(tr)\}$.
Let $done\_[[tr]]$ be a boolean variable for each $tr \in parentTR_s$.
Let $OpEvts_s \subseteq \bigcup_{tr \in TR_s} tr_{evt}(tr)$ be the operation calls specified for transitions in TR_s.
Let $SigEvts_s \subseteq \bigcup_{tr \in TR_s} tr_{evt}(tr)$ be the signal triggers specified for transitions in TR_s.
According to Definition 3.14, we generate a rudimentary I/O-Interval Structure
$IS_{objId,s} = \langle Q, Pr, Pr_{input}, S, s_0, Var_0, T, L, I, I_{input}, I_{output}, T_{assgn}\rangle$
with:

$$Q = \begin{cases} \{state, activated\} \cup \{done\_[[tr]] \mid tr \in parentTR_s\}, & \text{if } s \in Xor \\ \{activated\} \cup \{done\_[[tr]] \mid tr \in parentTR_s\}, & \text{if } s \in And \end{cases}$$

$$Pr = \left\{ (var_i \equiv val_{var_i}) \mid 1 \leq i \leq |Q| \wedge var_i \in Q \wedge val_{var_i} \in Val(var_i) \right\}$$

$$\begin{aligned} Pr_{input} = & \left\{ (input\_[[opEvt]] := DispatchIS_{[}[objId]].dispatch\_[[opEvt]]) \mid opEvt \in OpEvts_s \right\} \\ \cup & \left\{ (input\_[[sigEvt]] := DispatchIS_{[}[objId]].dispatch\_[[sigEvt]]) \mid sigEvt \in SigEvts_s \right\} \\ \cup & \left\{ (grdValue\_[[tr]] := VarIS\_[[objId]].[[tr]]) \mid tr \in TR_s \text{ with } tr_{grd}(tr) \neq \varnothing \right\} \\ \cup & \left\{ (fire\_[[tr]] := input\_[[tr_{evt}(tr)]] \wedge grdValue\_[[tr]]) \mid tr \in TR_s, tr_{evt}(tr) \neq \varnothing, tr_{grd}(tr) \neq \varnothing \right\} \\ \cup & \left\{ (fire\_[[tr]] := input\_[[tr_{evt}(tr)]]) \mid tr \in TR_s, tr_{evt}(tr) = \varnothing, tr_{grd}(tr) \neq \varnothing \right\} \\ \cup & \left\{ (fire\_[[tr]] := grdValue\_[[tr]]) \mid tr \in TR_s, tr_{evt}(tr) = \varnothing, tr_{grd}(tr) \neq \varnothing \right\} \end{aligned}$$

$$S = Value(state) \times Value(activated)$$

$$s_0 = \begin{cases} (state \equiv [[init(s)]]) \wedge (activated \equiv true), & \text{if } s \in Xor \cap InitConf_{top} \\ (state \equiv [[init(s)]]) \wedge (activated \equiv false), & \text{if } s \in Xor \setminus InitConf_{top} \\ (activated \equiv true), & \text{if } s \in And \cap InitConf_{top} \\ (activated \equiv false), & \text{if } s \in And \setminus InitConf_{top} \end{cases}$$

$$Var_0 = \bigwedge \left\{ (done\_[[tr]] \equiv false) \mid tr \in parentTR_s \right\}$$

$$L : S \to \mathcal{P}(Pr) \text{ with } L(s) = \left\{ (state \equiv s), (activated \equiv true) \right\}$$

$$I_{output} = \left\{ (executed\_[[tr]] := (done\_[[tr]] \equiv true)) \mid tr \in parentTR_s \right\}$$

Figure 5.4: Rudimentary I/O-Interval Structures for Composite States

Finite Value Sets. Note here that only finite value sets for variables can be handled in the translation, such that specified variables of type `Real`, `String`, or `Integer` have to be restricted to a finite value set. For simplicity reasons, we here require that all variables are defined over finite value sets already on the modeling level of UML within Class Diagrams and State Diagrams, i.e., all variables are basically of enumeration types with a limited set of values.

Related approaches that have formal verification by model checking in mind also restrict the value sets in order to be able to map models into corresponding input languages, e.g., [Qui01]. They additionally employ reduction techniques like abstraction from data values to get a model representation with a reduced state space. However, this is not in the scope of this thesis.

Let $state$ be a variable with $Value(state) = \{ok\}$.
Let $VARS_c = \{var_1, ..., var_m\}$ be all attributes defined for class c to which SC is associated.
For each var_i, $1 \leq i \leq m$, let $Value(var_i)$ be a finite enumeration of variable values and let $init(var_i)$ be the initial value of var_i.
We generate a rudimentary I/O-Interval Structure
$VarIS_{objId} = \langle Q, Pr, Pr_{input}, S, s_0, Var_0, T, L, I, I_{input}, I_{output}, T_{assgn}\rangle$
according to Definition 3.14 and define:

$$\begin{array}{ll}
Q & = \{state\} \cup VARS_c \\
Pr & = \{(var_i \equiv val_{var_i}) \ \mid \ 1 \leq i \leq |Q| \ \wedge \ var_i \in Q \ \wedge \ val_{var_i} \in Val(var_i)\} \\
Pr_{input} & = \{fire\_[[tr]] \ \mid \ tr \in TR, tr_{grd}(tr) \neq \varnothing\} \\
S & = Value(state) \\
s_0 & = (state \equiv ok) \\
Var_0 & = (var_1 \equiv [[init(var_1)]]) \ \wedge \ \ldots \ \wedge \ (var_m \equiv [[init(var_m)]])
\end{array}$$

$L : S \rightarrow \mathcal{P}(Pr)$ with $L(s) = (state \equiv ok)$.

$$I_{output} \ = \{(grdValue\_[[tr]] := [[tr_{grd}(tr)]]) \ \mid \ tr \in TR, tr_{grd}(tr) \neq \varnothing\}$$

The set T of transitions and functions I, I_{input}, and T_{assgn} are still to be defined.

Figure 5.5: Rudimentary I/O-Interval Structure for Variables

5.4.1.3 I/O-Interval Structures for Operation and Signal Calls

Synchronous and asynchronous call events (i.e., operation calls and signals sent) are handled in separate I/O-Interval Structures. More precisely, for each pair of objects (one calling and one callee object), an own I/O-Interval Structure for each operation/signal is generated. To support readability of this translation, we are going to provide these I/O-Interval Structures by means of RIL syntax, the RAVEN Input Language. Note that the expressions inside double square brackets have to be replaced by the values gained from evaluating these expressions, based on the given State Diagram SC.

For a concrete system with a given number of objects, we can determine which objects can send call events to which other objects. For each operation of a class c, we then have to build the cross product of all possible calling objects with all possible callee objects, i.e., we generate a set of I/O-Interval Structures for each operation name `op` (e.g., `getStatus()`), while attaching corresponding object identifiers to the structure's name to be able to distinguish which particular calling object is sending an event to which particular callee object.

For example, `getStatus_agv1_mill` represents the name of the I/O-Interval Structure that keeps track of an operation call `mill.getStatus()` sent from $agv1$ to $mill$. Recall that such a synchronous operation call is only allowed as a *do-activity* in our restricted version of UML State Diagrams.

In the generated I/O-Interval Structures shown below, `objId` represents the identifier of the callee object receiving an operation call event named `opEvt_i` for each $i \in \{1, \dots, n\}$, $n = |OP_c|$. We denote by `objId_i_1` ... `objId_i_q(i)` those calling objects that potentially send an operation call event `opEvt_i` $\in OP_c$ to `objId`. Thus, we generate $\sum_{i=1}^{n} q(i)$ many I/O-interval Structures of the following form, where `i` $\in \{1, \dots, n\}$ and `j` $\in \{1, \dots, q(i)\}$:

```
Module op_[[opEvt_i]]_[[objId]]_[[objId_i_j]]
  STATES  state: {absent,waiting}
  INPUTS  opReceived := IS_[[objId_i_j]].sent_[[opEvt_i]]_[[objId]]
          opExecuted := IS_[[objId]].executed_[[opEvt_i]]
  DEFINE  queued := (state=waiting)
          opReturn := opExecuted
  INIT    state=absent
  TRANS
    |- state=absent  -- opReceived   --> state:=waiting
                                     |-> state:=absent
    |- state=waiting -- opExecuted   --> state:=absent
                                     |-> state:=waiting
```

Internal states are `absent` and `waiting` to model the fact that an operation call is currently queued or not.

Input signals `opReceived` and `opExecuted` are monitoring whether a call has been sent and has been executed, respectively. Based on these 'signals', transitions between internal states `absent` and `waiting` are performed.

Output signal queued is used in the dispatching mechanism to select an event. Output signal `opReturn` is used to synchronize with the calling object to proceed its computation.

Analogously, we denote the `r(i)`-many objects that possibly send a signal call event `sigEvt_i`, `i` $\in \{1, \dots, m\}$, $m = |SIG_c|$, to `objId` by `sigObj_i_1` ... `sigObj_i_r(i)`, and generate I/O-Interval Structures of the following form, where `i` $\in \{1, \dots, m\}$ and `j` $\in \{1, \dots, r(i)\}$:

```
Module sig_[[sigEvt_i]]_[[objId]]_[[sigObj_i_j]]
  STATES  state: {absent,waiting}
  INPUTS  sigReceived := IS_[[sigObj_i_j]].sent_[[sigEvt_i]]_[[objId]]
          sigConsumed := IS_[[objId]].consumed_[[sigEvt_i]]
  DEFINE  queued := (state=waiting)
  INIT    state=absent
  TRANS
```

```
    |- state=absent  -- sigReceived  --> state:=waiting
                                     |-> state:=absent
    |- state=waiting -- sigConsumed  --> state:=absent
                                     |-> state:=waiting
```

Limited Event Queue. The I/O-Interval Structures defined above can only store one event of a certain kind at a time. In other words, the event queue of the State Diagram for an object `objId_2` cannot store an event evt sent from an object `objId_1` more than once. Repeated sending of such events is thus ignored up to the point of time, where the queued event is consumed. For synchronous operation calls, this assumption is legitimate, as the calling object is blocked until the operation returns, i.e., in the meantime no second synchronous operation call can be made by the same calling object (at least unless orthogonal regions do not make concurrent calls). Concerning signals, the situation is different. The semantics we assume here imply that there is no ‘loop’ that iteratively sends signals faster than they can be consumed. Otherwise, events that represent a signal call will be discarded, in contrast to the UML semantics that queue each individual event. However, this situation can be overcome by introducing additional counter variables in the signal modules. But then, one has to specify an upper limit for that counter in order to be able to express this issue in the target language RIL.

5.4.1.4 I/O-Interval Structure for Event Dispatching

We generate another I/O-Interval Structure to non-deterministically select one out of the queued events. Note that UML does not make any statements of how the event dispatching mechanism is implemented. Thus, we here assume that one of the currently queued events is non-deterministically chosen.

Each transition $tr \in TR$ will have a corresponding set of transitions in the affected I/O-Interval Structures to be generated for composite states (see Section 5.4.2). But for each transition tr, there is one particular I/O-Interval Structure IS_x (with $x = parent(tr_{src}(tr)) \in Composite$) that is especially responsible for executing the reactions induced by tr, while all other affected I/O-Interval Structures are just synchronized with IS_x by additional signals.

The following code fragment represents the event dispatching mechanism in RIL syntax, where

- `objId` is a given (uniquely identified) object of an active class c with State Diagram SC,
- for each $i \in \{1, \dots, n\}$, $n = |OP_c|$, we denote the objects that possibly send an operation call event $op_i \in OP_c$ to `objId` by `opObj_i_1`, ..., `opObj_i_q(i)`,
- for each $i \in \{1, \dots, m\}$, we denote the objects that possibly send a signal call event $sig_i \in SIG_c$ to `objId` by `sigObj_i_1`, ..., `sigObj_i_r(i)`,
- `opEvt_1_1`, ..., `opEvt_n_q(n)` denote whether an operation call event is currently in the event queue,
- `sigEvt_1_1`, ..., `sigEvt_m_r(m)` denote whether a signal call event is currently in the event queue,

- for each $i \in \{1, \ldots, |TR|\}$, `IS_[[x_i]]` is the name of the I/O-Interval Structure that manages transition tr_i,
- `executed_[[tr_i]]` is the signal of I/O-Interval Structure `IS_[[x_i]]` that indicates that transition tr_i has been completely executed.

```
Module DispatchIS_[[objId]]
  STATES  state: { select,
                   wait_[[opEvt_1_1]] , ..., wait_[[opEvt_n_q(n)]] ,
                   wait_[[sigEvt_1_1]], ..., wait_[[sigEvt_m_r(m)]] }
   INPUTS
   // list all operation and signal triggers
          in_[[opEvt_1_1]]     := op_[[objId]]_[[opObj_1_1]].queued
          ...
          in_[[opEvt_n_q(n)]]  := op_[[objId]]_[[opObj_n_q(n)]].queued
          in_[[sigEvt_1_1]]    := sig_[[objId]]_[[sigObj_1_1]].queued
          ...
          in_[[sigEvt_m_r(m)]] := sig_[[objId]]_[[sigObj_r(m)_m]].queued
   // transition completion signals
          completed            := IS_[[x_1]].executed_[[tr_1]]
                                  | ...
                                  | IS_[[x_k]].executed_[[tr_k]]
  DEFINE
  // output signals for triggering transitions
          dispatch_[[opEvt_1]]  := (state=wait_[[opEvt_1_1]])
                                   | ... | (state=wait_[[opEvt_1_q(1)]])
          ...
          dispatch_[[opEvt_n]]  := (state=wait_[[opEvt_n_q(n)]])
                                   | ... | (state=wait_[[opEvt_1_q(n)]])
          dispatch_[[sigEvt_1]] := (state=wait_[[sigEvt_1_1]])
                                   | ... | (state=wait_[[sigEvt_1_r(1)]])
          ...
          dispatch_[[sigEvt_m]] := (state=wait_[[sigEvt_m_r(m)]])
                                   | ... | (state=wait_[[sigEvt_m_r(m)]])
  INIT    state=select
  TRANS
  // non-deterministic choice
  |- state=select -- in_[[opEvt_1_1]]     --> state:=wait_[[opEvt_1_1]]
                  ...
                  -- in_[[opEvt_n_q(n)]]  --> state:=wait_[[opEvt_n_q(n)]]
                  -- in_[[sigEvt_1_1]]    --> state:=wait_[[sigEvt_1_1]]
                  ...
                  -- in_[[sigEvt_m_r(m)]] --> state:=wait_[[sigEvt_m_r(m)]]
                                          |-> state:=select
  // wait for completion of transition
  |- state=wait_[[opEvt_1_1]]     -- completed --> state:=select
                                               |-> state:=wait_[[opEvt_1_1]]
  ...
  |- state=wait_[[opEvt_n_q(n)]]  -- completed --> state:=select
                                               |-> state:=wait_[[opEvt_n_q(n)]]
  |- state=wait_[[sigEvt_1_1]]    -- completed --> state:=select
                                               |-> state:=wait_[[sigEvt_1_1]]
  ...
```

```
 |- state=wait_[[sigEvt_m_r(m)]] -- completed --> state:=select
                                               |-> state:=wait_[[sigEvt_m_r(m)]]
```

Based on the chosen event, each I/O-Interval Structure for composite states checks whether it has to execute a transition. A transition mapping to establish this behavior is detailly described in the next section.

5.4.2 Transition Mapping

Some more definitions are necessary to prepare the transition mapping. Basically, each transition $tr \in TR$ must be mapped to a set of transitions, i.e., one new transition is generated for each composite state that is affected by transition t. We divide the composite states of sets $ExitStates_{tr}$ and $EnterStates_{tr}$ into the three following distinct sets.

$$\begin{aligned}
ExitEnter_{tr} &= \{x \in Composite \mid substates(x) \cap ExitStates_{tr} \neq \varnothing \wedge \\
&\qquad substates(x) \cap EnterStates_{tr} \neq \varnothing\} \\
ExitOnly_{tr} &= \{x \in Composite \mid substates(x) \cap ExitStates_{tr} \neq \varnothing \wedge \\
&\qquad substates(x) \cap EnterStates_{tr} = \varnothing\} \\
EnterOnly_{tr} &= \{x \in Composite \mid substates(x) \cap ExitStates_{tr} = \varnothing \wedge \\
&\qquad substates(x) \cap EnterStates_{tr} \neq \varnothing\}
\end{aligned}$$

In $ExitEnter_{tr}$, we find composite states that have (potentially several) exited as well as an entered substate w.r.t. transition tr. The other two sets comprise of composite states that either have exited substates or an entered substate.

As actions in I/O-Interval Structures can only be associated to transitions (and not to states as in UML State Diagrams), entry and exit actions of State Diagram states must be attached to corresponding transitions in I/O-Interval Structures. Unfortunately, we cannot allow interdependencies among these actions, as they are executed in a single step when running the I/O-Interval Structures. Nevertheless, we here keep the order of actions, as this might be useful for future approaches that can consider sequential execution of actions and interdependencies.

For each state that is left, we collect exit actions of all left states and entry actions of all entered states and sort them according to the state hierarchy. The final sequence of actions associated with transition t is then

$$actions_{tr} = \langle\langle exit(x_1), \ldots, exit(x_n), tr_{act}(tr), enter(y_1), \ldots, enter(y_m)\rangle\rangle,$$

where $n = |ExitEnter_{tr} \cup ExitOnly_{tr}|$, $m = |ExitEnter_{tr} \cup EnterOnly_{tr}|$, and actions $exit(x_1)$, ..., $exit(x_n)$ are sorted from lowest to uppermost state, i.e.,

$$\bigcup_{i=1\ldots n} x_i = ExitEnter_{tr} \cup ExitOnly_{tr} \;\wedge\; \forall i \in \{1, \ldots, n-1\} : x_i \in substates(x_{i+1}),$$

and actions $enter(y_1), \ldots, enter(y_m)$ are sorted from uppermost to lowest state, i.e.,

$$\bigcup_{i=1\ldots m} y_i = ExitEnter_{tr} \cup EnterOnly_{tr} \;\wedge\; \forall i \in \{1, \ldots, m-1\} : y_i = parent(y_{i+1}).$$

Transitions with and without Elapsed Time Event. In the following, we distinguish between transitions with and without elapsed time events.

$$\begin{aligned} elapsedTimeTR &= \{tr \in TR \mid tr_{evt}(tr) = \text{'after tm'},\ \text{with tm } > 1\}, \\ commonTR &= TR \setminus elapsedTimeTR \end{aligned}$$

First, we are going to translate all transitions of $commonTR$. A corresponding mapping of transitions with elapsed time events is presented in a separate subsection thereafter.

5.4.2.1 Mapping of Common Transitions

For each $tr \in commonTR$, we add transitions to all affected I/O-Interval Structures IS_x (with $x \in ExitEnter_{tr} \cup ExitOnly_{tr} \cup EnterOnly_{tr} \subseteq Composite$) as follows.

Transition Source State. If the transition source state $tr_{src}(tr)$ does not have a specified timed activity, we add the following transition to the parent state $x = parent(tr_{src}(tr))^3$. We introduce a transition $t = \langle a, b, condVars\rangle$ to the set of transitions T in IS_x, where[4]

$$\begin{aligned} a &= \langle (state \equiv [[tr_{src}(tr)]]), (activated \equiv true)\rangle, \\ dest &= substates(x) \cap EnterStates_{tr}, \\ b &= \begin{cases} \langle (state \equiv [[dest]]), (activated \equiv true)\rangle & \text{if } dest \neq \varnothing \\ \langle (state \equiv [[tr_{src}(tr)]]), (activated \equiv false)\rangle & \text{otherwise,} \end{cases} \\ condVars &= \{input\_[[tr_{evt}(tr)]], grdValue\_[[tr]]\}. \end{aligned}$$

To complete the definition of transition t, we set

$$\begin{aligned} I(t) &= \{1\}, \\ I_{input}(t) &= fire\_[[tr]], \\ T_{assgn}(t) &= \langle (done\_[[tr]] := true)\rangle. \end{aligned}$$

If the transition source state $tr_{src}(tr)$ is a simple state that has a specified timed activity with timing interval [$minTime$,$maxTime$], the transition is – due to our syntactical restrictions – a triggerless transition without any further annotations (i.e., no condition and action). We can thus handle the activity duration simply like an elapsed time event `after [minTime,maxTime]` attached to the transition. Note here that we allow a timing interval for elapsed time events, which is an extension of UML syntax. The transition is then translated in Subsection 5.4.2.2.

Remark. Assume for the moment that all transitions $tr \in TR_s$ have already been considered. For each $tr \in TR_s$ with a specified event trigger $tr_{evt}(tr)$, we still have to reset variables, i.e., setting $done\_[[tr]] := false$, by assignment actions in the respective subsequent transitions (defined for b above), i.e., those transitions that start in the respective destination states. A detailed formal description is left out here.

[3]Note here that in our restricted version of State Diagrams, we always have $x \in Xor$ in this context.

[4]Note that for Xor-states x, it holds $0 \leq |substates(x) \cap EnterStates_{tr}| \leq 1$, such that the destination state b can uniquely be determined.

Transition Actions. For the actions in $actions_{tr}$, we have to add some more new output signals to I/O-Interval Structure IS_x. These output signals guarantee that in the corresponding I/O-Interval Structure $VarIS_{objId}$ for variables and actions, an appropriate transition is synchronously taken that reflects the variable assignment or sending of a call event, respectively.

Assignments. We extract the subsequence $assignments_{tr}$ that includes all variable assignments of $actions_{tr}$ and make some extensions to I/O-Interval Structure $VarIS_{objId}$ as follows.

1. Add a new input signal $(assigns\_of\_[[tr]] \ := \ [[IS_x]].fire\_[[tr]])$ to set Pr_{input} of $VarIS_{objId}$.

2. Add a new transition $t = \langle a, b, condVars \rangle$ to set T in $VarIS_{objId}$, where

$$\begin{aligned} a &= (state \equiv ok), & I(t) &= \{1\}, \\ b &= (state \equiv ok), & I_{input}(t) &= [[IS_x]].fire\_[[tr]], \\ condVars &= \varnothing, & T_{assgn}(t) &= [[assignments_{tr}]]. \end{aligned}$$

In the end, I/O-Interval Structure $VarIS_{objId}$ manages all variables and all assignments of all transitions. In RIL code, $VarIS_{objId}$ has the following form.

```
Module VarIS_[[objId]]
  STATES  state     : { ok }
          // var_1 to var_m are attribute names of class c
          [[var_1]] : { [[Value(var_1)]] }
          ...
          [[var_m]] : { [[Value(var_m)]] }

  INPUTS  // triggers to execute a particular transition with assignments
          // (i.e., variable update triggers)
          assigns_of_[[tr_1]] := [[IS_x]].fire_[[tr_1]]
          ...
          assigns_of_[[tr_n]] := [[IS_x]].fire_[[tr_n]]

  DEFINE  // conditions over variables that are used in other modules
          grdValue_[[tr_1]] := [[tr_grd(tr_1)]]
          ...
          grdValue_[[tr_n]] := [[tr_grd(tr_n)]]
  INIT    state=ok

  TRANS
  |- state=ok -- assigns_of_[[tr_1]] --> state:=ok; [[assignments_tr_1]]
              ...
              -- assigns_of_[[tr_n]] --> state:=ok; [[assignments_tr_n]]
                                     |-> state:=ok
```

Input signals `assigns_of_[[tr_i]]` are synchronously set to true when the corresponding transition `tr_i` is fired. Thus, the transitions specified in the `TRANS` compartment ensure that all necessary assignments are done in the same step as the corresponding transition `tr_i`.

Signal Call Actions. Different syntactical styles for signal call actions can be found in the literature, e.g., `/send targetObj.signalName` or `/targetObj.^signalName`. But basically, a signal call action $sigAct \in actions_{tr}$ comprises of a target object `targetObj` and a signal name `signalName`. If a signal call action is to be executed as part of the action sequence $actions_{tr}$, a corresponding signal in the Interval Structure that simulates the event queue of the target object has to be made.

For each signal call action $sigAct \in actions_{tr}$, if not yet defined, add a new output signal $sent\_[[signalName]]_{[}[targetObj]] := fire\_[[tr]]$ to $VarIS_{objId}$. These output signals guarantee that in the corresponding I/O-Interval Structure $sig\_[[signalName]]_{[}[targetObj]]_{[}[objId]]$ that simulates the event queue, a transition is made to the internal state `waiting`. As signals are asynchronous, no reply has to be waited for (as opposed to operation calls).

Operation Call Actions. In our version of State Diagrams, we abstract from local operation calls as actions, as they are assumed to take no additional time. Note that local operation calls with a specified time greater than 1 as well as synchronous operation calls to other objects are considered as *activities*, and their mapping is addressed in the corresponding sections.

After the described procedure has been performed for all transitions of $commonTR$, all trigger events, all conditions, and all actions are translated and corresponding I/O-Interval Structures for all variables and all events have been generated. But for each transition $tr \in TR$, we still have to establish synchronization of the 'transition parent' state with the other affected states determined by the set $ExitEnter_{tr} \cup ExitOnly_{tr} \cup EnterOnly_{tr} \setminus \{parent(tr_{src}(tr))\}$. We present this step by step with several cases.

Transitions both Exiting and Entering direct Substates. We first consider all Xor-states $x \in ExitEnter_{tr} \cap Xor \setminus \{parent(tr_{src}(tr))\}$. For all $val \in substates(x) \cap ExitStates_{tr}$, we add transitions $t_{val} = \langle a_{val}, b, condVars\rangle$ to the set of transitions T in IS_x, where

$$\begin{aligned} a_{val} &= \langle (state \equiv [[val]]), (activated \equiv true)\rangle,\\ b &= \langle (state \equiv [[substates(x) \cap EnterStates_{tr}]]), (activated \equiv true)\rangle, \text{ and}\\ condVars &= \{[[parent(tr_{src}(tr))]].fire\_[[tr]]\}. \end{aligned}$$

For all these transitions t_{val}, we set

$$\begin{aligned} I(t_{val}) &= \{1\},\\ I_{input}(t_{val}) &= ([[parent(tr_{src}(tr))]].fire\_[[tr]] \equiv true),\\ T_{assgn}(t_{val}) &= \xi_{act}. \end{aligned}$$

For And-states $x \in ExitEnter_{tr} \cap And$ (by definition, these x cannot be $parent(tr_{src}(tr))$), we add a transition $t = \langle a, b, condVars\rangle$ to the set of transitions T in IS_x, where

$$\begin{aligned} a &= \langle (activated \equiv true)\rangle,\\ b &= \langle (activated \equiv true)\rangle,\\ condVars &= \{[[parent(tr_{src}(tr))]].fire\_[[tr]]\} \cup \{[[s]].state \mid s \in substates(x)\}. \end{aligned}$$

The latter set is necessary to keep track of the status of all (Xor-)substates. $I(t)$ and $T_{assgn}(t)$ are defined as above, whereas $I_{input}(t)$ must be defined considering the completion event:

$$\begin{aligned} I_{input}(t) = \; & ([[parent(tr_{src}(tr))]].fire\_[[tr]] \equiv true) \wedge \\ & \textstyle\bigwedge_{s \in substates(x)} \langle ([[s]].state \equiv [[final(s)]]), ([[s]].activated \equiv true) \rangle. \end{aligned}$$

Transitions only Exiting a Direct Substate. For Xor-states $x \in ExitOnly_{tr} \cap Xor \setminus \{parent(tr_{src}(tr))\}$ and for all $val \in substates(x) \cap ExitStates_{tr}$, we add transitions $t_{val} = \langle a_{val}, b, condVars \rangle$ to the set of transitions T in IS_x, where

$$\begin{aligned} a_{val} \quad & = \langle (state \equiv [[val]]), (activated \equiv true) \rangle, \\ b \quad & = \langle (state \equiv [[val]]), (activated \equiv false) \rangle, \text{ and} \\ condVars \quad & = \{[[parent(tr_{src}(tr))]].fire\_[[tr]]\}. \end{aligned}$$

Note here that the current value of variable $state$ is retained. This information might be useful when re-entering the exited state via a history state – however, history states are not yet regarded in this translation.[5] As before, we set

$$\begin{aligned} I(t_{val}) \quad & = \{1\}, \\ I_{input}(t_{val}) \quad & = ([[parent(tr_{src}(tr))]].fire\_[[tr]] \equiv true), \\ T_{assgn}(t_{val}) \quad & = \xi_{act}. \end{aligned}$$

For And-states $x \in ExitOnly_{tr} \cap And$, we take the same approach as above for And-states $\in ExitEnter_{tr}$, but now replace the value of variable $activated$ by $\langle (activated \equiv false) \rangle$ for the destination state b.

Transitions only Entering a Direct Substate. For Xor-states $x \in EnterOnly_{tr} \cap Xor$[6] and for all $val \in substates(x)$, we add transitions $t_{val} = \langle a_{val}, b, condVars \rangle$ to the set of transitions T in IS_x, where

$$\begin{aligned} a_{val} \quad & = \langle (state \equiv [[val]]), (activated \equiv false) \rangle, \\ b \quad & = \langle (state \equiv [[substates(x) \cap EnterStates_{tr}]]), (activated \equiv true) \rangle, \text{ and} \\ condVars \quad & = \{[[parent(tr_{src}(tr))]].fire\_[[tr]]\}. \end{aligned}$$

We set

$$\begin{aligned} I(t_{val}) \quad & = \{1\}, \\ I_{input}(t_{val}) \quad & = ([[parent(tr_{src}(tr))]].fire\_[[tr]] \equiv true), \\ T_{assgn}(t_{val}) \quad & = \xi_{act}. \end{aligned}$$

[5]The main challenge here is to determine the different cases of history states in combination with default transitions and other common transitions. These cases have all to be treated differently, which makes the mapping to I/O-Interval Structures more complex. However, as there are already dedicated states in the target I/O-Interval Structures that indicate whether a state is activated, no additional states have to be introduced to the I/O Interval Structures when history states are considered.

[6]Note that by definition for all $x \in EnterOnly_{tr}$ it holds $x \neq parent(tr_{src}(tr))$.

For And-states $x \in EnterOnly_{tr} \cap And$, we add a transition $t = \langle a, b, condVars \rangle$ to the set of transitions T in IS_x, where $a = \langle (activated \equiv false) \rangle$, $b = \langle (activated \equiv true) \rangle$, and $condVars = Var(tr_{evt}(tr)) \cup Var(tr_{grd}(tr))$. Finally, we set

$$\begin{aligned} I(t) &= \{1\}, \\ I_{input}(t) &= ([[parent(tr_{src}(tr))]].fire\_[[tr]] \equiv true), \\ T_{assgn}(t) &= \xi_{act}. \end{aligned}$$

5.4.2.2 Mapping of Transitions with Elapsed Time Events

Transitions with elapsed time events do not have conditions and actions. The UML standard syntax is '`after tm`', where `tm` is a non-negative natural number greater than 1. As we handle local do-activities of simple states by elapsed time events of the form `after [minTime,maxTime]`, the syntax is extended, such that timing intervals are now allowed in contrast to standard UML. As we abstract from physical time units, it is important that `tm` is specified in relation to the basic time unit that is assumed for performing a run-to-completion step. For example, if the basic time unit is milliseconds, `after 1000` refers to an elapsed time of 1 second.

For $tr \in elapsedTimeTR$, let $minTime_{tr}$ and $maxTime_{tr}$ be the specified timing interval values in $tr_{evt}(tr)$. The I/O-Interval Structure IS_x with $x = parent(tr_{src}(tr))$ gets an additional counter variable for counting the number of steps passed since entering state $tr_{src}(tr)$. Note that each I/O-Interval Structure does only need at most as many counter variables as the number of concurrent substates, even if several transitions with an elapsed time event are specified. In particular, for $x \in Xor$, at most one counter variable is necessary. I/O-Interval Structures for composite states $y \in ExitStates_{tr} \cup EnterStates_{tr}$ that are indirectly affected by transitions with elapsed time events (i.e., $y \neq parent(tr_{src}(tr))$) get an additional input variable to synchronize with IS_x.

In the remainder, we concentrate on mapping transitions for $x \in Xor$. A mapping for $x \in And$ can be easily derived by iterating through all concurrent substates.

- **Introducing counter variables.** For all $x \in Xor$, if there is a transition $tr \in elapsedTimeTR$ with $tr_{src}(tr) \in substates(x)$, we add a new variable $count$ to Pr in IS_x with
$$count : RANGE[0..maxTm_x],$$
where $maxTm_x$ is the maximum of all specified timing values of transitions with elapsed time events relevant for x, i.e.,
$$\begin{aligned} maxTm_x = max\{maxTime_{tr} \mid\ & tr \in elapsedTimeTR \wedge \\ & \exists y \in substates(x) \text{ with } y = tr_{src}(tr)\}. \end{aligned}$$
Variable $count$ is initialized by adding $count$ to s_0, i.e., $s_0 := s_0 \ \wedge\ (count \equiv 0)$.

- **Adding input variables.** For all transitions $tr \in elapsedTimeTR$ and for all states $y \in ExitEnter_{tr} \cup ExitOnly_{tr} \cup EnterOnly_{tr}$ with $y \neq parent(tr_{src}(tr))$, we add an input variable $[[parent(tr_{src}(tr))]].count$ to Pr_{input} in IS_y.

We can now add transitions to the I/O-Interval Structures, using the variables defined above. For all $tr \in elapsedTimeTR$, we add the following transitions.

- For state $x = parent(tr_{src}(tr))$, we add four transitions t_1, t_2, t_3, t_4 to the set T of transitions in IS_x. Let $a = \langle(state \equiv [[tr_{src}(tr)]]), (activated \equiv true)\rangle$ denote the transition source state, and $b = \langle(state \equiv [[tr_{dst}(tr)]]), (activated \equiv true)\rangle$ the destination state. We set

$$t_1 = \langle a, a, \{count\}\rangle,\ t_2 = \langle a, a, \{count\}\rangle,$$
$$t_3 = \langle a, b, \{count\}\rangle \quad t_4 = \langle a, b, \{count\}\rangle.$$

  and the corresponding transition-related function values

$$\begin{aligned} I(t_1) &= \{1\}, & I(t_2) &= \{1\}, \\ I_{input}(t_1) &= (count < [[minTime_{tr}]]), & I_{input}(t_2) &= (\ (count >= [[minTime_{tr}]]) \\ & & &\quad \wedge (count < [[maxTime_{tr}]])), \\ T_{assgn}(t_1) &= \langle(count := count + 1)\rangle, & T_{assgn}(t_2) &= \langle(count := count + 1)\rangle, \end{aligned}$$

$$\begin{aligned} I(t_3) &= \{1\}, & I(t_4) &= \{1\}, \\ I_{input}(t_3) &= (\ (count >= [[minTime_{tr}]]) & I_{input}(t_4) &= (count = [[maxTime_{tr}]]), \\ &\quad \wedge (count < [[maxTime_{tr}]])), & & \\ T_{assgn}(t_3) &= actions_{tr}; \langle(count := 0)\rangle, & T_{assgn}(t_4) &= actions_{tr}; \langle(count := 0)\rangle. \end{aligned}$$

  Transition t_1 simply increments the counter variable while remaining in the source state until the minimal time is reached. Then, it can non-deterministically be chosen between firing t_2 or t_3. Transition t_2 remains in the source state and increments the counter, while t_3 changes to the target state and resets the counter to 0. Transition t_4 guarantees that the targetstate is entered when the maximum ealpsed time is reached.

- For all states $x \in ExitEnter_{tr} \setminus \{parent(tr_{src}(tr))\}$ and for all substates $val \in substates(x) \cap ExitStates_{tr}$, we add transitions $t_{val} = \langle a_{val}, b, condVars\rangle$ to the set of transitions T in IS_x, where

$$\begin{aligned} a_{val} &= \langle(state \equiv [[val]]), (activated \equiv true)\rangle, \\ b &= \langle(state \equiv [[substates(x) \cap EnterStates_{tr}]]), (activated \equiv true)\rangle, \text{ and} \\ condVars &= Var(tr_{evt}(tr)) \cup Var(tr_{grd}(tr)). \end{aligned}$$

  We set the following transition-related function values

$$\begin{aligned} I(t_{val}) &= \{1\}, \\ I_{input}(t_{val}) &= ([[parent(tr_{src}(tr))]].count == maxTime_{tr}), \\ T_{assgn}(t_{val}) &= \xi_{act}. \end{aligned}$$

- For all states $x \in ExitOnly_{tr} \setminus \{parent(tr_{src}(tr))\}$ and for all substates $val \in substates(x) \cap ExitStates_{tr}$, we add transitions $t_{val} = \langle a_{val}, b, condVars \rangle$ to the set of transitions T in IS_x, where

$$\begin{aligned} a_{val} &= \langle (state \equiv [[val]]), (activated \equiv true) \rangle, \\ b &= \langle (state \equiv [[val]]), (activated \equiv false) \rangle, \text{ and} \\ condVars &= Var(tr_{evt}(tr)) \cup Var(tr_{grd}(tr)). \end{aligned}$$

  As before, we set

$$I(t_{val}) = \{1\},\ I_{input}(t_{val}) = (tr_{evt}(tr)\ \wedge\ tr_{grd}(tr)),\ T_{assgn}(t_{val}) = \xi_{act}.$$

- For all states $x \in EnterOnly_{tr}$ and for all $val \in substates(x)$, we add transitions $t_{val} = \langle a_{val}, b, condVars \rangle$ to the set of transitions T in IS_x, where

$$\begin{aligned} a_{val} &= \langle (state \equiv [[val]]), (activated \equiv false) \rangle, \\ b &= \langle (state \equiv [[substates(x) \cap EnterStates_{tr}]]), (activated \equiv true) \rangle, \text{ and} \\ condVars &= Var(tr_{evt}(tr)) \cup Var(tr_{grd}(tr)). \end{aligned}$$

  We set

$$I(t_{val}) = \{1\},\ I_{input}(t_{val}) = (tr_{evt}(tr)\ \wedge\ tr_{grd}(tr)),\ T_{assgn}(t_{val}) = \xi_{act}.$$

Assume for the moment that all transitions for T are yet defined in all I/O-Interval Structures IS_x with a counter variable *count* as introduced above. In all these structures, we have to add the assignment $count := 0$ to reset the counter in all transitions that change states, i.e., for all $t \in \{r \in T \mid r[1] \neq r[2] \wedge (count := count + 1) \notin T_{assgn}(r)\}$, we add $T_{assgn}(t) = T_{assgn}(t) +_{append} \langle (count := 0) \rangle$,

5.4.2.3 Conflicting Transitions

Conflicting Transitions might occur, when there are two activated ancestrally related states, say s_1 and s_2, which each have an outgoing transition with the same triggering event evt and valid conditions (i.e., the transition guards both evaluate to true). In this case, the UML standard specifies that the transition with the *lower source state* has precedence and is taken. However, this does not yet solve the case $s_1 = s_2$. For $s_1 = s_2$, we simply assume non-deterministic behavior, i.e., a non-deterministic choice is made.

But for $s_1 \neq s_2$, we have to add some additional conditions in our generated I/O-Interval Structures to follow the precedence concept of UML State Diagrams. First, for a given event $e \in EVT$ we define the help sets

$$\begin{aligned} trans_e &= \{tr \in TR \mid e = tr_{evt}(tr)\}, \\ sourceStates_e &= \{tr_{src}(tr) \in S | tr \in trans_e\}, \text{ and} \\ parentStates_{e,s} &= sourceStates_e \cap parents^{+}(s) \text{for each } s \in sourceStates_e. \end{aligned}$$

Function $parents^{+}(s)$ gives the transitive superstates of a state s, excluding s. For each event e and state s with $parentStates_{e,s} \neq \emptyset$, the corresponding transition conditions $fire_{[}[tr]]$ of all $s' \in parentStates(e, s)$ have to be equipped with an additional negated condition, i.e., $fire_{[}[tr]] := fire_{[}[tr]] \ \wedge \ \neg[[tr_{grd}(s)]]$, to prevent the superstates from firing their transitions.

5.5 Contributions of the Chapter

In this chapter, a UML State Diagram variant has been defined that restricts on a subset of standard UML State Diagram model elements. In particular, rarely used pseudostates and critical interlevel transitions are omitted.

The regarded subset of UML State Diagrams has then been extended in the sense that timing specifications attached to operations – as specified in Class Diagrams – are considered as estimated activity times in State Diagrams. The execution semantics of this timed State Diagram variant is defined by a translation to I/O-Interval Structures over discrete time.

Chapter 6

Modeling Manufacturing Systems with MFERT

As complexity rises,
precise statements lose meaning,
and meaningful statements lose precision.
—Lotfi Zadeh

MFERT constitutes a generic approach for specification and implementation of planning and control assignments in manufacturing processes [DW93, Sch96, DW97]. It has been applied in different projects with industrial partners and received the German science award of logistics. Basically, MFERT builts upon concepts of Petri Nets, i.e., the structure of an MFERT model is a bipartite graph of nodes that represent either production processes or storages for production elements. *Production Element Nodes* (PENs) are used to model logical storages of material and resources. The can be compared with places in Petri Nets. *Production Process Nodes* (PPNs) represent logical locations where material is transformed. PPNs are comparable to transitions in Petri Nets. *Directed edges* between nodes denote the flow of production elements.

Example. Consider the production flow illustrated in Figure 6.1 that models the flow of production items of the case study presented in Section 1.2. Primary input is modeled by PENs `RawEngines` and `RawShafts`. Corresponding processes are used to supply these production items into PENs `EnginesSupplied` and `ShaftsSupplied`.

Different processes model the transformation of production items in the stations *mill*, *drill*, and *wash*, i.e., `Milling`, `Drilling`, and `Washing`. Input and output buffers of stations are modeled by PENs named `ItemsBeforeMill`, `ItemsAfterMill`, `ItemsBeforeDrill`, `Items-AfterDrill`, `ItemsBeforeWash`, and `ItemsAfterWash`.

Transports between stations are modeled by transporting processes named `Transporting-ToMill`, `TransportingToDrill`, and `TransportingToWash`. A fourth transporting process is used to take production items to the output station. To perform a transport, an item and an automated guided vehicle (AGV) is needed as an input. In the example, AGVs are modeled as a production resource by means of a PEN. For performing a transport, the corresponding process allocates an AGV from that PEN (i.e., it takes one AGV from that PEN), and after the process has finished, it vacates that AGV (i.e., the AGV is re-entering the PEN).

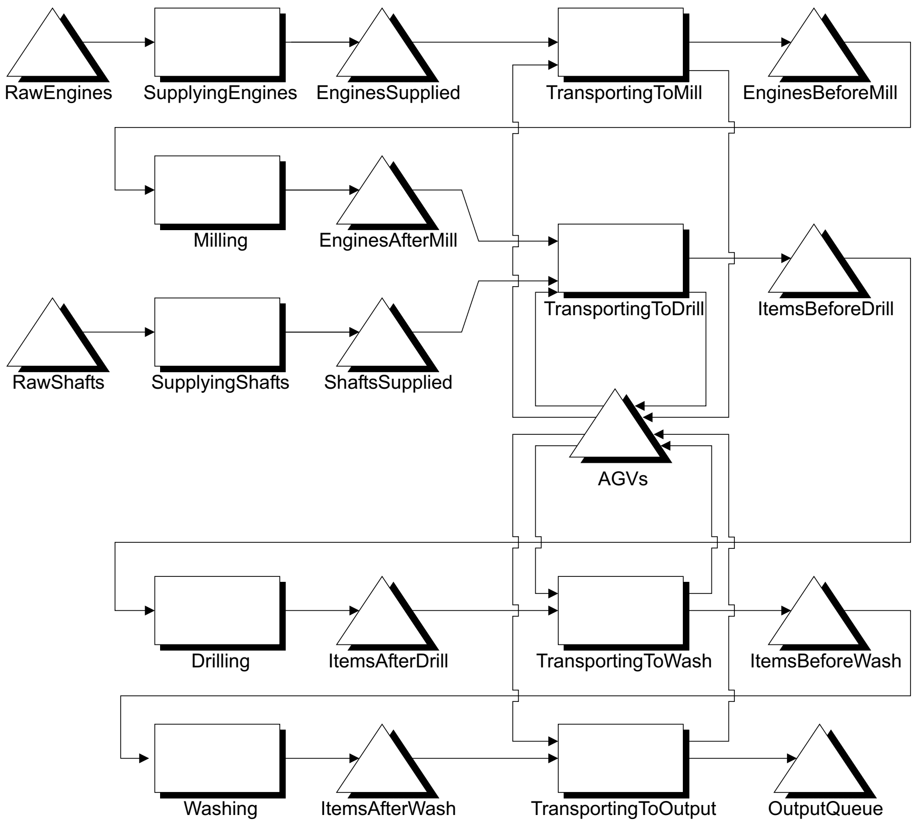


Figure 6.1: MFERT Graph of the Case Study

The *dynamics* of an MFERT model is defined by local functions associated with the nodes. These local functions are also called *local managers* or *agents*. An instantiation of an MFERT model is therefore seen as a distributed system of interacting ‘agents’. Different approaches are suitable to define local functions in this context, for instance, Quintanilla formally defines a graphical notation called *interaction diagrams* [Qui01]. However, timing aspects are not regarded in that approach.

We focus in this thesis on finite state machines (FSMs) and the timed UML State Diagram variant that was introduced in Chapter 5 to define the local functionality of MFERT nodes – more specifically, we employ additional restrictions on the set of actions and activities defined in the timed UML State Diagram variant. In the context of MFERT, we only regard actions and activities that denote (a) requests of PPNs to put and get elements to and from PENs, (b) actual transfers of production items between MFERT nodes, and (c) local transformation activities that have a notable duration.

For example, an FSM assigned to node `TransportingToDrill` in Figure 6.1 may specify that such a transport requests a resource (i.e., an AGV) and an item to transport, i.e., either a shaft or an engine. Additionally, the local activity of transporting an item is required to be executed within 20 to 40 time units. This might be a well-known duration derived from the concrete setting of the physical system or an estimated duration that is only assumed at time of modeling. If we checked system properties based on this assumption, they are only valid if the estimated duration is really met in the running system.

In contrast to PPNs that actually control the production flow via their associated functions, PENs are organized in are rather passive and wait for requests to react on. A PEN is mainly used to store incoming and outgoing production elements in queues and shifts elements from the input to the output queue with a certain delay.

In the following three sections, we describe the graphical notation, the formal model, and the dynamic semantics of MFERT. In Section 6.4, we then outline a UML Profile for MFERT models. This profile is the basis for applying OCL constraints to UML-based MFERT models, which will be presented in Chapter 8.

6.1 MFERT Graphs

Graphically, two types of nodes are distinguished. An annotated triangle represents a PEN, an annotated rectangle stands for a PPN. The annotation indicates the unique name of the node. Directed edges from PENs to PPNs denote a material or resource input into a production process, while directed edges from PPNs to PENs represent the output flow after completion of production processes.

Edges are only allowed between nodes of different types (i.e. triangles and rectangles), so that the resulting MFERT graph is bipartite. MFERT graphs show both a static and a dynamic view of a manufacturing system. On the one hand, the nodes statically represent the participating production processes and element storages. On the other hand, edges represent the dynamic flow of production elements (i.e., material and resources) within the manufacturing system.

In the original work on MFERT, annotations by means of *time bars* are attached to PENs and PPNs to illustrate incoming and outgoing events. In terms of UML, this basically conforms to an interface specification of operations and signals that can be handled by a PEN or PPN, respectively. The particular events drawn on the arrows only informally show possible in- and outputs, i.e., they do not put any restrictions on their order. Therefore, we omit the graphical element of time bars in our description of MFERT and instead assume that the interface of PPNs and PENs is specified in a manner similar to UML. This is an important aspect when we are going to develop a UML Profile for MFERT in Section 6.4.

6.2 Formal MFERT Model

The type system used in our MFERT definition is similar to the one of Coloured Petri Nets (CP-nets) [Jen91]. The tokens that are running through the system are production elements of a certain data type. A data type D is a tuple of value sets. For instance, a data type $EngineOrder$ may be defined by tuples of form

$$\langle kind, kw, turbo, id\rangle \subseteq \{petrol, diesel\} \times \{40, 56, 70\} \times Boolean \times Integer\ .$$

We assume that standard types, such as $Integer$ or $Boolean$, together with their common operations are well-defined. In the following, let DT refer to the set of all data types defined in the system.

Expressions, Variables, and Sequences In general, complex expressions are combined from primitives (i.e., variables or constants) and subexpressions by use of functions and operations. We do not give a concrete expression syntax here; we just assume that such a syntax exists together with a well-defined semantics, such that it is possible to talk about

- the type of a variable v, denoted by $Type(v)$,
- the value of a variable v, denoted by $Value(v)$,
- the type of an expression $expr$, denoted by $Type(expr)$,
- the set of variables in an expression $expr$, denoted by $Var(expr)$. This set only contains the free variables, i.e., variables that are not bound internally in $expr$, e.g., by a local definition.

A sequence q over a data type $D \in DT$ is a function

$$q \in \{f \mid f : \mathbb{N} \to D \cup \{\epsilon\}, D \in DT\} .$$

We denote a sequence q by an enumeration $q = \langle\langle a_1, a_2, \dots\rangle\rangle$, with $a_i \in D$. We can stop the enumeration if there in an index i such that all following sequence elements are equal to the empty word ϵ. We denote the set of all well-defined sequences by

$$Seq_D = \{f \mid f : \mathbb{N} \to D \cup \{\epsilon\}, \forall i \in \mathbb{N} : (f(i) = \epsilon) \Rightarrow \forall j \geq i : f(j) = \epsilon\} .$$

We assume that the common operations on sequences are well-known, e.g., concatenation of sequences, accessing the i-th element, deleting at i-th position, etc. In particular, we denote by $Type(q)$ the unique data type $D \in DT$ over which a sequence q is defined. In the following, let

$$Seq_{DT} = \bigcup_{D \in DT} Seq_D .$$

With these preliminaries, we can define the static structure of an MFERT model as follows.

Definition 6.1 *An* **MFERT model** *is a tuple*

$$\begin{aligned} G \overset{def}{=} \langle\ & PE, PP, E, DT, C, In, Out, Cap_{In}, Cap_{Out}, \\ & Time, FSM, M_{FSM}, Init_{In}, Init_{Out}\ \rangle, \end{aligned}$$

where

(i) PE *is a finite set of ProductionElementNodes,*

(ii) PP *is a finite set of ProductionProcessNodes,* $PP \cap PE = \varnothing$.

(iii) $E \subseteq (PP \times PE) \cup (PE \times PP)$ *is a finite set of directed edges between PENs and PPNs. The edges represent the element flow between nodes.*

(iv) *DT is a finite set of Data Types. A data type $D = \langle VS_{D,1}, VS_{D,2}, \ldots, VS_{D,n}\rangle \in DT$ is a tuple of – possibly infinite – value sets $VS_{D,i} \neq \varnothing, 1 \leq i \leq n$.*

(v) *$C : PE \rightarrow DT$ is the mapping that gives each $pe \in PE$ a data type.*

(vi) *$In : PE \rightarrow Seq_{DT}$ is the mapping that associates an input sequence to each $pe \in PE$. It holds $\forall pe \in PE : Type(In(pe)) = C(pe)$. In the following, let $In_{PE} = \bigcup_{pe\in PE} In(pe)$.*

(vii) *$Out : PE \rightarrow Seq_{DT}$ is the mapping that associates an output sequence to each $pe \in PE$. It holds $\forall pe \in PE : Type(Out(pe)) = C(pe)$. In the following, let $Out_{PE} = \bigcup_{pe\in PE} Out(pe)$.*

(viii) *$Cap_{In}, Cap_{Out} : PE \rightarrow \mathbb{N}$ give each input and output sequence a maximal capacity.*

(ix) *$Time : PE \rightarrow \mathbb{N}_0$ is the mapping that associates a delay time to each $pe \in PE$.*

(x) *$FSM = \bigcup_{pp\in PP} fsm_{pp}$ is a set of finite state machines, $|FSM| = |PP|$.*

In order to be able to talk about particular elements of FSMs, we give a structural definition in Appendix A.[1] We assume that it is possible to talk about a set of 'proper states' of an $fsm \in FSM$, denoted by $STATES_{fsm}$, which is taken to describe the current status of an FSM. In the following, let $STATES_{FSM} = \bigcup_{fsm\in FSM} STATES_{fsm}$.

Restrictions: *Note that we restrict the type of actions in state transitions of FSMs; actions that consume and actions that produce elements must not appear in the same transition.*

(xi) *$M_{FSM} : PP \rightarrow FSM$ is the bijective mapping that associates a finite state machine $fsm \in FSM$ to each production process $pp \in PP$.*

(xii) *$Init_{In} : PE \rightarrow Seq_{DT}$ is an initialization function that represents the initial marking of the input sequence of each $pe \in PE$. $Init_{In}$ is defined from PE into Sequences such that*

$$\forall pe \in PE : Type(Init_{In}(pe)) = C(pe) \wedge Var(Init_{In}(pe)) = \varnothing\ .$$

(xiii) *$Init_{Out} : PE \rightarrow Seq_{DT}$ is an initialization function that represents the initial marking of the output sequences of PENs. $Init_{Out}$ is defined from PE into Sequences such that*

$$\forall pe \in PE : Type(Init_{Out}(pe)) = C(pe) \wedge Var(Init_{Out}(pe)) = \varnothing\ .$$

Note that this is a general functional definition of MFERT. In Section 6.4, we are going to define a UML Profile for the domain-specific structural elements of MFERT, i.e., sets PP, PE, and E, to have a graphical notation for modeling purposes. Additionally, we employ the timed UML State Diagram notation as introduced in Chapter 5 for the FSMs in MFERT.

We define the overall runtime status of an instantiated MFERT model by means of the *configuration* the current values in the input and output sequences of PENs and the state configurations of FSMs as follows.

[1]Though, that definition does not prescribe a finite state machine formalism with a particular execution semantics, as this general MFERT definition abstracts from internal behavior of PPNs.

Definition 6.2 *In an MFERT model with the finite sets PE, PP, and FSM, let*

$$configuration : FSM \rightarrow \mathcal{P}(STATES_{FSM})$$

be the function that returns the current configuration of a finite state machine. An MFERT configuration *is described by a tuple*

$$\Big\langle \; Out_{PE}, In_{PE}, \bigcup_{fsm \in FSM} configuration(fsm) \; \Big\rangle \; .$$

6.3 Dynamic Semantics of MFERT

Concerning the micro view semantics, i.e., the execution semantics of single processes, the general MFERT definition in [Sch96] makes no particular assumptions. Also the general MFERT definition 6.1 of this thesis so far has just an abstract notion of finite state machines $FSMs$ to describe the behavior of production processes, but for an executable or verifiable model, we need a particular execution semantics. We therefore take the timed UML State Diagram variant as the FSM formalism for MFERT. Recall that we have defined a mapping to I/O-Interval Structures for these timed UML State Diagrams, such that the execution semantics is given by means of the formal notion of *runs* of I/O-Interval Structures (cf. Definition 3.13 on page 75).

Concerning the macro view of communication between different MFERT nodes, there are different execution semantics of MFERT identified, i.e., synchronous, asynchronous, and simulation semantics. As we focus on synchronous semantics in the remainder, we briefly discuss the other two in Section 6.3.5.

The dynamic semantics of *synchronous* MFERT models is best described by an abstract interpreter for PPNs and PENs, respectively. For PPNs, an interpreter controls the execution of the FSMs. For PENs, the corresponding interpreter is cyclically shifting production elements from input to output sequences.

6.3.1 Production Process Nodes

We assume that each PPN has its own thread of control and runs independently of a global signal. A PPN works along the lines of an abstract interpreter as it is described in Figure 6.2. Each interpreter cycle starts with selection of an applicable transition. After the selection phase, first some checks on adjacent PENs are necessary. Note here, that some actions associated with the selected state transition have an effect on adjacent PENs, e.g., by consuming production elements from preceding nodes. It must be ensured that this is a valid operation. Finally, the transition is fired and a new state is entered after the delay time determined by the selected transition. A delay time greater than 1 usually represents the time that is necessary for a particular production step.

Given a state s of an FSM, `DetermineTransition`(s) chooses a transition $t \in T$ out of the set of applicable transitions. We do not specify how this transition is going to be chosen, as this is defined in the dynamic semantics of the underlying state machine model, but we assume that `Conditions`(t) evaluates to true at the time of chosing t.

```
currentState = $s_0$                                  // $s_0$ is the initial state of the FSM
while true {
    $t$ = DetermineTransition(currentState) $\in Tr \cup \{\epsilon\}$          // selection phase
    if $(t \neq \epsilon)$ {
        grant = true
        parallel do {                                   // checking phase
            $C = \{\ (D, i) \mid \forall v \in Var(ConsumeActions(t)) : D = Type(v) \wedge i = Value(v)\}$
            if $(C \neq \varnothing)$ { grant = SendConsumeRequests($C$) }
        } || {
            $P = \{\ (D, i) \mid \forall v \in Var(ProduceActions(t)) : D = Type(v) \wedge i = Value(v)\}$
            if $(P \neq \varnothing)$ { grant = SendProduceRequests($P$) }
        }

        if (grant) {
            Execute(Actions($t$), Delay($t$) - 1)               // execution phase
            currentState = NextState($t$)
        }
    }
}
```

Figure 6.2: Abstract Interpreter for PPNs

Due to our definition of finite state machines in Appendix A, at least one of the sets C and P is empty in each cycle. `SendConsumeRequests`(C) and `SendProduceRequests`(P) take C resp. P to send request messages to all corresponding adjacent PENs. These messages are sent as a multicast with the same time stamp. We further require that these request messages are sent synchronously, i.e., the interpreter waits until all replies are received. In the following, we assume that all requests can be immediately answered by all PENs in one time step.

Replies are either acknowledgments or denials. `SendConsumeRequests`(C) and `SendProduceRequests`(P) return a Boolean value, indicating whether access to all corresponding adjacent PENs is granted or not. Only if all requested PENs have replied with an acknowledgement, `Execute` (`Actions`(t), `Delay`(t) -1) is called. That function executes the actions associated with t and returns after `Delay`(t) - 1 time units to proceed.

Concerning the current state configuration, we have to consider the evolution of time. We therefore define the following timing restrictions for the interpreter cycle of PPNs.

1. The selection phase takes no time.

2. The checks for valid actions take one time step (send now and receive answers one step later).

3. The execution phase takes as many time steps as specified in the selected transition, minus one time step due to the checks. This means, that the new state is actually entered after the corresponding number of time steps has passed.

The latter condition implies that the FSM remains in its current configuration up to the end of the delay time and enters the new configuration exactly at that time when the specified time steps have passed.

6.3.2 Production Element Nodes

The dynamic semantics of a PEN $pe \in PE$ is described by the following abstract interpreter illustrated in Figure 6.3.

For shifting elements, we require exclusive access to $In(pe)$ and $Out(pe)$, as it is indicated by the functions `Block()` and `Unblock()`. Queries and manipulations on $In(pe)$ and $Out(pe)$ invoked by messages from PPNs are handled independently of the main cycle, but are also performed with exclusive access to avoid conflicts.

```
while true {
    Wait($Time(pe)$ - 1)
    Block($Out(pe)$)                                  // exclusive access while shifting elements
    Block($In(pe)$)
    $Out(pe)$ = Concat($In(pe)$, $Out(pe)$)          // shifting
    $In(pe)$ = ∅
    Unblock($In(pe)$)
    Unblock($Out(pe)$)
}
```

Figure 6.3: Abstract Interpreter for PENs

Concerning the current state configuration, we here only have to define the timing restriction that blocking, unblocking, and shifting takes place within 1 time step. This implies that the PEN remains in its current configuration up to the end of the `Wait()` command and enters the new configuration exactly after the specified time $Time(pe)$ has passed.

6.3.3 Message Passing

Due to the required implicit multicast message passing mechanism in the PPN interpreter cycle, conflicts like dead- or livelocks might occur when different PPNs send messages to common PENs at the same point of time. In order to avoid such conflicts, we need to define an input channel $MsgQueue_{pe}$ for each $pe \in PEN$ that takes incoming messages from adjacent PPNs. Formally, $MsgQueue_{pe}$ is a sequence over a data type Msg that is defined by

$$Msg = \underbrace{PPN}_{sender} \times \underbrace{\{reqCons, reqProd, consume, produce\}}_{message\ type} \times \underbrace{\mathbb{N}_0}_{timeID} \times \underbrace{DT \times \mathbb{N}_0}_{content} .$$

In the following, we may also write $Msg(pp, pe, timestamp)$ for an element of Msg, as the parameters $pp \in PP$ and $timestamp \in \mathbb{N}_0$ uniquely determine an incoming message for $pe \in PE$. The message type can be derived, as only one message may be sent at each point of time from pp to pe, and $DT \times \mathbb{N}_0$ denotes the actual message content, i.e., the number of production elements that are to be consumed or produced. The following assumptions are of particular importance for the message passing model:

1. We require that there is at most one action in each state transition that invokes a message.

2. PPNs send synchronous messages to PENs, i.e., each PPN waits for corresponding replies.

3. Messages sent from PPNs immediately appear in each of the PEN's input message queue $MsgQueue_{pe}, pe \in PE$.

4. We assume that all incoming messages are handled immediately after they are inserted into $MsgQueue_{pe}$, i.e., an answer to a request (for consumption or production), consume, or produce message is sent back exactly one time step later. Note that it can happen that shifting and replying to messages must happen in the same step. We therefore define a priority scheme in Section 6.3.4.

It follows that at each point of time there can be at most one message from each adjacent PPN in each $MsgQueue_{pe}$. This limits the size $|MsgQueue_{pe}|$ of the message queues to the number of adjacent PPNs.

6.3.4 Conflict Resolution in PENs

Assuming that messages are immediately answered by PENs, several actions might have to be handled at a single point of time. Among these actions, we have to define a priority in order to have a well-defined execution semantics. We define for a given point of time t:

1. If at the current time point the execution of `Wait`($Time(pe)$ -1) ends, shifting elements from the input queue to the output queue has highest priority and is executed first.

2. Then, if there are produce messages in $MsgQueue_{pe}$, these messages are handled in an arbitrary order.

3. Then, if there are consume messages in $MsgQueue_{pe}$, these messages are handled in an arbitrary order.

4. Then, if there are request messages for production in $MsgQueue_{pe}$, these messages are handled in an arbitrary order.

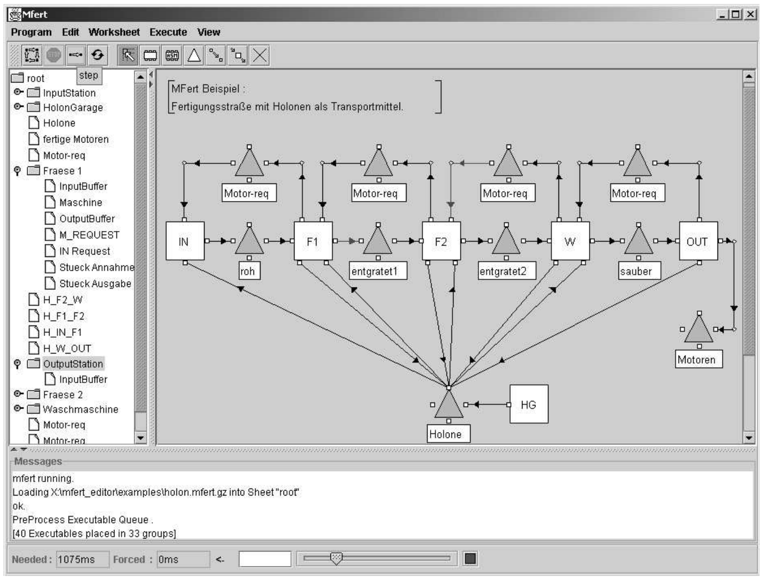
5. Then, if there are request messages for consumption in $MsgQueue_{pe}$, these messages are handled in an arbitrary order.

6.3.5 Simulation Implementation

The *simulation implementation* of MFERT focuses on supervision of the capacities in PENs [Zab03]. Though no formal semantics are defined, the models are executed in a very similar way. The nodes act on a global signal, i.e., all nodes select a transition and communicate at the same time with their adjacent nodes.

A visual interface for specification and simulation of MFERT models is presented in [DDF$^+$02]. In this tool, a notion of hierarchy has been introduced to MFERT to model different layers of abstraction. This means that a rectangle does not only represent a PPN, but can also be seen as a placeholder for another MFERT (sub)model with particular in- and output restrictions.

In the simulation environment, active transitions are marked red in each step, and underflow in output sequences as well as overflow in input sequences is illustrated (cf. Figure 6.4).



mfert running.
Loading X:\mfert_editor\examples\holon.mfert.gz into Sheet "root"
ok.
PreProcess Executable Queue .
[40 Executables placed in 33 groups]

Needed : 1075ms Forced : 0ms <-

Figure 6.4: Simulation of an MFERT Model

6.4 A UML Profile for MFERT

Modeling the static and dynamic aspects w.r.t. material and resource flow of manufacturing systems can be done by UML Class Diagrams. We here introduce stereotypes that represent production processes, storages, and element flow by the virtual metamodel shown in Figure 6.5, where

1. a `ProductionDataType` defines a tuple of data types. Only query and constructor operations are allowed for production data types, which may only aggregate or be composed of `DataTypes` or `ProductionDataTypes`.

2. The `ElementList` stereotype represents a parameterized interface that provides certain operations to manage lists with elements of a certain `ProductionDataType`. We assume that appropriate operations for lists are specified.

3. `MFERTNode` is the abstract superclass of `ProductionProcessNode` and `ProductionElementNode`. MFERT nodes may only inherit from other MFERT nodes of the same kind. Associations between two MFERT nodes have to be modeled using `ElementFlow` associations. There is at most one relationship between each pair of MFERT nodes, which is either a generalization or an `ElementFlow` association. If the relationship is a generalization, the participating MFERTNodes must be of the same subclass, i.e. either ProductionProcessNodes or ProductionElementNodes.

4. An MFERTNode may not have an association among itself.

5. We do not allow aggregation and composition of MFERTNodes.

6. `ProductionProcessNodes` (PPNs) consume from and send production elements to `ProductionElementNodes`. Each PPN has its own thread of control, i.e., the instances are active objects.

7. `ProductionElementNodes` (PENs) store production elements for further processing by subsequent PPNs. Two lists with production elements (`ElementLists`) are managed by a PEN; one for incoming, one for outgoing production elements. The two lists are storing elements of a certain `ProductionDataType` that is specified by the tagged value `elementType`.

8. `ElementFlow` represents a restricted association between MFERT nodes. For brevity reasons, the tagged value `source` is set to the classifier that is identified via the participant association of the first element in the ordered list of AssociationEnds. The tagged value `target` is set to the classifier that is identified via the participant association of the second element in the ordered list of AssociationEnds. The tagged value `type` identifies a ProductionDataType. It represents the type of instances that may be exchanged between the connected MFERTNodes from the source towards the target end. The main constraints of ElementFlow are: They are only allowed between two concrete MFERT nodes. Tagged values `source` and `target` refer to the two classifiers that are determined by the association ends of `ElementFlow`. ElementFlow associations are only allowed between concrete subclasses of MFERTNodes of different types, i.e., between ProductionProcess-Nodes and ProductionElementNodes. We restrict multiplicity of these association ends to 1, as an `ElementFlow` association shall indicate a relationship between two instances of MFERT nodes. Though it is allowed to navigate on `ElementFlow` associations in both ways, we graphically represent these associations as directed edges towards the target end to indicate the direction of element flow. ElementFlow associations neither specify aggregation nor composition relationships. Qualifying attributes are not considered for ElementFlows.

This is only a summary of the model-inherent restrictions defined by the profile. For a complete outline we refer to Appendix C. In [FM02a], a complete definition of the MFERT profile is given including OCL constraints, the formal MFERT model, and a mapping to I/O-Interval Structures.

The official UML 1.5 specification suggests a graphical notation for stereotypes [OMG03d, Sections 3.17, 3.18, 3.35 and 4.3]. Based on this notation, we present an overview of the MFERT profile by the virtual metamodel shown in Figure 6.5.

6.4.1 MFERT Graphical Notation in Class Diagrams

The possible alternative notation for MFERT nodes is already indicated in Figure 6.5. Though navigable in both ways, we denote `ElementFlow` associations as directed edges towards the target end to indicate the direction of production element flow. The multiplicities are omitted,

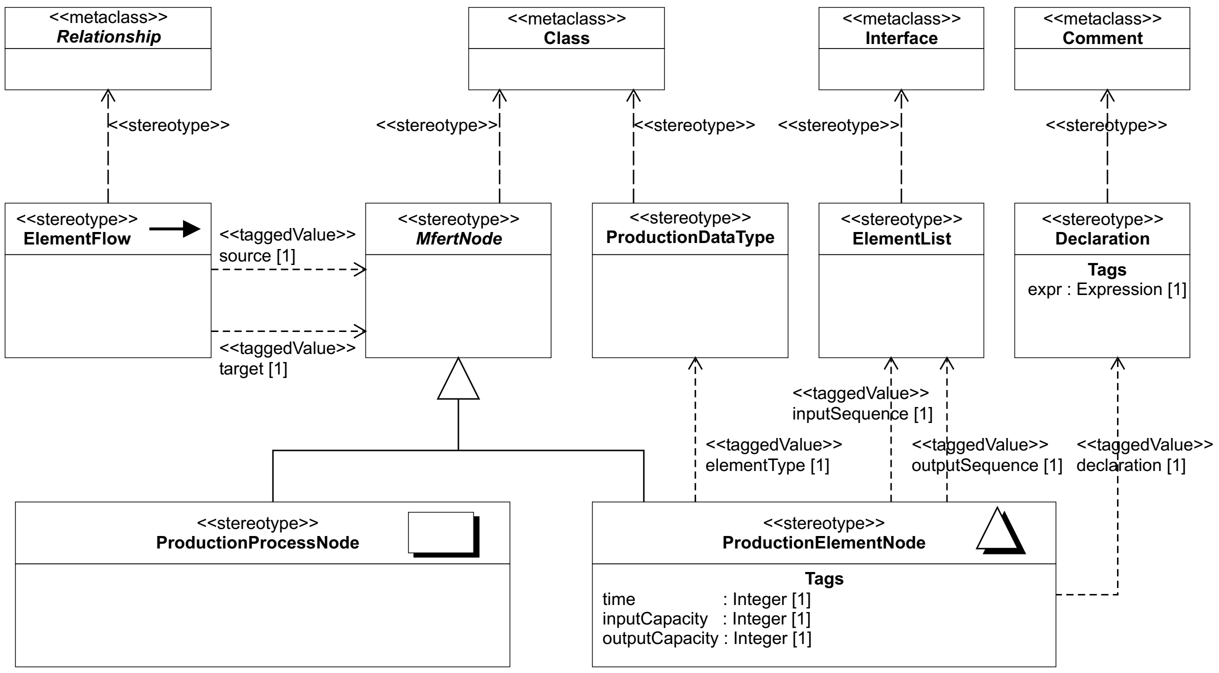


Figure 6.5: MFERT Stereotypes

as ElementFlow associations represent 1:1 relations. Annotation of the association name is optional. Some examples that illustrate the MFERT notation are shown in Figure 6.6.

6.4.2 Validation Constraints

In the previous section, we already restricted the standard UML State Diagram notation. But in order to be able to perform a mapping of MFERT designs to I/O-Interval Structures vis the UML Profile for MFERT and the timed UML State Diagram variant, we have to make the following additional restrictions.

1. We require that each concrete MFERT node is complete in the sense that its behavior description is given in form of a single timed UML State Diagram.

2. Use, composition and generalization relationships between two classes are taken into account for verification iff the two classes are (subclasses of) MFERT nodes.

3. As already mentioned, data exchanges between two MFERT nodes are necessarily performed using ElementFlow associations.

4. An MFERT node may communicate with a non MFERT node by operation call, signal or attribute modification, but these communications are not further considered.

5. Variables must have a finite value range to be applicable for the translation of timed UML State Diagrams to I/O-Interval Structures, i.e., attributes can only be enumerations or finite non-negative subsets of type Integer.

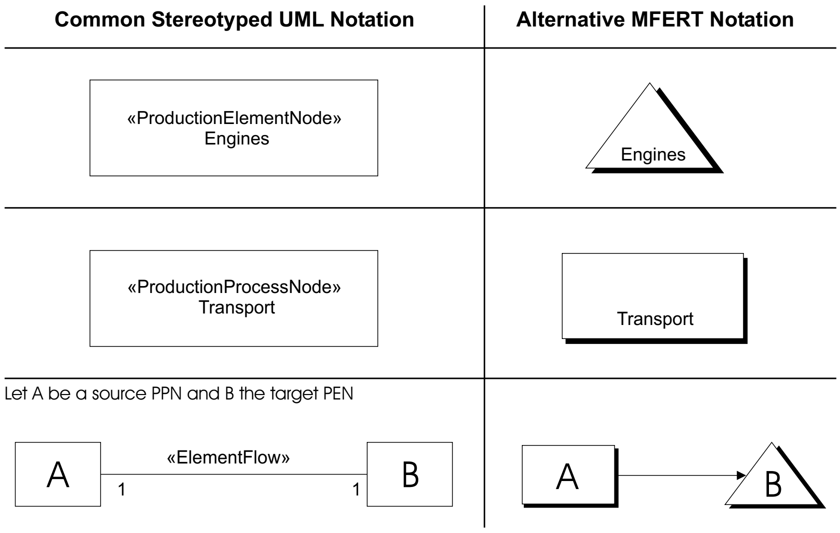

Common Stereotyped UML Notation
Alternative MFERT Notation
«ProductionElementNode» Engines
Engines
«ProductionProcessNode» Transport
Transport
Let A be a source PPN and B the target PEN
A
«ElementFlow»
1
1
B
A
B

Figure 6.6: MFERT Notation Samples

6.4.3 Mapping to the Formal MFERT Model

In this section, we map the UML Class Diagram elements for MFERT as introduced in the previous sections to the corresponding elements of the formal MFERT definition as described in Section 6.2. As UML does not provide a particular way how to define such a mapping, we take the following approach.

We extract instances of a particular UML model by means of OCL expressions and assign the results from evaluating these OCL expressions to the corresponding elements of the formal MFERT model.

As standard OCL expressions cannot result in a mathematical function definition, we here introduce a new type called `OclFunction` and an operation called `asFunction()` defined for `OclAny`. The new type `OclFunction` is introduced for technical reasons to indicate that a mathematical function is built from a given set of tuples. Basically, `OclFunction` is has attributes `definitionSet:Set(OclAny)` and `targetSet:Set(OclAny)` and an operation `f(arf:OclAny):OclAny` that constitutes the actual function. A definition of operation `asFunction()` is given by the following declaration.

```
obj->asFunction(): OclFunction
  pre: obj.oclIsKindOf(Set(Tuple(OclAny,OclAny)))
  pre: obj->collect(elem : Tuple(OclAny,OclAny) | elem.at(1))
         ->isUnique(name)
  post: result.definitionSet = obj->collect(elem | elem.at(1))->asSet()
  post: result.targetSet     = obj->collect(elem | elem.at(2))->asSet()
  post: obj->forAll(elem | result.f(elem.at(1)) = elem.at(2))
```

This operation can be applied to finite collections of tuples with 2 elements each. The first elements of the tuples form the definition set of the function, the second elements form the target set. The result is a function with a definition and a target set and a mapping that can be accessed via operation `f()`. In order to provide a total function, we implicitly set `result.f(elem) =`

`OclUndefined` for all elements of the definition set type that do not appear in the regarded set of tuples.

With this help function, the structural elements of a UML model that complies to the validation constraints for MFERT can be mapped to the components of the formal MFERT model. Table 6.1 on page 159 lists the corresponding definitions.

Concerning the set FSM of finite state machines, we here simply substitute the general notion of $FSMs$ by the timed UML State Diagram variant. Note that this notation allows more general behavior of MFERT nodes than the abstract interpreter functionality defined in Sections 6.3.1 and 6.3.2. We therefore restrict the actions and activities of MFERT nodes according to the definition of timed FSMs in Appendix A.

6.5 Contributions of the Chapter

This chapter addressed the following issues:

- In this chapter, a set-theoretic formal model of MFERT is defined. Basically, it is a simplified version of the general functional MFERT description scheme by Uta Schneider [Sch96]. In particular, certain kinds of actions and activities are identified for MFERT nodes, i.e, requests for putting and getting production elements, actual transfers of production items between MFERT nodes, and transformation activities with a notable duration.

- The dynamic semantics of MFERT is defined by means of abstract interpreters that are local to each node.

- For a concrete notation of the behavior of MFERT nodes, FSMs in form of the timed UML State Diagram variant of Chapter 5 are employed. Note that further restrictions on the set of actions and activities are employed.

- A UML Profile for the structural elements of MFERT is defined. This allows to associate OCL constraints to MFERT nodes, as MFERT nodes are interpreted as stereotypes of classes. For UML classes, in turn, OCL constraints in form of invariants can be applied (cf. Section 2.3.3).

Table 6.1: Mapping to Formal MFERT

PE	:= `ProductionElementNode->allInstances()`
PP	:= `ProductionProcessNode->allInstances()`

E	:= `ElementFlow->allInstances()`
	    `->collect(e:ElementFlow | Sequence{e.source, e.target})`
	    `->asSet()`

DT	:= `ProductionDataType->allInstances()`
	    `->collect(pdt:ProductionDataType |`
	            `pdt.allAttributes()->collect(type)->sortedBy(name))`
	    `->asSet()`

C	:= `ProductionElementNode->allInstances()` `->collect(p | Sequence{p, p.elementType})->asFunction()`
In	:= `ProductionElementNode->allInstances()` `->collect(p | Sequence{p, p.inputSequence} )->asFunction()`
Out	:= `ProductionElementNode->allInstances()` `->collect(p | Sequence{p, p.outputSequence})->asFunction()`
Cap_{In}	:= `ProductionElementNode->allInstances()` `->collect(p | Sequence{p, p.inputCapacity} )->asFunction()`
Cap_{Out}	:= `ProductionElementNode->allInstances()` `->collect(p | Sequence{p, p.outputCapacity})->asFunction()`
$Time$	:= `ProductionElementNode->allInstances()` `->collect(p | Sequence{p, p.time})->asFunction()`

FSM	:= `ProductionProcessNode->allInstances()`
	    `->collect(p | p.behavior)    -- 'behavior' is a link to`
	    `->asSet()                     -- a timed UML State Diagram`

M_{FSM}	:= `ProductionProcessNode->allInstances()` `->collect(p | Sequence{p, p.behavior})->asFunction()`

$Init_{In}$	:= `ProductionElementNode->allInstances()`
	    `->collect(p |`
	        `Sequence{p, p.inputDeclaration.expr.oclAsType(String)})`
	    `->asFunction()`

$Init_{Out}$	:= `ProductionElementNode->allInstances()`
	    `->collect(p |`
	        `Sequence{p, p.outputDeclaration.expr.oclAsType(String)})`
	    `->asFunction()`

Chapter 7

Real-Time Properties with OCL

Professional Engineers are expected to use discipline, science, and mathematics to assure that their products are reliable and robust. We should expect no less of anyone who produces programs professionally.
– David Lorge Parnas [Par95]

In the domain of database systems, different types of *semantic integrity constraints* are distinguished [EN00]. *Static constraints* define required properties on nontransient system states, i.e., static properties within one system state. *Transition constraints* deal with system changes between two subsequent states. In real-time systems design, additionally *temporal constraints* are identified that consider sequences of state transitions in combination with timing bounds. While static and transition constraints can already be expressed with OCL, it currently lacks means to express temporal constraints.

To overcome this, we introduce temporal OCL operations that enable modelers to specify state-oriented behavior. The proposed OCL extension reasons about future object states, since we define the semantics based on a future oriented tree temporal logic without loss of generality. Accordingly, OCL can also be easily extended for specification of past-oriented constraints. This work has evolved through the recent years. Due to a missing OCL metamodel in the current UML 1.5 specification, we first took the OCL type metamodel presented by Baar and Hähnle [BH00] and performed a rather heavyweight extension by directly extending that metamodel [FM02c]. In March 2003, the OCL 2.0 proposal by Ivner et al. [IHJ$^+$03] has been recommended for adoption by the Analysis and Design Platform Task Force of the OMG [OMG03a]. With the official adoption of OCL 2.0 in October 2003 [OMG03b], we can now develop a 'lightweight' extension by means of a UML Profile for our temporal OCL extension.

There are other works concerning temporal extensions of OCL. We will discuss them at the end of this chapter. As a guideline for developing a consistent and successful OCL extension, we here formulate some requirements to fulfill:

Requirement 7.1 *An OCL extension that enables to specify temporal requirements should reuse existing OCL concepts and keep the common syntax of OCL to keep the learning curve low for OCL users.*

Requirement 7.2 *With the resulting OCL extension, it must be possible to express all patterns identified in the specification pattern system by Dwyer et al. in [DAC98a] (cf. Section 3.2.2).*

As timing issues are not covered at all in the pattern system, it is necessary to additionally provide corresponding time-related specification means for the domain of real-time systems. For the context of this thesis, we therefore additionally consider the following additional, domain-specific requirement.

Requirement 7.3 *In order to be able to express real-time constraints, explicit timing annotations and timing intervals must be supported in the resulting OCL extension. The corresponding temporal OCL expressions must have a formal semantics.*

7.1 UML Profile for Real-Time Constraints with OCL

The integration of State Diagram states into the formal model for OCL expressions allows to extend OCL towards specification of constraints that regard the *state-related behavior* of a model.

For example, consider again the manufacturing scenario with classes `Machine` and `InputBuffer`. Assume that class `InputBuffer` has an associated State Diagram in which state configuration `Set{Loading}` represents that an item is currently being loaded into the buffer. In such simple cases, we allow to omit the set-notation and may simply specify `Loading` to denote a configuration.

To ensure production progress, we require that items have to periodically arrive at the input buffer within 400 time units. With other words, state `Loading` is always reached again within 400 time units. In our temporal OCL extension, a corresponding OCL constraint is

```
context InputBuffer inv:
  self@post(1,400)->forAll( trace | trace->includes(Loading))
```

Operation `post(a,b)` basically returns the set of all possible traces of state configurations starting in the current system state. Parameters `a` and `b` are timing delimiters that specify the timing interval to consider. In the example, this is the next 400 time units, i.e., `post(1,400)` returns a set of traces, where each trace is a sequence of 400 elements. The elements of a trace in turn are state configurations (formally, we restrict on the component Σ_{CONF} of trace $\sigma(M)$ as defined in Definition 4.17). In the example, state `Loading` already specifies a state configuration, such that we can apply `p->includes(Loading)` to require that trace `p` must include state configuration `Loading`.

Further examples can be found in Chapter 8. In the remainder of this section, we define a corresponding language extension based on the adopted OCL 2.0 specification by the following approach.

Syntactically, we first extend the abstract OCL syntax by stereotypes for temporal expressions in Subsection 7.1.1. But to support modeling at the user level, a concrete syntax and operations have additionally to be defined for this extension on layer M1 of the UML 4-layer architecture. Therefore, we add some new production rules to the concrete syntax grammar of the OCL 2.0 specification in Subsection 7.1.2. Note that we cannot avoid the overlap with the M1 layer in an OCL Profile, since OCL predefines types and operations on that level. As the concrete OCL syntax only partly provides the operations that are defined in OCL expressions, a standard library of predefined OCL operations is specified in [OMG03b, Chapter 11]. Correspondingly, we define operations in the context of temporal expressions in Subsection 7.1.3. Semantically, our proposed state-based temporal OCL extension makes use of the notion of *time-based traces* that are also defined in that subsection.

7.1.1 OCL Metamodel Extensions

The OCL 2.0 specification distinguishes two subpackages for its metamodel package `Ocl-AbstractSyntax` (see Figure 2.9); the *OCL type metamodel* describes the predefined OCL types and affiliated UML types, while the *OCL expression metamodel* describes the structure of OCL expressions.

States in OCL. In the OCL type metamodel, the metaclass for State Diagram states is `OclModelElementType`. Generally, the metaclass `OclModelElementType` represents the types of elements that are `ModelElements` in the UML metamodel. In that particular case, the model elements are states (or more precisely, instances of a concrete subclass of the abstract metaclass `State`), and the corresponding instance of `OclModelElementType` on layer M1 is the predefined OCL type `OclState`.

For each state, there implicitly exists a corresponding enumeration literal in `OclState`, i.e., `OclState` is seen as an enumeration type on the M1 layer, accumulating the state names of all State Diagrams. As there is no particular information provided how these enumeration literals are syntactically defined, we require here that the complete path – excluding the top state – is used (cf. Definition 4.5, item 7(c)). The state names along the path are syntactically separated by double colons, e.g., state `N` in Figure 7.1 becomes the enumeration literal `X::B::N`. In anticipation of the concrete syntax changes to be introduced, we identify final State Diagram states by the new OCL keyword `FinalState`.

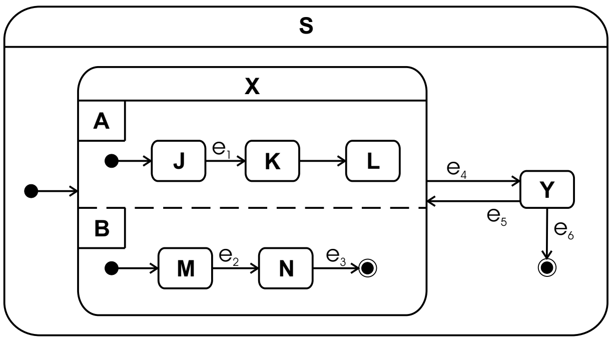


Figure 7.1: Concurrent State Diagram

Configurations. The building blocks of State Diagrams are hierarchically ordered states. Note that we do not regard pseudo states (like synch, stub, or history states) in this context and recall that a *composite state* is known as a state that has a set of substates and can be *concurrent*, i.e., consisting of orthogonal regions which in turn are (composite) states. *Simple* *states* are non-pseudo, non-composite states. To uniquely identify an active state configuration, it is sufficient to list the comprising simple states, which we denote as a *basic configuration* in accordance with Definition 4.14.

However, several other notions are imaginable in this context and can be easily adapted, e.g., the approach of UML 1.5 that takes the whole state tree as a configuration. For explicit specification purposes, we might also allow for *underspecified configurations* to represent sets of valid configurations. For instance, in Figure 7.1, `Set{X::A::J,X::B}` could be a valid configuration specification in the sense that it denotes the set of configurations

```
{ Set{X::A::J,X::B::M}, Set{X::A::J,X::B::N}, Set{X::A::J,X::B::FinalState} } .
```

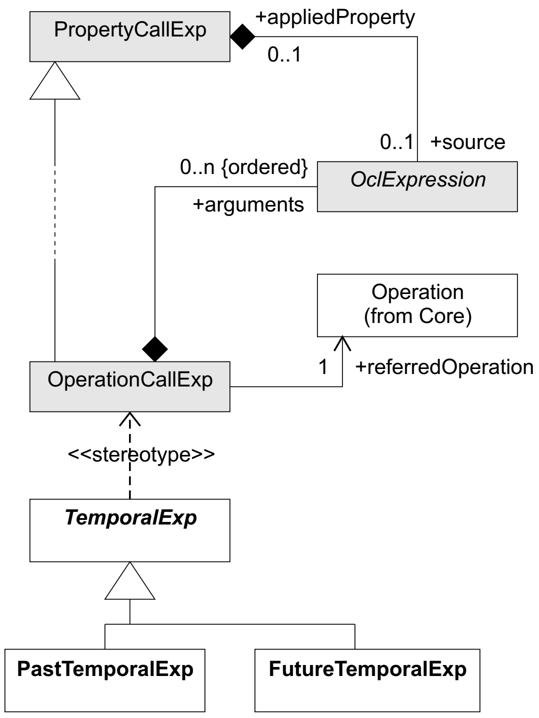


Figure 7.2: Stereotypes for Temporal Expressions

Temporal Expressions. In the OCL expression metamodel, we introduce a new kind of operation call, i.e., stereotype `TemporalExp` represents a temporal expression that refers to traces of state configurations (cf. Figure 7.2)[1]. It is the abstract superclass of stereotypes `PastTemporalExp` for past-oriented and `FutureTemporalExp` for future-oriented temporal expressions, respectively. We need these two stereotypes in order to define a semantics for corresponding temporal operations (see Section 7.1.4).

[1]For our stereotype definitions, we make use of the graphical notation suggested in the official UML 1.5 specification [OMG03d, Sects. 3.17, 3.18, 3.35, and 4.3]. In Figures 7.2 and 7.3, metaclasses taken from the OCL 2.0 metamodel are marked by gray boxes.

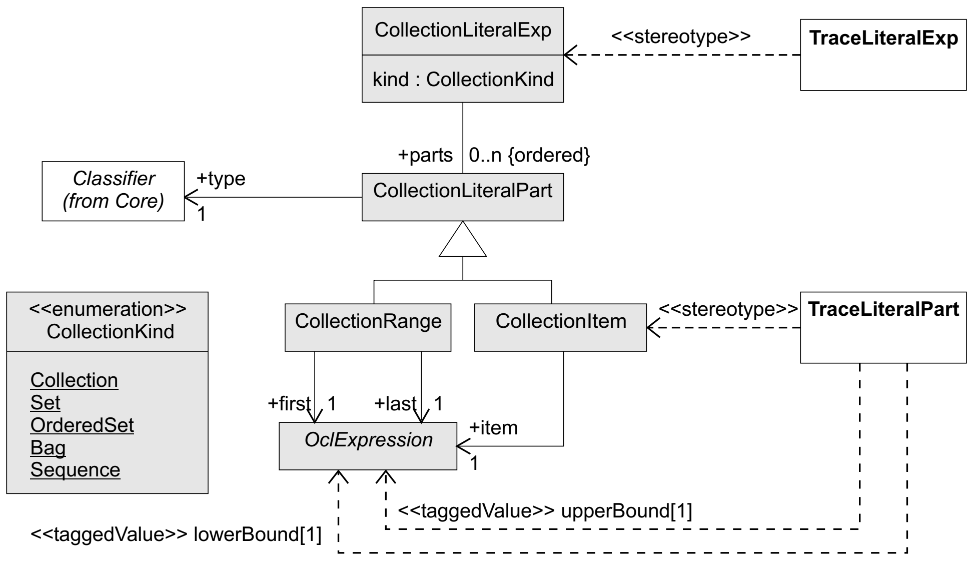


Figure 7.3: Parts of the OCL Expression Metamodel with Stereotypes for Traces

Trace Literals. As we want to reason about traces by means of states and configurations, we also need a mechanism to explicitly specify traces with annotated timing intervals by literals. We therefore define stereotypes `TraceLiteralExp` and `TraceLiteralPart` as illustrated in Figure 7.3. The following restrictions apply here, leaving out the corresponding well-formedness rules by means of OCL for reasons of brevity.

1. The collection kind of stereotype `TraceLiteralExp` is `CollectionKind::Sequence`.

2. The type associated with a `TraceLiteralPart` must be `Set(OclState)`. Note that we do not require explicit specification of a set when a state configuration can already be specified by one state only. In this case, type `OclState` is implicitly casted to `Set(OclState)`.

3. Each `TraceLiteralPart` has a lower bound and an upper bound.

4. Lower bounds must evaluate to non-negative Integer values.

5. Upper bounds must evaluate to non-negative Integer values or to the String ‘`inf`’ (for *infinity*). In the first case, the upper bound value must be greater or equal to the corresponding lower bound value.

7.1.2 Concrete Syntax Changes

Having defined new classes for temporal expressions on the abstract syntax level, modelers are not yet able to use these extensions, as they specify OCL expressions by means of a concrete syntax. In Chapter 4 of the OCL 2.0 specification, a concrete syntax is given that is compliant with the current OCL standard. The new concrete syntax is defined by an attributed grammar with production rules in EBNF that are annotated with synthesized and inherited attributes as well as disambiguating rules. *Inherited attributes* are defined for elements on the right hand side of production rules. Their values are derived from attributes defined for the left hand side of the corresponding production rule. For instance, each production rule has an inherited attribute `env` (environment) that represents the rule's namespace. *Synthesized attributes* are used to keep results from evaluating the right hand sides of production rules. For instance, each production rule has a synthesized attribute `ast` (abstract syntax tree) that constitutes the formal mapping from concrete to abstract syntax. *Disambiguating rules* allow to uniquely determine a production rule if there are syntactically ambiguous production rules to choose from.

In the following, we present some additional production rules for the concrete syntax of the OCL 2.0 specification. A mapping to the extended abstract OCL syntax is provided for each new production rule.

OperationCallExpCS[2]

Eight different forms of operation calls are already defined in the OCL 2.0 concrete syntax. In particular, it is distinguished between infix and unary operations, operation calls on collections, and operation calls on objects (with or without '@pre' annotation) or whole classes. We additionally introduce rule [J] for temporal operation calls and list the synthesized and inherited attributes for syntax [J] below. Disambiguating rules for syntax [J] are defined in the specific rules for temporal expressions.

```
[A] OperationCallExpCS ::= OclExpressionCS[1]   simpleNameCS OclExpressionCS[2]
[B] OperationCallExpCS ::= OclExpressionCS '->' simpleNameCS '(' argumentsCS? ')'
[C] OperationCallExpCS ::= OclExpressionCS '.'  simpleNameCS '(' argumentsCS? ')'
...
[J] OperationCallExpCS ::= TemporalExpCS

  Abstract Syntax Mapping:
    -- (Re)type the abstract syntax tree variable 'ast'
    OperationCallExpCS.ast : OperationCallExp
  Synthesized Attributes:
    -- Build the abstract syntax tree
    [J] OperationCallExpCS.ast = TemporalExpCS.ast
  Inherited Attributes:
    -- Derive the namespace stored in variable 'env'
    [J] TemporalExpCS.env = OperationCallExpCS.env
```

TemporalExpCS

A temporal expression is either a past- or future-oriented temporal expression.

```
[A] TemporalExpCS ::= PastTemporalExpCS
[B] TemporalExpCS ::= FutureTemporalExpCS
```

We leave out the rather simple attribute definitions here. Basically, the abstract syntax mapping defines `TemporalExpCS.ast` to be of type `TemporalExp`, the synthesized attribute `ast` is built from the right hand sides, and the inherited attribute `env` is derived from `TemporalExpCS`.

[2]All non-terminals are postfixed by 'CS' (short for *Concrete Syntax*) to better distinguish between concrete syntax elements and their abstract syntax counterparts.

FutureTemporalExpCS

A future-oriented temporal expression is a kind of operation call. We additionally have to introduce the operator '@' to indicate a subsequent temporal operation. Note that an operation call in the abstract syntax has a source, a referred operation, and operation arguments, so the abstract syntax tree `ast` must be built with corresponding synthesized attributes.

```
FutureTemporalExpCS ::= OclExpressionCS '@' simpleNameCS '(' argumentsCS? ')'

  Abstract Syntax Mapping:
     FutureTemporalExpCS.ast : FutureTemporalExp
  Synthesized Attributes:
     FutureTemporalExpCS.ast.source    = OclExpressionCS.ast
     FutureTemporalExpCS.ast.arguments = argumentsCS.ast
     FutureTemporalExpCS.ast.referredOperation =
         OclExpressionCS.ast.type.lookupOperation(simpleNameCS.ast,
                                                  if argumentsCS->notEmpty() then
                                                    argumentsCS.ast->collect(type)
                                                  else
                                                    Sequence{}
                                                  endif )
  Inherited Attributes:
     OclExpressionCS.env = FutureTemporalExpCS.env
     argumentsCS.env     = FutureTemporalExpCS.env
  Disambiguating Rules:
     -- Operation name must be a (future-oriented) temporal operator.
    [1] Set{'post'}->includes(simpleNameCS.ast)
     -- The operation signature must be valid.
    [2] not FutureTemporalExpCS.ast.referredOperation.oclIsUndefined()
```

If other temporal operations than @ `post(a,b)` need to be introduced at a later point of time, only disambiguating rule [1] has to be modified correspondingly. For instance, `next()` might be introduced as a shortcut for `post(1,1)`, or `post()` without any parameters could be the shortcut for `post(1,'inf')`.

A corresponding extension to past temporal operations can be easily introduced, e.g., by means of the operation name `pre()`. In the remainder, we only focus on `FutureTemporal-ExpCS`. Note that `pre` and `post` as operation names cannot be mixed up with pre- and postcondition labels or the @ `pre` time marker, because operations require subsequent brackets.

TraceLiteralExpCS

Trace literal expressions are a special form of collection literal expressions, as they represent sequences of explicitly specified configurations. In order to allow interval definitions for trace specifications, we have to specify some new production rules. We first introduce a new chain production rule to provide an alternative to common collection literal expressions.

```
[A] CollectionLiteralExpCS ::= CollectionTypeIdentifierCS
                                                '{' CollectionLiteralPartsCS? '}'
[B] CollectionLiteralExpCS ::= TraceLiteralExpCS

  Abstract Syntax Mapping:
     CollectionLiteralExpCS : CollectionLiteralExp
  Synthesized Attributes:
     ...
     [B] CollectionLiteralExpCS.ast.parts = TraceLiteralExpCS.ast.parts
     [B] CollectionLiteralExpCS.ast.kind  = TraceLiteralExpCS.ast.kind
  Inherited Attributes:
     ...
     [B] TraceLiteralExpCS.env = CollectionLiteralExpCS.env
```

In syntax [A], `CollectionTypeIdentifierCS` distinguishes between literals for collections (`Set`, `OrderedSet`, `Sequence`, and `Bag`), and production rule `CollectionLiteralPartsCS` collects a number of expressions. Option [B] is added to provide a notation for traces. The collection kind of traces is `CollectionKind::Sequence` by default, as specified below.

```
TraceLiteralExpCS ::= 'Trace' '{' TraceLiteralPartsCS '}'

  Abstract Syntax Mapping:
     TraceLiteralExpCS.ast : TraceLiteralExp
  Synthesized Attributes:
     TraceLiteralExpCS.ast.parts = TraceLiteralPartsCS.ast
     TraceLiteralExpCS.ast.kind  = CollectionKind::Sequence
  Inherited Attributes:
     TraceLiteralPartsCS.env = TraceLiteralExpCS.env
```

We here introduce the new keyword `Trace` to denote trace specifications, but note that no new kind of collection type is necessary on the metalevel, as we treat traces simply as sequences.

TraceLiteralPartCS

The production rule `TraceLiteralPartsCS` assembles the individual elements of a trace specification. It is defined correspondingly to the already existing production rule for collection literal parts, such that definitions of `ast` and `env` are left out for reasons of brevity.

```
TraceLiteralPartsCS[1] ::= TraceLiteralPartCS (',' TraceLiteralPartsCS[2] )?
```

For each trace literal part, a timing interval may be associated, which specifies how long a configuration is active. Intervals are of the syntactical form `[a,b]`, with `a` evaluating to a non-negative Integer, and `b` either a non-negative Integer with `b` $\geq$ `a` or the String `'inf'` (cf. well-formedness rules of `TraceLiteralExp` in Section 7.1.1). If only one delimiter is specified, this is taken as the upper bound, and the lower time bound is implicitly set to zero. If no interval is specified at all, the bounds are implicitly set to `[0,'inf']`. The corresponding grammar rule is as follows.

```
TraceLiteralPartCS ::= OclExpressionCS[1]
                            ( '[' (OclExpressionCS[2] ',')?
                                  (OclExpressionCS[3] | 'inf') ']'
                            )?

  Abstract Syntax Mapping:
     TraceLiteralPartCS.ast : TraceLiteralPart
  Synthesized Attributes:
     TraceLiteralPartCS.ast.item = OclExpressionCS[1].ast
```

```
    TraceLiteralPartCS.ast.lowerBound = if OclExpressionCS[2]->notEmpty() then
                                          OclExpressionCS[2].ast
                                        else
                                          '0'
                                        endif
    TraceLiteralPartCS.ast.upperBound = if OclExpressionCS[3]->notEmpty() then
                                          OclExpressionCS[3].ast
                                        else
                                          'inf'
                                        endif
 Inherited Attributes:
    OclExpressionCS[1].env = TraceLiteralPartCS.env
    OclExpressionCS[2].env = TraceLiteralPartCS.env
    OclExpressionCS[3].env = TraceLiteralPartCS.env
```

CollectionTypeCS
To allow trace specifications as part of variable definitions and provide a means for explicit typing on the concrete syntax level, we need to add a rule for *explicit* referencing to a type called `Trace`. We therefore add an alternative production rule in the context of `collectionTypeCS`.

```
[A] collectionTypeCS ::= collectionTypeIdentifierCS '(' typeCS ')'
[B] collectionTypeCS ::= 'Trace'

 Abstract Syntax Mapping:
   typeCS.ast : CollectionType
 Synthesized Attributes:
   ...
   [B] collectionTypeCS.ast.oclIsKindOf(SequenceType)
   [B] collectionTypeCS.ast.elementType.oclIsKindOf(SetType)
   [B] collectionTypeCS.ast.elementType.elementType.oclIsKindOf{OclState)
 Inherited Attributes:
   -- none for [B]
```

7.1.3 Standard Library Operations

In our previous work [FM02c], we introduced two new built-in types called `OclConfiguration` and `OclPath` on the M1 layer to handle temporal expressions. We present an alternative approach that avoids to introduce new types and instead operates on the already existing OCL collection types.

Configuration Operations. For configurations as a special form of sets of states, we have to elaborate on operations applicable to sets that return collections since the resulting collection can be an invalid configuration with an arbitrary set of states. Nevertheless, most of the existing general collection operations [OMG03b, Section 11.7] can be directly applied to configurations. These are: $=$, $<>$, size(), count(), isEmpty(), notEmpty(), includes(), includesAll(), excludes(), and excludesAll(). In addition, iterator operations exists(), forAll(), any(), one() are applicable as well [OMG03b, Section 11.9.1]. Other OCL set operations applied to configurations, e.g., union() and intersection(), might result in arbitrary sets of states rather than in valid configurations. We allow such operations, but explicitly mention that they have to be used with care.

Trace Operations. Similar to configurations, many of the existing OCL sequence operations can immediately be applied to traces of configurations. These operations are: $=$, $<>$, size(), isEmpty(), notEmpty(), includes(), includesAll(), excludes(), excludesAll(), subSequence(), prepend(), first(), at(), exists(), forAll(), any(), one(). Operations last() and append() can be applied to traces of finite length only. Note that some sequence operations may result in invalid traces, e.g., select() and collect().

Additional Operations for OclAny. We introduce an operation `oclInConf()` that checks for an active configuration. Given a system state $\sigma(\mathcal{M})$, an object $\underline{oid} \in \Sigma_{CLASS,c}$, and a set of states $cfg \in I_{type}(Set(OclState))$, the semantics of operation `oclInConf()` is then defined by function

$$I[[oclInConf : OclAny \times Set(OclState) \rightarrow Boolean]](\underline{oid}, cfg) \overset{def}{=} \begin{cases} true, & \text{if } \underline{oid} \in \Sigma_{ACTIVE,c} \wedge cfg \in B_c \\ & \quad \wedge\, cfg = \sigma_{CONF,c}(\underline{oid}), \\ false, & \text{if } \underline{oid} \in \Sigma_{ACTIVE,c} \wedge cfg \in B_c \\ & \quad \wedge\, cfg \neq \sigma_{CONF,c}(\underline{oid}), \\ \perp, & \text{if } \underline{oid} \notin \Sigma_{ACTIVE,c} \vee cfg = \perp \\ & \quad \vee\, (\underline{oid} \in \Sigma_{ACTIVE,c} \wedge cfg \notin B_c \cup \{\perp\}). \end{cases}$$

In the definition above, B_c denotes the set of basic configurations of State Diagram SC_c (based on Definition 4.14). The definition does not consider underspecified configurations as discussed in Subsection 7.1.1. We here only describe the idea how to achieve the complete formal semantics. First, we additionally define the set $UnderSpecified_c$ of valid underspecified configurations for a given State Diagram SC_c. Then, we provide a mapping $basicConfs_c : UnderSpecified_c \rightarrow B_c$ that gives for each underspecified configuration the corresponding set of basic configurations. Finally, the conditions of the formal semantics are adjusted, e.g., $UnderSpecified_c$ replaces B_c and condition $cfg = \sigma_{CONF,c}(\underline{oid})$ is replaced by the condition $\forall b \in basicConfs_c(cfg) : b \in \sigma_{CONF,c}(\underline{oid})$.

We also introduce operation `post(a,b)` as a new temporal operation of `OclAny` and allow the @-operator to be used only for such temporal operations. @ `post(a,b)` returns *a set of possible future traces* in the interval [a,b]. First, all possible traces that start with the current configuration are regarded, and then the timing interval [a,b] determines the subtraces that have to be returned by the operation. The result has to be a *set* of traces, as there are typically different orders of executions possible in the future steps of a State Diagram. Note that in an actual execution of a State Diagram there is of course only exactly one of the possible traces selected. An informal semantics of `post(a,b)` is given as follows.

```
OclAny.post(a:Integer,b:OclAny) : Set(Sequence(Set(OclState)))
  pre: a >= 0 and ( (b.oclIsTypeOf(Integer) and b >= a) or
                    (b.oclIsTypeOf(String) and b = 'inf'))

  -- The operation returns a set of possible future state configuration traces
  -- in the interval [a,b] including the configurations of time points a and b.
```

Additional operations, such as @post(a:Integer) or @next(), can be easily added [FM02c]. These are operations basically derived from @post(a,b).

7.1.4 Semantics of Temporal Expressions

In this subsection, we define a formal semantics of operation post(a,b). We make use of the nested collection type $TRACE \overset{def}{=} Sequence(Set(OclState))$.

When UML State Diagrams are equipped with time, system state traces as given by Definition 4.17 must be extended to capture timing information as well. In this context, the *UML Profile for Scheduling, Performance and Time* provides a variety of timing concepts [OMG03c, Chapter 5]. In particular, timing mechanisms by means of a stereotype ≪ RTclock≫ can be introduced together with appropriate tagged values, e.g., RTresolution. Progress of time is usually measured by counting the number of expired cycles of a strictly periodic *reference clock*. This results in a discretization of time, i.e., distinct physical instants might be associated with the same clock instant when they are temporally 'too close' to each other. Therefore, a sufficient resolution of the reference clock must be chosen for the particular model under investigation.

We assume in the following that a system-wide reference clock is defined together with a known resolution. The duration between two time instants is referred to as one *time unit*. This leads to an Integer-based notion of unit time delay, i.e., each time instant can be represented by an Integer value (in contrast to dense time, where time instants are represented by Real values). A trace in such a timed model is then defined as follows.

Definition 7.1 *(Time-based Trace)*
A time-based trace *for an instantiation of an extended object model $\mathcal{M}$ is an (infinite) sequence of system states,*

$$trace(\mathcal{M}) \overset{def}{=} \langle\langle\ \sigma(\mathcal{M})_{[0]}, \sigma(\mathcal{M})_{[1]}, \ldots, \sigma(\mathcal{M})_{[i]}, \ldots \rangle\rangle,$$

where each $\sigma(\mathcal{M})_{[i]}$, $i \in \mathbb{N}_0$, represents the system state i time units after start of execution. In particular, $\sigma(\mathcal{M})_{[0]}$ denotes the initial system state.

Note that we still require the same properties as in common traces, in particular, only one operation call per object is permitted in consecutive elements of the trace. This can be guaranteed by assuming that execution of an operation takes at least one time unit. System states of time-based traces can be compared to *clocked states* of *runs* for Interval Structures as described in Section 3.3 and Table 3.5. We describe the corresponding semantic mapping in Subsection 7.1.4.

Given a system state $\sigma(\mathcal{M})_{[i]}$ at time instant i, an object $\underline{oid} \in \Sigma_{CLASS,c}$, an integer value $a \in I_{type}(Integer)$, and a value $b \in I_{type}(Integer) \cup \{\infty\}$. For parameter b, we assume here that the string '`inf`' defined in the concrete syntax is directly mapped to ∞. For the symbol ∞, it holds that

$$\forall i \in \mathbb{N}_0 : i < \infty \ \wedge \ i + \infty = \infty \ \wedge \ i - \infty = \infty .$$

A trace $trace_{\underline{oid},a,b\ [i]} \in I_{type}(TRACE))$ for object $\underline{oid}$ that starts at time $i + a$ and ends at time $i + b - a$ is then defined by

$$trace_{\underline{oid},a,b\ [i]} \overset{def}{=} \langle cfg_0, \ldots, cfg_{b-a} \rangle, \text{ where } \forall j \in \{0, \ldots, b-a\} : cfg_j \in \sigma_{CONF}(\underline{oid})_{[i+j]}.$$

Each $trace_{\underline{oid},a,b\ [i]}$ is interpreted as a *possible* future execution path. It is just a possible trace, as is is not determined at time i whether $cfg_j, j \in \{1, \ldots, b-a\}$, will be reached at the later point of time $i + j$.

In the case that $b = \infty$, a trace $trace_{\underline{oid},a,b\ [i]}$ is of infinite length. An explicit instantiation of such traces as part of the model is therefore not intended. However, it is possible to give corresponding formal specifications by means of temporal logics, as illustrated below. Temporal logics specifications can then be directly used by model checkers.

Denoting the set of all possible future execution paths by $\{trace_{\underline{oid},a,b\ [i]}\}$, the semantics of operation `post(a,b)` is then defined by

$$I[[post : OclAny \times Integer \times OclAny \rightarrow Set(TRACE)]](\underline{oid}, a, b) \overset{def}{=}$$
$$\begin{cases} \{\ trace_{\underline{oid},a,b\ [i]}\ \}, & \text{if } \underline{oid} \in \Sigma_{ACTIVE,c} \\ & \quad \wedge\, a \geq 0 \ \wedge \ b \geq a, \\ \perp, & \text{if } \underline{oid} \notin \Sigma_{ACTIVE,c}, \\ \perp, & \text{if } a < 0 \ \vee \ a = \perp \\ & \quad \vee\, b < a \ \vee \ b = \perp . \end{cases}$$

7.2 Expressing Specification Patterns

In this section, we demonstrate how to express patterns of the pattern system presented in Section 3.2.2 by means of our temporal state-oriented OCL extension.

It turns out that only some minor extensions are necessary to cover all property patterns. Firstly, a new operation needs to be introduced that is particularly applicable to traces, i.e., operation `startsWith(Sequence(Set(OclState)):Boolean` that checks for a matching subsequence of configurations. And secondly, specification means for trace literal parts have to be extended. A trace literal part becomes a logical expression with configurations as operands and unary and binary operators (such as `not`, `and`, `or`) as connectives.

We here take the absence pattern as an example and provide corresponding temporal OCL expressions in Table 7.1. To understand the OCL expressions in that table, we informally explain their semantics in the remainder.

We apply the patterns in terms of *state configurations*, i.e., the set of states that uniquely determines the currently active states in a UML State Diagram. Consequently, in contrast to the original patterns, `P`, `Q`, and `R` denote configurations in the remainder. Nevertheless, note that in the simplest case a configuration consists of one state. As configurations uniquely determine

Table 7.1: OCL Expressions for Absence Pattern (Assumptions implicitly as in Table 3.2)

P is false . . .	
. . . globally	`inv: not self.oclInConf(P)`
. . . before R	`init: self@post()->forAll(g | g->startsWith( Sequence{not P, R} ))`
. . . after Q	`inv: self.oclInConf(Q) implies self@post()->forAll(g | g->excludes(P))`
. . . between Q and R	`inv: self.oclInConf(Q) implies self@post()->forAll(g | g->startsWith( Sequence{not P, R} ))`
. . . after Q until R	`inv: self.oclInConf(Q) implies not self@post()->exists(g | g->startsWith( Sequence{not R, P} ))`

the current state-related status of an object, conditions of form ‘`P and not Q`’ are equal to the simple formula ’`P`’, as two distinct configurations of a State Diagram can by definition never occur at the same time.

The following concepts and operations have been newly introduced to OCL to be able to express the specification patterns. Note that we keep compliant with the existing standard OCL syntax and reuse as often as possible existing collection operations like `forAll()`, `exists()`, `includes()`, and `excludes()`.

1. The only state-related operation of the current OCL standard as well as the new OCL 2.0 specification is called `oclInState(s:OclState)`. It is defined over objects of user-defined classes that have an associated State Diagram. Operation `oclInState(s:OclState)` returns true if state `s` is currently active.

   Additionally, we define and make use of operation `oclInConf(c:Set(OclState))` for State Diagram configurations. This operation returns true if the object is in configuration `c` at time of evaluation.

2. In addition to OCL invariants declared by the keyword `inv`, we introduce a new clause called `init`. In contrast to an invariant over an object `obj` that has to hold each time after `obj`’s status has changed, the expression of an `init`-clause has to hold only at the starting point of execution. Nevertheless, note that the expression of the `init`-clause may be a temporal OCL expression.

3. Temporal OCL expressions are a new concept introduced to enable specification of dynamic, behavioral constraints. In our approach, temporal OCL expressions make use of a special *temporal operation* with signature `post(a:Integer,b:OclAny)`. To further emphasize that this is a temporal operation, we make use of a leading separator `@` instead of the common dot-notation. The operation can be applied to objects of user-defined classes that have an associated State Diagram.

   When `@post(a,b)` is evaluated at a certain point of time `t`, we obtain the set of possible configuration sequences in the timing interval `[t+a,t+b]`. If parameters `a` and `b` are omitted, we set `a = 1` and `b = 'inf'` (short for infinity to cover infinite future executions).

4. We have defined an extended syntax for explicit specification of configuration sequences. This syntax is particularly tailored to the needs for formulating general execution paths that may be subject to some additional conditions. Basically, we allow that logical unary and binary operators such as '`not`', '`and`', '`or`' are applied to sequence elements [FM02c]. For the real-time domain, we also allow explicit timing intervals in this context, but note that these are not required for the general patterns we investigate in this thesis. E.g., the sequence specification

   ```
   Sequence{ not P [1,100], P [1,'inf'], Q }
   ```

   means that configuration `P` must not be true until within 100 time units configuration `P` is reached, and afterwards configuration `Q` must eventually become true. When the timing interval for one of the first $n-1$ sequence elements is left out, it is implicitly set to `[1,'inf']`, but note that consecutive configuration specifications must still eventually become true (so-called *strong until* semantics).

5. We newly introduce the boolean operation `startsWith(g:Sequence(T))`, which can be applied to sequences of objects of some type `T`. That operation checks whether a given sequence starts with a sequence specified by parameter `g`.

   In particular, when `T` is equal to type `Set(OclState)` and the elements of `T` denote state configurations, we can make use of operation `startsWith()` to formulate restrictions over State Diagram execution paths, using the syntax for configuration sequences as illustrated under item 4.

   Using operation `startsWith()` is similar to selecting a subsequence with the standard OCL operation `subSequence(a:Integer,b:Integer)` and then matching the extracted subsequence with `g`. But unfortunately, we cannot a priori provide an upper bound `b` from our particular viewpoint of possibly infinite execution runs, such that we cannot make use of existing OCL operations.

For the sake of completeness, Tables D.1, D.2, D.3, and D.4 in Appendix D provide corresponding temporal OCL expressions for the other main property specification patterns.

Expressing Assumptions with OCL. To capture the additional assumptions we make concerning the occurrence of scope delimiters, different approaches are imaginable. One idea uses only standard OCL language concepts. The expression

```
if <assumption> then
                  <pattern>
                else
                  OclUndefined
                endif
```

makes use of the three-valued logic of OCL that includes `OclUndefined` as the third logical value. For example, the complete OCL invariant for pattern '`P is false after Q`' is defined as follows.

```
inv: if @post()->forAll(g | g->includes(Q)) then
          self.oclInConf(Q)
          implies
          self@post()->forAll(g | g->excludes(P))
       else
         OclUndefined
       endif
```

Note that OCL has a three-valued logic, i.e., OCL type Boolean actually comprises the values `true`, `false`, and `OclUndefined`. In the expression above, `OclUndefined` is returned when the if-condition does not hold. Unfortunately, such an expression cannot directly be mapped to a temporal logic like CTL or LTL due to a missing third logical value.

Another idea is to extend OCL and introduce a dedicated new clause, e.g., named `assume`, to express an assumption in the same manner as a precondition of an operation. For instance, the assumption '`R becomes true on all paths`' can then be expressed by

```
assume: @post()->forAll(g | g->includes(R)) .
```

Similarly, it has already been suggested by other authors to introduce means to formulate *exceptions* with OCL, such that undesired situations can be specified [SF99] and dealt with [SS01]. We can make use of such an approach to specify a corresponding exception for each assumption simply by negating the assumption expression. When the exception evaluates to true, the assumption does not hold, and the respective pattern cannot be validated.

The advantage of this approach is that such assumption and exception expressions can directly be mapped to temporal logic formulae for further usage in verification tools.

7.3 Mapping to the Temporal Logics CCTL

In this section, we provide a mapping from instances of `FutureTemporalExpCS` to temporal logics formulae. We here concentrate on CCTL as described in Subsection 3.3, but it is also possible to derive similar mappings to other future-oriented temporal logic formulae, e.g., dense-time TCTL or timed LTL formulae. The mapping of temporal OCL constraints to temporal logics depends on the formal underlying model. As we consider I/O-Interval Structures as the formal model in the context of this thesis, we map temporal OCL constraints to the corresponding temporal logics CCTL. The relation of CCTL and I/O-Interval Structures is defined by a satisfaction relation that is described in Section 3.6.2 on page 77.

Representation of OCL states in CCTL. For a state specification in OCL, a corresponding representation of that state in CCTL has to be given. In this context, we have to consider how an *activated* OCL state of a State Diagram is represented in I/O-Interval Structures. Recall that composite OCL states `s` of objects `objectId` are translated into separate I/O-Interval Structures $IS_{objId,s}$ (or, in RIL syntax, `IS_[[objId]]_[[s]]`, respectively) (cf. Section 5.4.1.1) and that an additional internal boolean variable called *activated* is introduced to that I/O-Interval Structure to indicate whether the composite state is currently activated or not.

Given a state specification `stateName` for an object `objId` within an OCL expression, the corresponding CCTL formula is

```
(   (IS_[[objId]]_[[parent(stateName)]].state = [[stateName]])
  & (IS_[[objId]]_[[parent(stateName)]].activated = true) ) .
```

In this formula, `[[parent(stateName)]]` denotes the direct composite superstate of `stateName`. If `[[parent(stateName)]]` is the topmost state, this is the corresponding class name of `objId`.

A complete state configuration is then built by conjunction of all corresponding CCTL state formulae.

Mapping of Temporal OCL Expressions. By definition, OCL invariants for a given class must be true for all its instances at any time [OMG03b, Section 7.3.3]. In the context of (time-based) traces, this means that the invariant expression must be true on all possible traces at each position. Consequently, corresponding CCTL formulae have to start with the `AG` operator ('On **A**ll paths **G**lobally'), i.e., the expression following `AG` must be true on all possible future execution paths at all times.

Table 7.2 lists OCL operations that directly match to CCTL expressions. In that table, `expr` denotes a Boolean OCL expression. `cctlExpr` is the equivalent Boolean expression in CCTL syntax. `cfg` denotes a valid configuration and `cctlCfg` is the corresponding set of states in CCTL syntax. `p` and `c` are iterator variables for traces and configurations, respectively.

Consider, for example, the last row of Table 7.2. When taking the particular interval `[1,100]` and a configuration from Figure 7.1 for `cfg`, the resulting OCL expression is:

```
inv: obj@post(1,100)->forAll( trace | trace->includes(Set{X::A::L,X::B::N}))
```

We read that formula as: At any time, given the current configuration of the State Diagram associated to object `obj`, all future traces `p` starting from the current configuration reach – at a certain point of time within the next 100 time units – the configuration represented by `Set{X::A::L,X::B::N}`.

Note that with the CCTL formulae of Table 7.2 we can only investigate models with 'persistent' active objects, i.e., corresponding objects must exist from the initial system state onwards during the complete execution time. Otherwise, we have to determine the maximal number of created objects for a model *in advance*. Only then we are able to build a corresponding set of communicating finite state machines by means of I/O-Interval Structures for each object.

Dynamic object creation and deletion is not addressed in this work. However, an idea to represent this feature is to introduce additional variables within the according I/O-Interval Structures, e.g., by a Boolean variable `obj1.isAlive` for an object `obj1`. The value of that variable is then additionally checked in the CCTL formulae of the mapping. E.g., in the example above, the resulting CCTL formula is

```
AG( obj1.isAlive →
    A(obj1.isAlive U_[1,100]
        ( !obj1.isAlive | ( obj1.isAlive &
                            obj1.S_X_A = L &
                            obj1.S_X_B = N)
        )))
```

For mapping trace literal expressions, let $e_1, e_2, \ldots, e_n$ be the trace literal parts of `TraceLiteralExpCS` with timing intervals $[a_i, b_i]$, $1 \leq i \leq n-1$. The temporal OCL expression

`inv: obj@post(a,b)-> includes ( Sequence`$\{e_1[a_1,b_1], e_2[a_2,b_2], \ldots, e_n\}$ `)`

maps to CCTL applying the *strong until* temporal operator (i.e., $expr_1\ \underline{\mathtt{U}}_{[a,b]}\ expr_2$ requires that $expr_1$ must be true between a and b time units until $expr_2$ becomes true) as follows:

$\mathtt{AG}_{[a,b]}\ \mathtt{EF(\ E(}e_1\ \underline{\mathtt{U}}_{[a_1,b_1]}\ \mathtt{E(}e_2\ \underline{\mathtt{U}}_{[a_2,b_2]}\ \mathtt{E(...\ E(}e_{n-1}\ \underline{\mathtt{U}}_{[a_{n-1},b_{n-1}]}\ e_n\mathtt{)\ ...))))}$

Note here that the path quantifier, which is applied to each sequence element, depends on the preceding operations. Though we have given only some examples here, more complex formulae can be easily derived from the above.

Table 7.2 lists temporal OCL operations that directly match to CCTL expressions. In that table, `expr` is an OCL expression and `configuration` is a set of OCL states that denote a state configuration. The table gives a translation by templates and can easily be applied to form more complex expressions as well.

Table 7.2: Mapping Temporal OCL Expressions to CCTL Formulae

Temporal OCL Expression	**CCTL Formula**
inv: obj@post(a,b)→exists(p | p→forAll(c | expr))	AG EG$_{[a,b]}$(cctlExpr)
inv: obj@post(a,b)→exists(p | p→exists(c | expr))	AG EF$_{[a,b]}$(cctlExpr)
inv: obj@post(a,b)→exists(p | p→includes(cfg))	AG EF$_{[a,b]}$(cctlCfg)
inv: obj@post(a,b)→forAll(p | p→forAll(c | expr))	AG AG$_{[a,b]}$(cctlExpr)
inv: obj@post(a,b)→forAll(p | p→exists(c | expr))	AG AF$_{[a,b]}$(cctlExpr)
inv: obj@post(a,b)→forAll(p | p→includes(cfg))	AG AF$_{[a,b]}$(cctlCfg)

7.4 Temporal OCL Queries

Corresponding to the recently introduced notion of OCL 2.0 as a general expression and query language, we can also investigate objects in a model under the aspects of execution times needed (at least or at most) to get from one configuration to another. We here only provide an informal description of the proposed OCL operations. They can directly be mapped to the corresponding RAVEN analysis queries presented in Section 3.6.3.

```
OclAny::minStableTime(cfg:Set(OclState)) : Integer
  -- Returns the minimal time during which the corresponding State Diagram remains
  -- in the given configuration cfg.
  -- Returns 0 if never configuration cfg is never entered.
  -- Returns OclUndefined if the configuration is entered only once at remains
  -- forever in that configuration.
```

This operation refers to the RAVEN analysis query `MIN STABLE TIME OF (x)`, where `x` represents the state configuration `cfg` in CCTL syntax. For maximal stable times, the operation signature is very similar:

```
OclAny::maxStableTime(cfg:Set(OclState)):Integer
  -- Returns the maximal time during which the corresponding State Diagram remains
  -- in the given configuration cfg.
  -- Returns 0 if the configuration is never entered.
  -- Returns OclUndefined if the State Diagram can remain infinitely long in the
  -- configuration cfg.
```

Finally, the two following operations represent analysis queries that extract the minimal and maximal transition times between two configurations. Again, they can directly be mapped to RAVEN analysis queries of the form `MIN TIME OF FROM (x) TO (y)` and `MAX TIME OF FROM (x) TO (y)`, respectively.

```
OclAny::minTransitionTime(cfg1:Set(OclState), cfg2:Set(OclState)) : Integer
  -- Returns the minimal time of getting from configuration cfg1 to configuration
  -- cfg2.
  -- Returns 0 if the configuration cfg1 is never entered.
  -- Returns OclUndefined if configuration cfg2 is never entered.

OclAny::maxTransitionTime(cfg1:Set(OclState), cfg2:Set(OclState)) : Integer
  -- Returns the maximal time of getting from configuration cfg1 to configuration
  -- cfg2.
  -- Returns 0 if the configuration cfg1 is never entered.
  -- Returns OclUndefined if configuration cfg2 is never entered.
```

7.5 Related Work

In this section, we give an overview on proposals that either extend OCL to enable specification of temporal constraints or find another way to express real-time constraints in the context of UML. A more detailed comparison of temporal OCL extensions can be found in [FM02e, Fla03b].

Ramakrishnan et al. [RM99, RM00] extend OCL by additional rules with unary and binary temporal operators, e.g., `always` and `never` to specify safety and liveness properties. A very similar approach in the area of business modeling that also considers past temporal operators is published by Conrad and Turowski [CT00, CT01]. However, general user-defined operations are allowed in the temporal expressions of these works, whereas in standard OCL, only query operations may be used. Moreover, the resulting syntax of these works does not combine well with standard OCL, as temporal expressions appear to be similar to temporal logics formulae.

Kleppe and Warmer [KW00] introduce a so-called action clause to OCL. Basically, action clauses enable modelers to specify required (synchronous or asynchronous) executions of operations or dispatching of events. Similarly, the OCL 2.0 specification introduces message expressions [OMG03b].

Distefano et al. [DKR00] define BOTL (Object-Based Temporal Logic) in order to facilitate the specification of static and dynamic properties. BOTL is not directly an extension of

OCL; it rather maps a subset of OCL into object-oriented Computational Tree Logic (CTL). Syntactically, BOTL looks very similar to temporal logics formulae in common CTL.

Bradfield et al. [BKS02] extend OCL by useful causality-based templates for dynamic constraints. Basically, a template consists of clauses, the cause and the consequence. The cause clause starts with the keyword *after*, followed by a boolean expression, while the consequence is one of *eventually, immediately, infinitely* etc., followed by an OCL expression. The templates are formally defined by a mapping into observational mu-calculus, a two-level temporal logic, using OCL as the lower level logic.

Ziemann and Gogolla [ZG02, ZG03] present an OCL extension, in which future-oriented temporal development of attribute values and existence of objects and links can be restricted. Similar to other approaches, temporal operators like `always`, `next`, and `sometime` are introduced. For defining a formal semantics, they build upon the set-theoretic OCL semantics developed by Richters [Ric01] and define *traces*, i.e., sequences of system states. Such a trace employs a high-level notion of the development of a running system with only that information which is necessary to evaluate OCL expressions.

Note that none of the approaches mentioned so far considers real-time constraints. Besides the rudimentary and informal UML modeling elements described in Subsection 2.4 (in particular, time expressions attached as comments without semantics), we know of the following approaches.

The work presented by Roubtsova et al. [RvTdR01, RT01] defines a UML profile with stereotyped classes for dense time as well as parameterized specification templates for deadlines, counters, and state sequences. Each of these templates has a structural-equivalent dense-time temporal logics formula in TCTL (Timed Computation Tree Logic). Roubtsova et al. do not extend OCL on purpose, as they argue that *"... OCL has no path notion. Any extension of OCL to present properties of computation paths breaks the idea of the language and makes it eclectic"*. In contrast to this, we think that the notion of execution paths can be introduced to OCL, as shown in the next sections.

Sendall and Strohmeier [SS01, SS02b] introduce timing constraints on state transitions in the context of a restricted form of UML protocol statemachines called SIP (System Interface Protocol). A SIP defines the temporal ordering between operations. Five time-based attributes on state transitions are proposed, e.g., (absolute) completion time, duration time or frequency of state transitions. Using these attributes, one can then relate actions to timing constraint failures in an extended form of transition condition (or, in UML terms: transition guard).

Cengarle and Knapp [CK02] present OCL/RT, a temporal extension of OCL with modal operators `always` and `sometime` over event occurrences. These can be used for specifying deadlines and timeouts of operations and reactions on received signals. On the metamodel, events are equipped with time stamps by introducing a metaclass `Time` with attribute `now` to refer to the time unit at which an event occurs. In turn, each instance can access the set of current associated events at each point of time, i.e., at each *system state*.

Table 7.3 lists the mentioned approaches. The last row of Table 7.3 refers to publications this thesis is based upon.

Those approaches, for which a formal semantics is provided, all have formal verification by model checking in mind. Formal verification by theorem proving using OCL is investigated in the KeY project. That approach aims to facilitate the use of formal verification for software

Table 7.3: Temporal OCL Extensions and Real-Time Specification

Approach	Syntax	Formal Semantics	Real-Time
Ramakrishnan et al. [RM99]	OCL + temp. operators	–	no
Conrad/Turowski [CT00, CT01]	OCL + temp. operators	–	no
Kleppe/Warmer [KW00]	OCL + action clause	–	no
Distefano et al. [DKR00]	CTL + OCL subset	BOTL	no
Bradfield et al. [BKS02]	OCL + template clauses	Observational mu-calculus	no
Ziemann/Gogolla [ZG02, ZG03]	OCL + temp. operators	Trace semantics	no
Roubtsova et al. [RvTdR01, RT01]	Stereotyped classes	TCTL	yes
Sendall/Strohmeier [SS01]	OCL consistent	–	yes
Cengarle/Knapp [CK02]	OCL + temp. operators	Trace semantics	yes
Flake/Mueller [FM02c, FM02d]	OCL consistent	CCTL	yes

specifications [ABB$^+$00]. Here, OCL is applied without modifications to specify constraints on design patterns. As standard OCL currently has no formal semantics, this approach translates OCL specifications to dynamic logic (DL), an extension of Hoare logic [Hoa69]. DL is used as input for formal verification by theorem proving.

7.6 Implementation

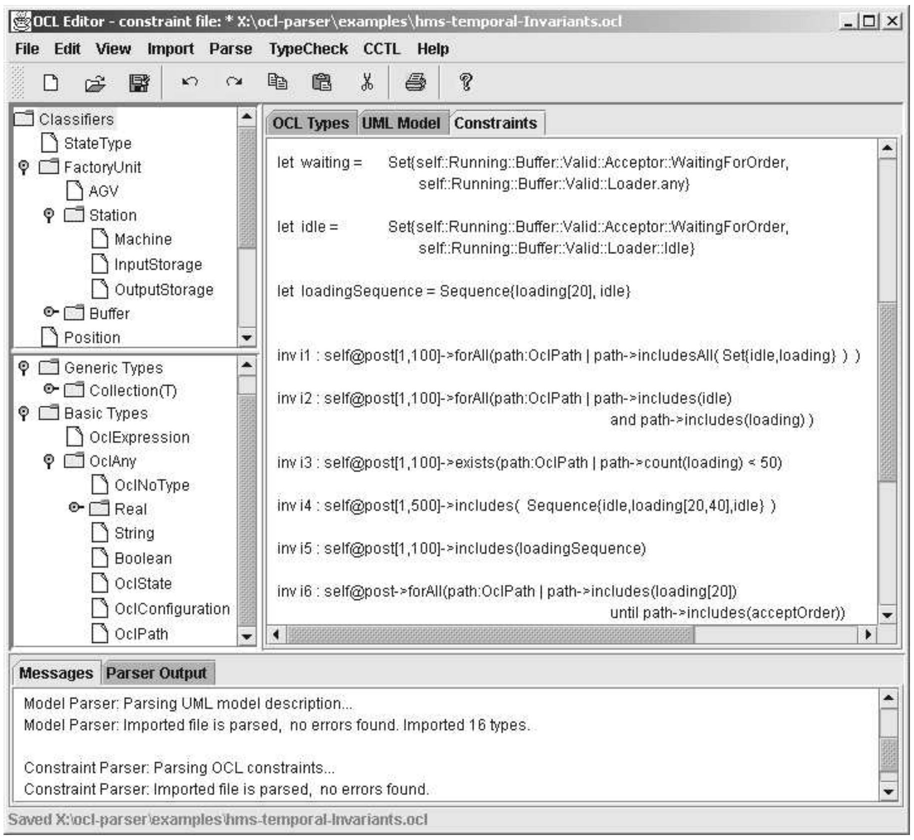
The temporal extensions as presented here are integrated into a prototype OCL parser and type checker (see Figure 7.4). The checker is implemented in Java 1.3 using Swing components. The visual capture loads and edits OCL types, model descriptions, and OCL constraints in parallel. The parsers are implemented with JavaCC[3] based on an early implementation of OCL Version 1.1 [War97]. Correctly parsed types are integrated into type tree structures. Class models and State Diagrams are currently modeled by textual means. For this, we have implemented a system to parse textual descriptions of class models and State Diagrams.

7.7 Contributions of the Chapter

This chapter provides the following contributions:

- A UML Profile for an extension of OCL is developed that allows to specify state-oriented real-time constraints. The profile builts upon the OCL 2.0 metamodel. Existing OCL concepts, such as invariants, sets, sequences, states, are re-used whenever possible. This meets Requirement 7.1.

[3]http:www.webgain.com



```
let waiting =     Set{self::Running::Buffer::Valid::Acceptor::WaitingForOrder,
                      self::Running::Buffer::Valid::Loader.any}

let idle =        Set{self::Running::Buffer::Valid::Acceptor::WaitingForOrder,
                      self::Running::Buffer::Valid::Loader::Idle}

let loadingSequence = Sequence{loading[20], idle}


inv i1 : self@post[1,100]->forAll(path:OclPath | path->includesAll( Set{idle,loading} ) )

inv i2 : self@post[1,100]->forAll(path:OclPath | path->includes(idle)
                                                   and path->includes(loading) )

inv i3 : self@post[1,100]->exists(path:OclPath | path->count(loading) < 50)

inv i4 : self@post[1,500]->includes( Sequence{idle,loading[20,40],idle} )

inv i5 : self@post[1,100]->includes(loadingSequence)

inv i6 : self@post->forAll(path:OclPath | path->includes(loading[20])
                                                   until path->includes(acceptOrder))
```

```
Model Parser: Parsing UML model description...
Model Parser: Imported file is parsed,  no errors found. Imported 16 types.

Constraint Parser: Parsing OCL constraints...
Constraint Parser: Imported file is parsed,  no errors found.
```

Saved X:\ocl-parser\examples\hms-temporal-Invariants.ocl

Figure 7.4: OCL Parser and Type Checker

Syntactically, the temporal OCL extension builts upon the concrete syntax of the OCL 2.0 specification. Semantically, the main new concept is the temporal expression that is modeled as a special kind of operation call.

- A separate section illustrates that the proposed OCL extension has the expressive power to express all kinds of specifications that are regarded as relevant in practice, based upon the specification pattern library by Dwyer et al. [DAC98a]. This meets Requirement 7.2.

- The semantics of state-oriented real-time OCL expressions is defined by means of a high-level time-based trace semantics. This conforms to Requirement 7.3.

  Additionally, a mapping of state-oriented real-time constraints to formulae of the time-annotated temporal logics CCTL is given. The CCTL formulas are built w.r.t. the mapping of timed UML State Diagrams to I/O-Interval Structures as presented in Section 5.4

- Some new useful OCL operations are proposed, in particular operations that reason about minimal and maximal times between state configurations.

  For further application, these operations can directly be mapped and applied in the model checker RAVEN.

Chapter 8

Manufacturing Case Study

The oldest, shortest words – yes and no –
are those which require the most thought.
– Pythagoras

This chapter addresses the translation of an MFERT model in the context of the manufacturing case study that has been presented in Section 1.2. We focus on early stages of system development and abstract from specific implementation issues, such as bidding among stations and AGVs (Automated Guided Vehicles) to select the most appropriate AGV for a transport. Recall that the formal MFERT model as presented in Chapter 6 represents production elements (i.e., material and resources) as data elements within PENs (Production Element Nodes) and PPNs (Production Process Nodes), such that production elements, e.g., AGVs, do not have an own thread of control in the context of MFERT.[1] The MFERT model presented in this chapter focuses on the time-constrained production progress of PPNs, and we abstract from

- the grid of positions along which the AGVs are moving,
- the bidding scheme that determines the most appropriate AGV for a transport, and
- unique identifiers for production items and resources.

One effect of these abstractions is that we cannot reason about the time a specific production item, e.g., an engine, needs from entering the manufacturing system until it is fully processed. In subsequent stages of system development, such parameters have to be included in the model, but formal analysis and verification then of course becomes more complex, if not too complex for formal verification tools like model checkers. Experiences with MFERT models and I/O-Interval Structures that also capture information about the grid positions are described in [DDF$^+$02, Ruf02]. Those works focus on the specification and verification of models that guarantee collision-free AGV movements in the context of the case study.

[1]In contrast, the holonic manufacturing systems (HMS) approach [WHS94] interprets AGVs – among other parts of manufacturing systems – as *holons*. Holons are originally seen as autonomous and cooperative entities with fixed rules and flexible strategies [Koe67]. In the HMS approach, holons are redefined to be autonomous and cooperative production units consisting of a software-based information processing component and a physical part [Dee03].

Nevertheless, the MFERT model presented in the remainder makes use of the timed State Diagram notation of Chapter 5 and is appropriate to demonstrate the applicability of the time-bounded state-oriented OCL extension of Chapter 7. The remainder of this chapter is divided into two sections. In Subsection 8.1, the case study-based MFERT model is shown and its translation to I/O-Interval Structures is outlined. Though the translation has been performed by hand, the provided code should demonstrate that an automated translation is possible. In this context, a graphical MFERT editor has recently been developed that maps very similar MFERT models to I/O-Interval Structures [Zab03] (see also Section 6.3.5). In that tool, PPNs are specified as finite state machines by means of rule tables. The tool is mainly used for simulation and analysis purposes, and property specification means to support formal verification by model checking have still to be integrated. Subsection 8.2 then presents some typical constraints expressed by time-bounded state-oriented OCL invariants. In each case, a corresponding CCTL formula is given that has been verified over the I/O-Interval Structures outlined in Subsection 8.1.

8.1 The MFERT Model

Though the timed State Diagram variant that has been presented in Chapter 5 is employed for modeling the timed behavior of PPNs, the general mapping to I/O-Interval Structures is simplified here w.r.t. the input queues of waiting signals and operations. This can be done because of the syntactical restrictions in MFERT models, i.e., the bipartite structure that demands that all nodes a PPN communicates with are PENs. PENs, in turn, immediately process their incoming messages synchronously in a reactive manner (cf. the semantics defined in Sections 6.3.2 and 6.3.4), such that corresponding replies are sent back to PPNs in the next time step. When PPNs determine their next actions based on these replies, they 'know' what incoming messages to wait for. I.e., all communications between PPNs and PENs take place as a pair of a request and a corresponding reply (which is either a grant for access or denial). This allows to omit explicit input queues in the I/O-Interval Structures of MFERT models.[2]

Many parts of the mapping of MFERT nodes to I/O-Interval Structures result in very similar RIL code (RAVEN Input Language code), such that we only list the code of representative (parts of) I/O Interval Structures here. For example, the PPNs for `SupplyingEngines`, `Milling`, `Drilling`, and `Washing` mainly differ in the times that represent the processing of an item, such that it is sufficient to describe one PPN, e.g., `SupplyingEngines`, in more detail. Figure 8.1 shows those parts of the MFERT model that are further regarded in the remainder. That figure is an excerpt of the MFERT graph shown in Chapter 6 on page 146, but note that PENs are now annotated with

- concrete values for their shifting intervals,
- the size of input and output sequences, and
- initial values when appropriate.

[2]However, in other modeling domains, the input queue of the timed UML State Diagram variant is essential. For example, the compositional verification approach of real-time UML designs published in [BFG$^+$03] could already make use of this work in the context of a shuttle railway system.

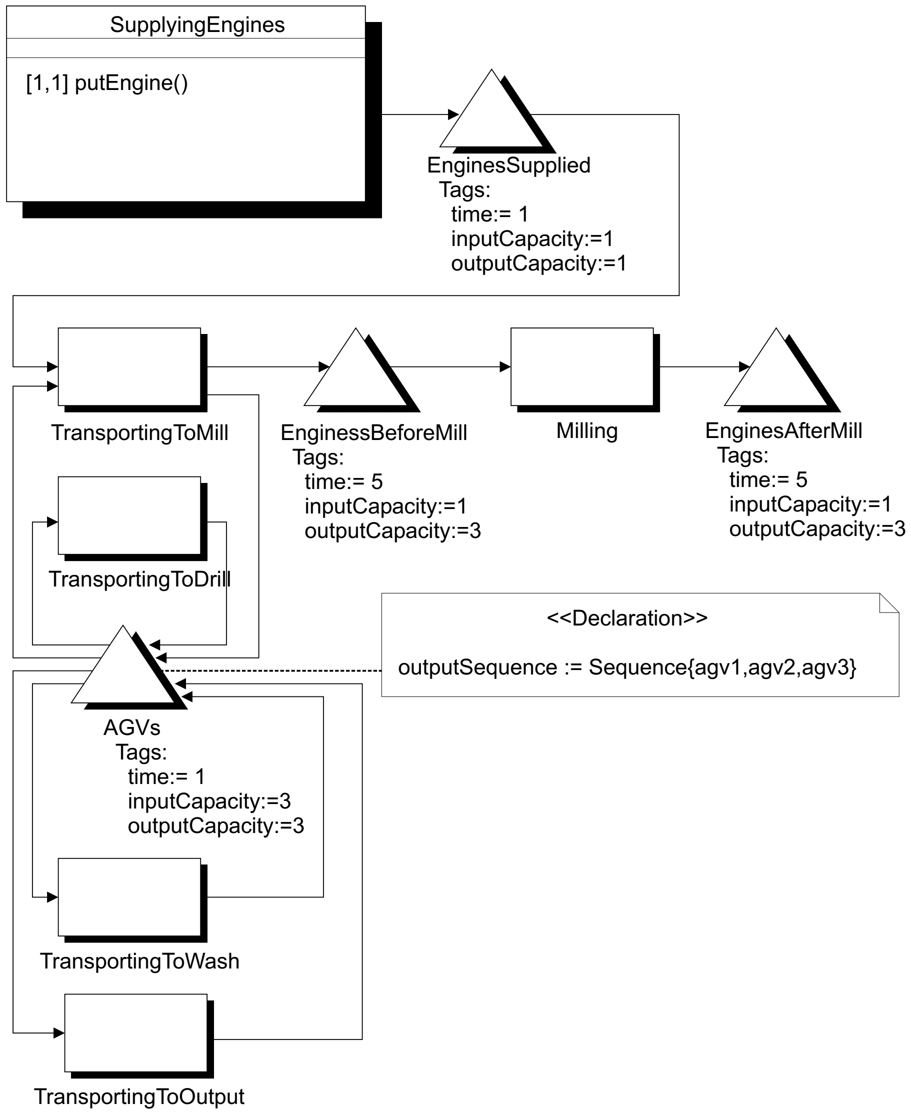


Figure 8.1: Selected Parts of the MFERT Graph of the Case Study (cf. Figure 6.1)

For example, PEN `AGVs` is initialized with 3 AGVs available for transports. If not explicitly specified, the sequences of PENs are initially empty by default. Moreover, the operations on PPN `SupplyingEngines` are shown, but operations of the other PPNs are hidden to remain concise.

PPNs to Process Items. First of all, PPN `SupplyingEngines` is used as a generator to fill the PEN `EnginesSupplied` each time a production item is taken out of the output sequence of that PEN, i.e., the input storage that keeps the items that are to be processed will never become empty. The corresponding timed State Diagram is shown in Figure 8.2 and the RIL code of the I/O-Interval Structure that is derived from that State Diagram is shown below. Note that due to the simple structure of the State Diagram there is no need to take care about activation and deactivation of composite substates. We also abstract from a timed activity to physically get an

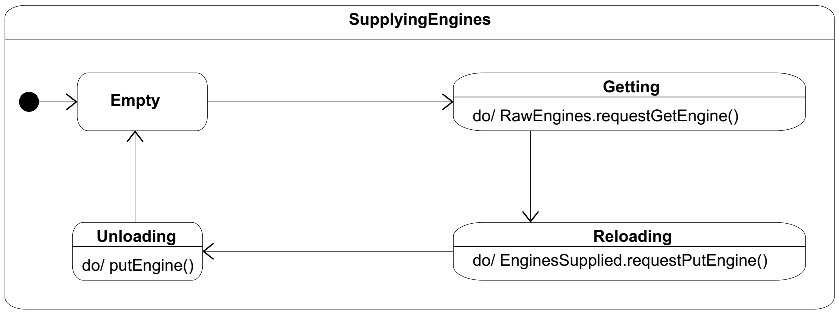

SupplyingEngines
Empty
Getting
do/ RawEngines.requestGetEngine()
Reloading
do/ EnginesSupplied.requestPutEngine()
Unloading
do/ putEngine()

Figure 8.2: PPN SupplyingEngines

item from PEN `RawEngines`, i.e., after an acknowledgement by `RawEngines.ackGetEngine`, we immediately shift that item to the subsequent PEN. The activity to physically put an item into the input sequence of the subsequent PEN `EnginesSupplied` is assumed to be performed in one time unit. The synchronous UML operation calls to the surrounding PENs `RawEngines` and `EnginesSupplied` are mapped to request signals and corresponding reply signals in the I/O-Interval Structure, as shown below.

```
MODULE SupplyingEngines
  SIGNAL
    state : { empty getting reloading unloading }
  INPUT
    ackGetEngine := RawEngines.ackGetEngine
    ackPutEngine := EnginesSupplied.ackPutEngine
  DEFINES
    requestGetEngine  := (state==getting)
    requestPutEngine  := (state==reloading)
  INIT
    (state==empty)
  TRANS
    |- (state==empty)     --                --> state:=getting
    |- (state==getting)   -- ackGetEngine   --> state:=reloading
                                            !-> state:=getting
    |- (state==reloading) -- ackPutEngine   --> state:=unloading
                                            !-> state:=reloading
    |- (state==unloading) --             :1 --> state:=empty
END
```

PPNs `Milling`, `Drilling`, and `Washing` are modeled in a very similar way, such that the resulting I/O-Interval Structures do not differ much from the one shown above. The main difference is an additional state called `working`. That additional state is equipped with an activity to model the time to physically process (e.g., wash) the current item.

Transporting PPNs. A State Diagram for PPN `TransportingToMill` is shown in Figure 8.3. It basically consists of a chain of activities to perform – such a PPN is thus controlling the time-dependent activities of an AGV object. The activities are initiated by operation calls and refer to the occupied AGV, such as `move()`, `load()`, and `unload()`. Recall that we allow to

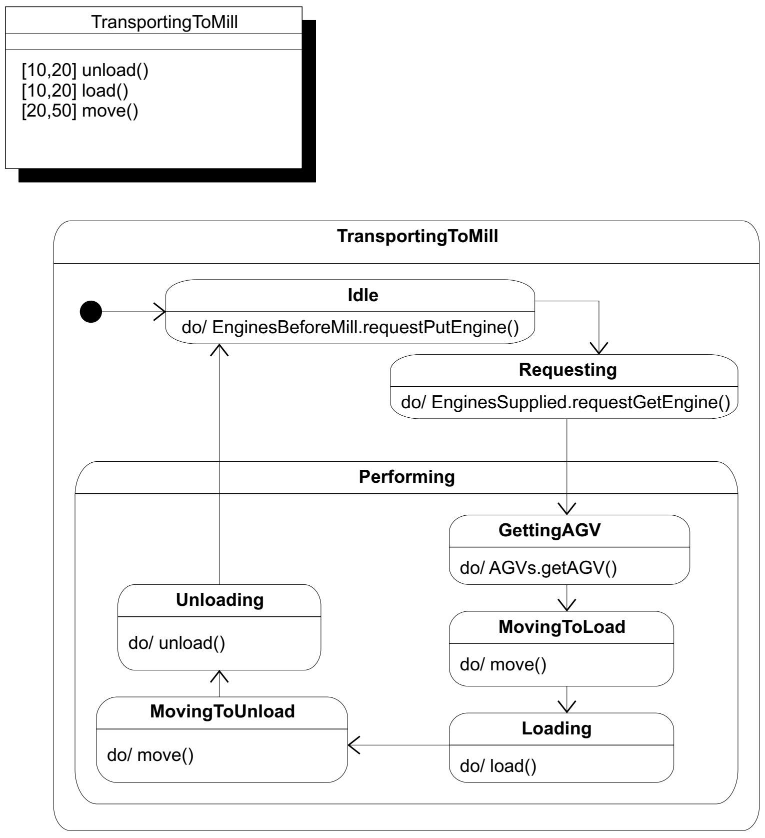


Figure 8.3: PPN TransportingToMill and its State Diagram

associate (estimated) execution times to these operations (see the PPN on top of Figure 8.3 and also consider Figure 5.1 on page 117). These timing specifications depend on the machines and the physical topology of the manufacturing system, such as the speed of the machines/AGVs to process/transport an item and the distances between the stations to deliver items. In the case study example, activities initiated by operations `load()` and `unload()` take between 10 and 20 time units, while the activity initiated by `move()` take between 20 and 50 time units, depending on the distances and the need of detours to avoid collisions.

According to the translation in Section 5.4, the composite substate `Performing` is getting an own I/O-Interval Structure and the interlevel transitions must be coordinated among the two I/O-Interval Structures to be generated. For our mapping, let `tr1` be the name of the transition with source state `Requesting` and target state `GettingAGV`, and let `tr2` denote the transition with source state `Unloading` and target state `Idle`. The outermost state named after its PPN `TransportingToMill` comprises the three direct substates `idle`, `requesting`, and `performing`. When entering the `performing` state, output signal `executed_tr1` is set to indicate that the I/O-Interval Structure that models the substate `Performing` has to be 'activated', i.e., the according internal variable `activated` has to be set to true. In turn, `TransportingToMill` leaves its state `performing`, when the input signal `fire_tr2` becomes true. That signal is initiated by the I/O-Interval Structure that models composite state `Performing`.

```
MODULE TransportingToMill
  SIGNAL
    state : { idle requesting performing }
  INPUT
    ackRequestGetEngine := EnginesSupplied.ackRequestGetEngine
    ackRequestPutEngine := EnginesBeforeMill.emptyInputBuffer
    fire_tr2            := TransportingToMill_performing.executed_tr2
  DEFINES
    executed_tr1     := (state==requesting) & ackRequestGetEngine
    requestPutEngine := (state==idle)
    requestGetEngine := (state==requesting)
  INIT
    (state==idle)
  TRANS
  |- (state==idle)       -- ackRequestPutEngine  --> state:=requesting
                                                 !-> state:=idle
  |- (state==requesting) -- ackRequestGetEngine  --> state:=performing
                                                 !-> state:=requesting
  |- (state==performing) -- fire_tr2             --> state:=idle
                                                 !-> state:=performing
END
```

For the composite substate `Performing`, we here list the resulting I/O-Interval Structure, but we omit the transitions for the cases `(state==loading) & (activated==true)` and `(state==movingToUnload) & (activated==true)`, as they are very similar to the other two cases with states `unloading` and `movingToLoad`. Note the non-deterministic transitions in the following code, e.g., the ones with condition `(count>=20) & (count<50)`. At any time between the minimal and maximal time bound, the transition to the next state can be taken. This reflects the timing intervals specified for the operations.

```
MODULE TransportingToMill_performing
  SIGNAL
    state     : { gettingAGV loading unloading movingToLoad movingToUnload }
    activated : BOOL
    count     : RANGE[0,50]
    movingToLoad_finished   : BOOL  // additional variables to synchronize with
    loading_finished        : BOOL  //                PENs and the parent state
    movingToUnload_finished : BOOL
    unloading_finished      : BOOL

  INPUT
    ackGetAGV  := AGVs.ack_transportingToMill
    fire_tr1   := TransportingToMill.executed_tr1

  DEFINES
    requestGetAgv := (state==gettingAGV)  & (activated==true)
    requestPutAgv := (unloading_finished) & (activated==true)
    getEngine     := (loading_finished)   & (activated==true)
    putEngine     := (unloading_finished) & (activated==true)
    executed_tr2  := (unloading_finished) & (activated==true)
  INIT
    (state==gettingAGV) & (activated==false) & (count==0)
    & (movingToLoad_finished==false) & (loading_finished==false)
    & (movingToUnload_finished==false) & (unloading_finished==false)
  TRANS
    |- (activated==false) -- fire_tr1 --> state:=gettingAGV; activated:=true;
                                          movingToUnload_finished:=false
                                      !-> state:=state;      activated:=false;
                                          movingToUnload_finished:=false

    |- (state==gettingAGV) & (activated==true)
       -- ackGetAGV                --> state:=movingToLoad; activated:=true; count:=0
                                   !-> state:=gettingAGV;   activated:=true

    |- (state==movingToLoad) & (activated==true)
       -- (count<20)               --> state:=movingToLoad; count:=count+1
       -- (count>=20) & (count<50) --> state:=movingToLoad; count:=count+1
       -- (count>=20) & (count<50) --> state:=loading; count:=0;
                                       movingToLoad_finished:=true
       -- (count==50)              --> state:=loading; count:=0;
                                       movingToLoad_finished:=true
    ...
    |- (state==unloading) & (activated==true)
       -- (count<10)               --> state:=unloading; count:=count+1;
                                       movingToUnload_finished:=false
       -- (count>=10) & (count<20) --> state:=unloading; count:=count+1
       -- (count>=10) & (count<20) --> state:=unloading; activated:=false; count:=0;
                                       unloading_finished:=true
       -- (count==20)              --> state:=unloading; activated:=false; count:=0;
                                       unloading_finished:=true
END
```

PEN for Automated Guided Vehicles. The PEN `AGVs` does not make a difference between input and output sequence, as the specified shifting time is 1, such that the waiting time between shifts is zero (determined by `wait(1-1)` according to the semantics of PENs). This means that each AGV resource that is released can immediately be accessed for another transport in this case. As a consequence, the most complex structure of this case study is built for the PEN `AGVs`, as shifting items might have to be performed at the same point of time as different inputs from associated PPNs for transports occur.

The mapping of requests and actions for producing items or releasing resources (indicated by prefix `put` in the code below) follows the following approach: Prior to actually putting an item into a PEN p, the preceding PPN checks whether there is space in the input sequence of p. Thus, no input buffer overflows should occur. However, to verify this formally, a corresponding internal boolean variable `error` is used as a monitor w.r.t. the size of the input sequence. The variable `error` becomes true when more `put`s occur than there is space left in the input sequence. It can easily be checked by a dedicated safety formula that no overflow occurs. For example, the corresponding CTL safety constraint for PEN `AGVs` is: `AG !(AGVs.error)`.

When a PPN requests to put an item into the input sequence of a PEN and there is currently no space in that sequence, the PPN is waiting until the input buffer has an available position. In the corresponding I/O-Interval Structure, this synchronization is achieved by mapping the synchronous UML operation call to repeated sendings of a signal until a positive reply signal is received.

Mapping of requests and actions to consume items or occupy resources is indicated by prefix `get` in the code below and follows the same approach.

For PEN `AGVs`, basically $2^8 = 256$ combinations for 8 potentially parallel input signals (4 transporting PPNs, each with two kinds of requests) have to be considered over 0 up to 3 potentially available AGVs. Most of these combinations (to be precise, 806) are not valid, as getting and putting an AGV by the same PPN at the same point of time is not allowed. There remain 218 transitions to consider; we list only some typical examples in the code and the omitted lines are indicated by dots.

```
MODULE AGVs
  SIGNAL
    count : RANGE[0,3]              // at most 3 available AGVs
    error : BOOL                    // internal error variable
    ack_1 : BOOL   ack_2 : BOOL     // acknowledgement signals
    ack_3 : BOOL   ack_4 : BOOL
  INPUT
    // input signals to release AGVs after performing a transport:
    putAgv_1 := TransportingToMill.putAgv
    putAgv_2 := TransportingToDrill.putAgv
    putAgv_3 := TransportingToWash.putAgv
    putAgv_4 := TransportingToOutput.putAgv
    // input signals to request AGVs for transports:
    getAgv_1 := TransportingToMill.getAgv
    ...
  DEFINES
    ack_TransportingToMill   := ack_1     // output signals that grant requests
    ack_TransportingToDrill  := ack_2
    ack_TransportingToWash   := ack_3
    ack_TransportingToOutput := ack_4
  INIT
    // initially, 3 AGVs are available, all other signals are false:
    (count == 3)     & (error==false)
    & (ack_1==false) & (ack_2==false) & (ack_3==false) & (ack_4==false)
  TRANS
    |- (count==0) & (error==false)
        -- !putAgv_1 & !putAgv_2 & !putAgv_3 & !putAgv_4
           &  getAgv_1 &  getAgv_2 &  getAgv_3 & !getAgv_4
        --> count:=count; ack_1:=false; ack_2:=false; ack_3:=false; ack_4:=false
        -- !putAgv_1 &  putAgv_2 &  putAgv_3 &  putAgv_4
           &  getAgv_1 & !getAgv_2 & !getAgv_3 & !getAgv_4
        --> count:=count+2; ack_1:=true; ack_2:=false; ack_3:=false; ack_4:=false
    ...
    |- (count==3) & (error==false)
       -- !putAgv_1 & !putAgv_2 & !putAgv_3 & !putAgv_4
          & !getAgv_1 & !getAgv_2 &  getAgv_3 &  getAgv_4
       --> count:=count-2; ack_1:=false; ack_2:=false; ack_3:=true; ack_4:=true
    ...
END
```

8.2 Real-Time OCL Constraints and CCTL Formulae

In this subsection, we provide some typical time-bounded constraints that are applicable to the MFERT model described in the previous section. Although we here focus on the PPN `TransportingToMill`, several similar constraints are employed for other PPNs. Note that we have not modeled concurrent State Diagrams in the case study and can therefore simply refer to single states instead of complex set-based state configurations.

1. When `TransportingToMill` is in state `Idle`, we require that it gets a grant to put an engine into the subsequent PEN `EnginesBeforeMill` within the next 100 time units.

   ```
   // time-bounded state-oriented OCL constraint:
   context TransportingToMill inv:
     self.oclInState(TransportingToMill::Idle)
     implies
     self@post(1,100)->forAll(p:Sequence(OclState) |
                              p->includes(TransportingToMill::Requesting) )

   // CCTL formula:
   AG ( (TransportingToMill.state==TransportingToMill.idle)
       -> AF[1,100](TransportingToMill.state==TransportingToMill.requesting))
   ```

   A corresponding constraint can also be formulated to require a transition from state `Requesting` to `Performing`.

2. A performed transport – once started after the acknowledgements have been received – has to be completed within 300 time units.

   ```
   // time-bounded state-oriented OCL constraint:
   context TransportingToMill inv:
     self.oclInState(TransportingToMill::Performing)
     implies
     self@post(1,300)->forAll(p:Sequence(OclState) |
                              p->exists(s:OclState |
                                        s = TransportingToMill::Idle) )

   // CCTL formula:
   AG ( (TransportingToMill.state==TransportingToMill.performing)
       -> AF[1,300](TransportingToMill.state<>TransportingToMill.performing))
   ```

3. An acknowledgement for an available AGV within composite state `TransportingToMill::Performing` must be received within 150 time units.

   ```
   // time-bounded state-oriented OCL constraint:
   context TransportingToMill inv:
     self.oclInState(TransportingToMill::Performing::GettingAGV)
     implies
     self@post(1,150)->forAll(p:Sequence(OclState) |
   ```

```
                              p->exists(s:OclState |
                                  s <> TransportingToMill::Performing::GettingAGV))

    // CCTL formula:
    AG ( ((TransportingToMill_performing.state==
                                         TransportingToMill_performing.gettingAGV)
           & (TransportingToMill_performing.activated==true))
        -> AF[1,150]( (TransportingToMill_performing.state==
                                       TransportingToMill_performing.movingToLoad)
                    & TransportingToMill_performing.activated
                 )
       )
```

Note here that the activation of composite substate `Performing` has to be considered explicitly in the CCTL formula, as it is explained in Section 7.3.

4. Production progress is ensured by requiring that a transport to station mill can always again be performed, i.e., at each point of time, state `Performing` will eventually be entered, and at each point of time, state `Idle` will eventually be entered. (The latter condition guarantees that state `Performing` is also eventually left again.)

```
    // temporal state-oriented OCL constraint:
    context TransportingToMill inv:
      self@post()->forAll(p:Sequence(OclState) |
                          p->includes(TransportingToMill::Performing) )
      and
      self@post()->forAll(p:Sequence(OclState) |
                          p->includes(TransportingToMill::Idle) )

    // (C)CTL formula:
    AG AF (TransportingToMill.state==TransportingToMill.performing)
    &
    AG AF (TransportingToMill.state==TransportingToMill.idle)
```

State-oriented OCL Specifications over Concurrent State Diagrams. For more complex situations, such as the concurrent State Diagram shown in Figure 2.4 on page 28, we can also make use of the newly introduced operation `oclInConf()` (cf. Section 7.1.3).

We want to specify that an AGV must never be in an accepting state in the negotiation part while it is performing a transport. This can be expressed by excluding that states `WaitingForAcknowledgement` and certain substates of `Transport` are both active at the same time.

```
// state-oriented OCL constraint:
context AGV inv:
  not self.oclInConf( Set{Negotiator::WaitingForAcknowledgement,
                          Transport::MovingToLoad} )
  and
  not self.oclInConf( Set{Negotiator::WaitingForAcknowledgement,
                          Transport::MovingToLoad} )
  and
  not self.oclInConf( Set{Negotiator::WaitingForAcknowledgement,
                          Transport::MovingToLoad} )
  and
  not self.oclInConf( Set{Negotiator::WaitingForAcknowledgement,
                          Transport::MovingToLoad} )

// CCTL formula:
AG !( ( (AGV_negotiator.state==AGV_negotiator.waitingForAcknowledgement)
        &(AGV_transport.state==AGV_transport.movingToLoad) )
     |( (AGV_negotiator.state==AGV_negotiator.waitingForAcknowledgement)
        &(AGV_transport.state==AGV_transport.loading) )
     |( (AGV_negotiator.state==AGV_negotiator.waitingForAcknowledgement)
        &(AGV_transport.state==AGV_transport.movingToUnload) )
     |( (AGV_negotiator.state==AGV_negotiator.waitingForAcknowledgement)
        &(AGV_transport.state==AGV_transport.unloading) )
    )
```

To ensure production progress, we require that an AGV object is not idle for too long, e.g., after at most 400 time units it has to again load an item. Note here that it is not sufficient to specify that state `Idle` will eventually be left within 400 time units, as leaving state `Idle` may also be due to a movement to vacate a position. Thus, a corresponding OCL constraint is, e.g.,

```
context AGV inv:
  self@post(1,400)->forAll(trace:Sequence(Set(OclState)) |
                           trace->exists(conf:Set(OclState) |
                                         conf->includes(AGV::Transport::Loading)))
```

Further examples of time-bounded state-oriented OCL constraints in the context of other UML and MFERT models can be found in [FM02a, FM02d, BFG$^+$03]. Also the property specification patterns listed in Appendix D show how to express certain constraints with the state-oriented OCL extension.

Chapter 9

Conclusion

A book is never finished, it is only published.
– Derick Wood

This thesis presented a state-oriented real-time extension of the Object Constraint Language OCL and its application in the area of modeling manufacturing systems with a UML-based variant of MFERT. Some preliminaries were necessary to be able to define a formal semantics of this OCL extension. This mainly concerns an existing formal model for parts of OCL and the introduction of a notion of time to UML State Diagrams. The results of this thesis can be summarized as follows.

Extended Object Models. Currently, there is no official formal semantics of OCL in UML, but the adopted OCL 2.0 specification [OMG03b] has included and extended the set-theoretic OCL semantics developed by Richters in [Ric01]. In both documents, a metamodel for OCL is defined and a semantics is given by a formal description of Class Diagrams in form of an *object model* and a meaning function that maps OCL expressions to a semantic domain, i.e., objects and basic data values. Nevertheless, there are still deficiencies w.r.t. the integration of UML State Diagrams. Although there is a standard operation called `oclInState(s:OclState)`, no corresponding semantics has been defined yet, i.e., the formal object model lacks of a State Diagram description with states and active state configurations. We therefore first formally integrated relevant UML State Diagram concepts into OCL. We extended the formal *object model* by a notion of state configurations, such that a formal relationship of UML State Diagrams and state-oriented OCL constraints is well-defined.

Timed UML State Diagrams. OCL constraints do not make sense without a given user model to refer to. When time-bounded constraints are specified, the behavioral description of the corresponding user model must be equipped a notion of time as well. We therefore introduced a timed variant of UML State Diagrams for behavioral modeling with explicit timing assumptions of activities. We identified which of the UML State Diagram concepts can be omitted and applied further restrictions that are appropriate for the regarded domain of this thesis, i.e., modeling of manufacturing systems. A formal semantics with discrete time has been defined by a mapping of the chosen State Diagram concepts to RIL, which is the input language of the RAVEN model checker and corresponds to the formal language of I/O-Interval Structures.

Modeling of Manufacturing Systems. The chosen application domain in this thesis is the modeling of manufacturing systems. In this context, we employed the graphical MFERT notation and defined UML stereotypes for the structural elements of MFERT graphs, i.e., Production Process Nodes, Production Element Nodes, and links between them that represent production element flow. Moreover, the presented timed UML State Diagram variant is used to model the behavior of MFERT nodes. UML-based MFERT models that comply to a number of identified structural validation constraints can be mapped to RIL (or I/O-Interval Structures, respectively). This is indicated in Figure 9.1 by the mapping called M_{MFERT}. This UML Profile is the basis that enables modelers to directly apply OCL constraints to MFERT models.

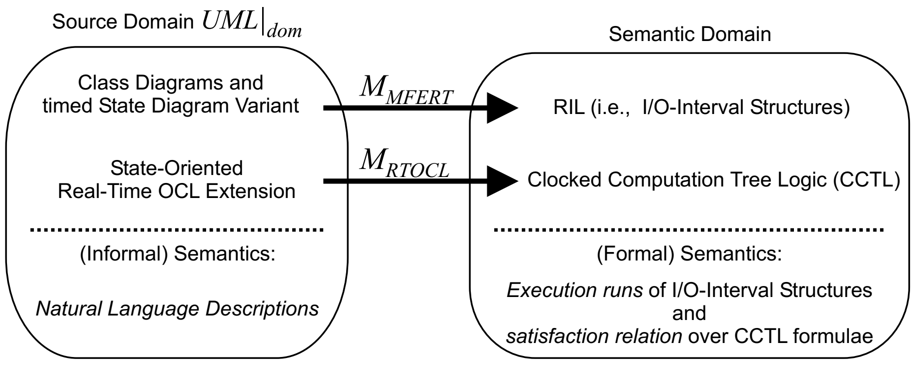

Source Domain $UML|_{dom}$
Class Diagrams and timed State Diagram Variant
State-Oriented Real-Time OCL Extension
(Informal) Semantics:
Natural Language Descriptions
M_{MFERT}
M_{RTOCL}
Semantic Domain
RIL (i.e., I/O-Interval Structures)
Clocked Computation Tree Logic (CCTL)
(Formal) Semantics:
Execution runs of I/O-Interval Structures and satisfaction relation over CCTL formulae

Figure 9.1: Semantic Domain Mapping

State-Oriented Real-Time OCL Extension. Independently, a number of extensions of OCL have already been proposed to enable modelers to specify temporal properties, e.g., concerning occurrences of events and their timing properties such as deadlines, delay times, and response times [CK02, RvTdR01]. However, state-related temporal properties have not yet been considered in the context of UML and OCL. The state-oriented real-time extension of OCL presented in this thesis enables modelers to express time-bounded properties w.r.t. progress of system execution by means of sequences of state configurations. The extension is introduced by means of a UML Profile and is compliant with common OCL syntax and concepts. We have shown that our state-oriented OCL extension has the expressive power to express the property specification patterns identified by Dwyer et al. [DAC98a].

Based on the formal definition of execution traces of timed UML State Diagrams (i.e., runs of I/O-Interval Structures), newly introduced OCL operations allow to extract possible future execution paths of our timed UML State Diagram variant. Then, already existing OCL operations for sequences and sets can be applied to access and manipulate these traces.

The mapping of temporal OCL constraints to the temporal logics CCTL establishes a formal relationship of UML-based MFERT models and state-oriented real-time OCL constraints. This is indicated by the mapping called M_{RTOCL} in Figure 9.1. A formal semantics of the combination of UML-based MFERT notation and state-oriented real-time OCL constraints is then automatically derived, as the two formal target languages, i.e., I/O-Interval Structures and CCTL, already have a well-defined formal relationship by the notion of runs and a satisfaction relation.

As a next step, the real-time model checker RAVEN can be applied to verify whether a model (given as a set of RIL modules or I/O-Interval Structures, respectively) satisfies properties specified as CCTL formulae. However, note that the mapping M_{MFERT} presented in this thesis is not designed to build an efficient model representation for the model checker. Besides the explicit times, especially the support of composite states with interlevel transitions leads to a number of additional internal variables in the target language RIL that even expands the state space. An approach to overcome this problem is mentioned in the outlook on future work below.

Formalizations of UML to perform model checking is also addressed by other authors, but most of these approaches do not consider explicit time, e.g., [LMM99a, LP99]. In [DMY02], a mapping of a dense-time UML State Diagram variant to hierarchical timed automata, i.e., the input language of the real-time model checker UPPAAL, is presented. However, this approach only considers the UPPAAL specification language for property specifications, which is a restricted form of Timed Computation Tree Logic (TCTL) dedicated to perform reachability analysis over models with dense time. As a consequence, the property specification patterns of Dwyer et al. [DAC98a] are not completely supported. This and other approaches that investigate model checking of UML designs can benefit from the state-oriented real-time OCL extension presented in this thesis, as it allows to abstract from temporal logic formulae, yet has sufficient expressive power, and builds upon already existing concepts of standard UML.

9.1 Future Work

The work presented in this thesis directly leads to issues that can be investigated in future work. We here give a list of ideas how to continue this work, without claiming that this list is complete.

- The formal semantics of OCL have to be completed. Extended object models still lack of a formalization of tuples, ordered sets and the concept of OCL messages. As a first step, the formal semantics of OCL messages has recently been published in [FM04]. That work enhances the extended object model and corresponding system states with appropriate additional components, in particular to keep track of the history of messages sent during operation execution.

- Applying the OCL extension to other timed variants of Statecharts or UML State Diagrams should be possible without too much effort. However, the semantics likely have to be adjusted in each case.

- The mapping of our timed State Diagrams variant to I/O-Interval Structures can be enhanced. For example, additional modeling elements such as history states can be included.

- The UML-based MFERT models presented in this thesis were translated by hand to I/O-Interval Structures. It should be investigated whether we can build upon the implementation in [Zab03] to support automated translations of our timed UML State Diagram variant to I/O-Interval Structures.

- To cope with the state-explosion problem in model checking, additional techniques such as decomposition or abstraction are necessary to efficiently perform model checking on models of bigger size. In this context, the approach presented in [Zab03] already applies *profiling* to find optimal orderings of BDDs to perform model checking more efficiently.

  Additional domain-specific assumptions might allow for a decomposition to be able to perform model checking on submodels. A similar approach was taken in the domain of modeling shuttle convoys as part of a railway system [BFG$^+$03].

Bibliography

[AB01] David H. Akehurst and Behzad Bordbar. On Querying UML Data Models with OCL. In Gogolla and Kobryn [GK01], pages 91–103.

[ABB$^+$00] Wolfgang Ahrendt, Thomas Baar, Bernd Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY Approach: Integrating Object Oriented Design and Formal Verification. In M. Ojeda-Aciego, I.P. de Guzmán, G. Brewka, and L.M. Pereira, editors, *8th European Workshop on Logics in AI (JELIA), Malaga, Spain, October 2000*, volume 1919 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2000.

[ACD90] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model Checking for Real-Time Systems. In *5th Annual Symposium on Logic in Computer Science, Philadelphia, PA, USA, June 1990*, pages 414–425. IEEE Computer Society Press, 1990.

[AdSSL$^+$01] Ludovic Apvrille, Pierre de Saqui-Sannes, Christophe Lohr, Patrick Sénac, and Jean-Pierre Couriat. A new UML Profile for Real-Time System Formal Design and Validation. In Gogolla and Kobryn [GK01], pages 287–301.

[Bal03] Hermann Balsters. Modelling Database Views with Derived Classes in the UML/OCL-Framework. In Stevens et al. [SWB03], pages 295–309.

[BCM$^+$90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *5th Annual Symposium on Logic in Computer Science, Philadelphia, PA, USA, June 1990*, pages 1–33. IEEE Computer Society Press, 1990.

[BCR00] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. Modeling the Dynamics of UML State Machines. In Y. Gurevich, P.W. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines, Theory and Applications (ASM 2000), Monte Verità, Switzerland, March 2000*, volume 1912 of *Lecture Notes in Computer Science*, pages 223–241. Springer, 2000.

[Bec00] Kent Beck. *Extreme Programming Explained : Embrace Change*. Addison-Wesley, 2000.

[Bee94] Michael von der Beeck. A Comparison of Statechart Variants. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Joint Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, Lübeck, Germany, September 1994*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer, 1994.

[Bee01] Michael von der Beeck. Formalization of UML-Statecharts. In Gogolla and Kobryn [GK01], pages 406–421.

[Bee02] Michael von der Beeck. A structured operational semantics for UML-Statecharts. *Software and Systems Modeling (SoSyM), Springer*, 1(2):130–141, December 2002.

[Ber89] Gérard Berry. Real Time Programming: Special Purpose or General Purpose Languages. In G. Ritter, editor, *Information Processing 89, Proceedings of the IFIP 11th World Computer Congress, San Francisco, CA, USA, August/September 1989*, pages 11–17. North-Holland/IFIP, 1989.

[BFG$^+$03] Sven Burmester, Stephan Flake, Holger Giese, Wilhelm Schäfer, and Matthias Tichy. Towards the Compositional Verification of Real-Time UML Designs. In P. Inverardi and J. Paakki, editors, *Joint 9th European Software Engineering Conference (ESEC) and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11), Helsinki, Finland, September 2003*, pages 38–47. ACM Press, 2003.

[BFMW00] Arnulf Braatz, Stephan Flake, Wolfgang Müller, and Engelbert Westkämper. Prototyping einer Fahrzeugsteuerung in virtueller 3D-Umgebung. In T. Schulze, P. Lorenz, and V. Hinz, editors, *Simulation und Visualisierung 2000, Magdeburg, Germany, March 2000*, pages 319–332. SCS Europe BVBA, Ghent, Belgium, 2000. (in German).

[BH00] Thomas Baar and Reiner Hähnle. An Integrated Metamodel for OCL Types. In R. France, B. Rumpe, J.-M. Bruel, A. Moreira, J. Whittle, and I. Ober, editors, *OOPSLA'2000 Workshop Refactoring the UML: In Search of the Core, Minneapolis, MN, USA*, 2000.

[BKPPT00] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. Consistency Checking and Visualization of OCL Constraints. In Evans et al. [EKS00], pages 294–308.

[BKPPT01] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. A Visualization of OCL Using Collaborations. In Gogolla and Kobryn [GK01], pages 257–271.

[BKS02] Julian C. Bradfield, Juliana Küster Filipe, and Perdita Stevens. Enriching OCL Using Observational Mu-Calculus. In R.-D. Kutsche and H. Weber, editors, *5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002). Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2002), Grenoble, France, April 2002*, volume 2306 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2002.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[Bur02] Sven Burmester. Generierung von Java Real-Time Code für zeitbehaftete UML Modelle. Master's thesis, University of Paderborn, Paderborn, Germany, September 2002. (in German).

[BW02] Achim D. Brucker and Burkhart Wolff. HOL-OCL: Experiences, Consequences and Design Choices. In Jézéquel et al. [JHC02], pages 196–211.

[CAB$^+$98] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[CCM97] Sérgio V.A. Campos, Edmund M. Clarke, and Marius Minea. The Verus Tool: A Quantitative Approach to the Formal Verification of Real-Time Systems. In O. Grumberg, editor, *9th International Conference on Computer Aided Verification (CAV'97), Haifa, Israel, June 1997*, volume 1254 of *Lecture Notes in Computer Science*, pages 452–455. Springer, 1997.

[CD94] Steve Cook and John Daniels. *Designing Object Systems: Object-oriented Modelling with Syntropy*. Prentice-Hall, 1994.

[CE81] Edmund M. Clarke and E. Allan Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CES86] Edmund M. Clarke, E. Allan Emerson, and Aravinda Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[CHR91] Zhou Chaochen, C.A.R. Hoare, and Anders P. Ravn. A calculus of duration. *Information Processing Letter*, 40(5):269–276, 1991.

[CK01] María V. Cengarle and Alexander Knapp. A Formal Semantics for OCL 1.4. In Gogolla and Kobryn [GK01], pages 118–133.

[CK02] María V. Cengarle and Alexander Knapp. Towards OCL/RT. In L.-H. Eriksson and P.A. Lindsay, editors, *11th International Symposium of Formal Methods Europe (FME 2002), Formal Methods: Getting IT Right, Copenhagen, Denmark, July 2002*, volume 2391 of *Lecture Notes in Computer Science*, pages 389–408. Springer, 2002.

[CKM$^+$99] Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer, and Alan Wills. The Amsterdam Manifesto on OCL. Technical Report TUM-I9925, Technische Universität München, Munich, Germany, December 1999.

[CKM$^+$02] Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer, and Alan Wills. The Amsterdam Manifesto on OCL. In Clark and Warmer [CW02], pages 115–149.

[CT00] Stefan Conrad and Klaus Turowski. Vereinheitlichung der Spezifikation von Fachkomponenten auf der Basis eines Notationsstandards. In J. Ebert and U. Frank, editors, *Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik (Beiträge des Workshops Modellierung 2000), St. Goar, Germany, April 2000*, pages 179–194. Koblenzer Schriften zur Informatik, Band 15, Fölbach-Verlag, Koblenz, Germany, 2000. (in German).

[CT01] Stefan Conrad and Klaus Turowski. Temporal OCL: Meeting Specifications Demands for Business Components. In K. Siau and T. Halpin, editors, *Unified Modeling Language: Systems Analysis, Design, and Development Issues*, pages 151–165. IDEA Group Publishing, 2001.

[CW02] Tony Clark and Jos Warmer, editors. *Object Modeling with the OCL. The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*. Springer, 2002.

[DAC98a] Matthiew B. Dwyer, George S. Avrunin, and James C. Corbett. A System of Specification Patterns, 1998. http://www.cis.ksu.edu/santos/spec-patterns (last visited on December 11th, 2003).

[DAC98b] Matthiew B. Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite-State Verification. In M. Ardis, editor, *Second ACM Workshop on Formal Methods in Software Practice, Clearwater Beach, FL, USA, March 1998*, pages 7–15. ACM Press, 1998.

[DAC99] Matthiew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *21st International Conference on Software Engineering (ICSE 99), Los Angeles, CA, USA, May 1999*, pages 411–420. ACM Press, 1999.

[DDF$^+$02] Wilhelm Dangelmaier, Carsten Darnedde, Stephan Flake, Wolfgang Müller, Ulrich Pape, and Henning Zabel. Graphische Spezifikation und Echtzeitverifikation von Produktionsautomatisierungssystemen. In *4. Paderborner Frühlingstagung, April 2002*, ALB-HNI-Verlagsschriftenreihe, Paderborn, Germany, 2002. (in German).

[Dee03] S.M. Deen, editor. *Agent-Based Manufacturing. Advances in the Holonic Approach*. Springer, 2003.

[DJHP98] Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli. A Compositional Real-Time Semantics of STATEMATE Designs. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference, International Symposium (COMPOS'97), Malente, Germany, September 1997*, volume 1536 of *Lecture Notes in Computer Science*, pages 186–238. Springer, 1998.

[DK01] Werner Damm and Jochen Klose. Verification of a radio-based signaling system using the STATEMATE verification environment. *Formal Methods in System Design*, 19(2):121–141, 2001.

[DKM$^+$94] Laura K. Dillon, George Kutty, Louise E. Moser, P. Michael Melliar-Smith, and Y.S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, 3(2):131–165, 1994.

[DKR00] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. On a Temporal Logic for Object-Based Systems. In S.F. Smith and C.L. Talcott, editors, *IFIP TC6/WG6.1 Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000), Stanford, CA, USA, September 2000*, pages 305–326. Kluwer Academic Publishers, 2000.

[DLM02] Vieri Del Bianco, Luigi Lavazza, and Marco Mauri. A Formalization of UML Statecharts for Real-Time Software Modeling. In H. Ehrig, B.J. Krämer, and A. Ertas, editors, *6th Biennial World Conference on Integrated Design Process Technology (IDPT 2002), Session "Towards a rigorous UML", Pasadena, CA, USA, June 2002*. Society for Design and Process Science, 2002.

[DM01] Alexandre David and M. Oliver Möller. From HUPPAAL to UPPAAL. A Translation from Hierarchical Timed Automata to Flat Timed Automata. Technical Report RS-01-11, BRICS, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2001.

[DMY02] Alexandre David, M. Oliver Möller, and Wang Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In R.-D. Kutsche and H. Weber, editors, *5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002). Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2002), Grenoble, France, April 2002*, volume 2306 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2002.

[Dou00] Bruce P. Douglass. *Doing Hard Time: Developing Real Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 2000.

[DW93] Wilhelm Dangelmaier and Harald Wiedenmann. *Modell der Fertigungssteuerung*. Beuth Verlag GmbH, Berlin, Wien, Zürich, 1st edition, 1993.

[DW97] Wilhelm Dangelmaier and Hans-Jürgen Warnecke. *Fertigungslenkung: Planung und Steuerung des Ablaufs der diskreten Fertigung*. Springer, 1997.

[EC80] E. Allan Emerson and Edmund M. Clarke. Characterizing Correctness Properties of Parallel Programs using Fixpoints. In J.W. de Bakker and J. van Leeuwen, editors, *Automata, Languages, and Programming. 7th Colloquium. Noordweijkerhout, The Netherlands, July 1980*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1980.

[EE94] Jürgen Ebert and Gregor Engels. Observable or Invocable Behaviour: You have to Choose. Technical report, Universität Koblenz, Koblenz, Germany, 1994.

[EKS00] Andy Evans, Stuart Kent, and Bran Selic, editors. *UML 2000 – The Unified Modeling Language. Advancing the Standard. Third International Conference. York, UK, October 2000*, volume 1939 of *Lecture Notes in Computer Science*. Springer, 2000.

[EMSS92] E. Allan Emerson, Aloysius K. Mok, Aravinda Prasad Sistla, and Jai Srinivasan. Quantitative temporal reasoning. *Journal of Real-Time Systems*, 4(4):331–352, 1992.

[EN00] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 3rd edition, 2000.

[EW00] Rik Eshuis and Roel Wieringa. Requirements-Level Semantics for UML Statecharts. In S.F. Smith and C.L. Talcott, editors, *IFIP TC6/WG6.1 Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000), Stanford, CA, USA, September 2000*, pages 121–140. Kluwer Academic Publishers, 2000.

[EW01] Rik Eshuis and Roel Wieringa. A Real-Time Execution Semantics for UML Activity Diagrams. In H. Hußmann, editor, *4th International Conference on Fundamental Approaches to Software Engineering (FASE 2001). Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2001), April 2001, Genova, Italy*, volume 2029 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2001.

[FGK96] Jürgen Froessl, Joachim Gerlach, and Thomas Kropf. An Efficient Algorithm for Real-Time Symbolic Model Checking. In *European Design and Test Conference and Exhibition (EDTC'96), Paris, France, March 1996*, pages 15–21. IEEE Computer Society Press, 1996.

[FGM$^+$01] Stephan Flake, Christian Geiger, Wolfgang Müller, Volker Paelke, Waldemar Rosenbach, and Jürgen Ruf. Customer-Oriented Systems Design through Virtual Prototypes. In *10th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'01), Cambridge, MA, USA, June 2001*, pages 263–268. IEEE Computer Society Press, 2001.

[FHD$^+$99] Thomas Firley, Michaela Huhn, Karsten Diethers, Thomas Gehrke, and Ursula Goltz. Timed Sequence Diagrams and Tool-Based Analysis – A Case Study. In France and Rumpe [FR99], pages 645–660.

[Fla03a] Stephan Flake. Modeling and Verification of Manufacturing Systems: A Domain-Specific Formalization of UML. In M.H. Hamza, editor, *7th IASTED International Conference on Software Engineering and Applications (SEA 2003), Los Angeles, CA, USA, November 2003*, pages 580–586. ACTA Press, Calgary, Canada, 2003.

[Fla03b] Stephan Flake. Temporal OCL Extensions for Specification of Real-Time Constraints. In S. Graf, O. Haugen, I. Ober, and B. Selic, editors, *UML 2003 Workshop "Specification and Validation of UML models for Real Time and Embedded Systems" (SVERTS'03), San Francisco, CA, USA, October 2003*, 2003. http://www-verimag.imag.fr/EVENTS/2003/SVERTS/PAPERS-WEB/12-Flake-temporalOclExtensions.pdf (last visited on December 11th, 2003).

[Fla04] Stephan Flake. OclType – A Type or Metatype? In T. Baar, T. Clark, R. France, R. Hähnle, H. Hußmann, and P.H. Schmitt, editors, *UML 2003 Workshop "OCL 2.0 – Industry Standard or Scientific Playground?", San Francisco, CA, USA, October 2003*, Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam, The Netherlands, 2004.

[FM01] Stephan Flake and Wolfgang Müller. Schnittstellendefinition zur 3D-Animation eines holonischen Fertigungssystems. Technical Report 09/2001, C-LAB, Paderborn, Germany, August 2001. (in German).

[FM02a] Stephan Flake and Wolfgang Müller. A UML Profile for MFERT. Technical Report 04/2002, C-LAB, Paderborn, Germany, March 2002.

[FM02b] Stephan Flake and Wolfgang Müller. A UML Profile for Real-Time Constraints with the OCL. In Jézéquel et al. [JHC02], pages 179–195.

[FM02c] Stephan Flake and Wolfgang Müller. An OCL Extension for Real-Time Constraints. In Clark and Warmer [CW02], pages 150–171.

[FM02d] Stephan Flake and Wolfgang Müller. Specification of Real-Time Properties for UML Models. In R.H. Sprague, Jr., editor, *35th Hawaii International Conference on System Sciences (HICSS-35), Big Island, HI, USA, January 2002*. IEEE Computer Society Press, 2002.

[FM02e] Stephan Flake and Wolfgang Müller. Temporale Erweiterungen der OCL – Überblick und Aussichten. In *2. Workshop "Ablaufmodellierung in ingenieurwissenschaftlichen Anwendungen", Halle(Saale), Germany*, April 2002. (in German).

[FM03a] Stephan Flake and Wolfgang Müller. Expressing Property Specification Patterns with OCL. In *The 2003 International Conference on Software Engineering Research and Practice (SERP'03), Las Vegas, NV, USA, June 2003*, pages 595–601. CSREA Press, Las Vegas, NV, USA, 2003.

[FM03b] Stephan Flake and Wolfgang Müller. Formal semantics of static and temporal state-oriented OCL constraints. *Software and Systems Modeling (SoSyM), Springer*, 2(3):164–186, October 2003.

[FM03c] Stephan Flake and Wolfgang Müller. Semantics of State-Oriented Expressions in the Object Constraint Language. In *15th International Conference on Software Engineering and Knowledge Engineering (SEKE 2003), San Francisco Bay, CA, USA, July 2003*, pages 142–149. Knowledge Systems Institute, Skokie, IL, USA, 2003.

[FM04] Stephan Flake and Wolfgang Müller. Formal Semantics of OCL Messages. In *UML 2003 Workshop "OCL 2.0 – Industry Standard or Scientific Playground?", San Francisco, CA, USA, October 2003*, Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam, The Netherlands, 2004.

[FMPR00] Stephan Flake, Wolfgang Müller, Ulrich Pape, and Jürgen Ruf. Modellprüfung für den Entwurf von Fertigungssteuerungssystemen. In H. Schmidt, editor, *Modellierung betrieblicher Informationssysteme, Proceedings der MobIS-Fachtagung 2000, Siegen, Germany, October 2000*, Rundbrief der GI-Fachgruppe 5.10, 7. Jahrgang, Heft 1, pages 251–262, 2000. (in German).

[FMPR01] Stephan Flake, Wolfgang Müller, U. Pape, and Jürgen Ruf. Analyzing Timing Constraints in Flexible Manufacturing Systems. In *International NAISO Symposium on Information Science Innovations in Intelligent Automated Manufacturing (IAM'2001), Dubai, United Arab Emirates*, pages 1036–1042. ICSC Academic Press, March 2001.

[FMR00] Stephan Flake, Wolfgang Müller, and Jürgen Ruf. Structured English for Model Checking Specification. In K. Waldschmidt and C. Grimm, editors, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, Frankfurt/M., Germany, February 2000*, pages 251–262. VDE Verlag, Berlin, Germany, 2000.

[FR99] Robert France and Bernhard Rumpe, editors. *UML'99 – The Unified Modeling Language. Beyond the Standard. Fort Collins, CO, USA*, volume 1723 of *Lecture Notes in Computer Science*. Springer, 1999.

[FS99] Martin Fowler and Kendall Scott. *UML Distilled : A Brief Guide to the Standard Object Modeling Language*. Object Technology Series. Addison-Wesley, 1999.

[Gaj97] Daniel D. Gajski. *Principles of Digital Design*. Prentice Hall, 1997.

[GHK99] Joseph Gil, John Howse, and Stuart Kent. Constraint Diagrams: A Step Beyond UML. In *Technology of Object-Oriented Languages and Systems. Delivering Quality Software (TOOLS USA'99), Santa Barbara, CA, USA, August 1999*, pages 453–463. IEEE Computer Society Press, 1999.

[GK01] Martin Gogolla and Chris Kobryn, editors. *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference. Toronto, Canada. October 2001*, volume 2185 of *Lecture Notes in Computer Science*. Springer, 2001.

[GKC99] Dimitra Giannakopoulou, Jeff Kramer, and Shing-Chi Cheung. Behaviour analysis of distributed systems using the Tracta approach. *Journal of Automated Software Engineering, special issue on Automated Analysis of Software*, 6(1):7–35, January 1999.

[GR99] Martin Gogolla and Mark Richters. Transformation Rules for UML Class Diagrams. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 – Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 1999.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HK03] Martin Hitz and Gerti Kappel. *UML@Work: Von der Analyse zur Realisierung*. dpunkt-Verlag, Heidelberg, Germany, 2nd edition, 2003. (in German).

[HN96] David Harel and Amnon Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):292–333, 1996.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hol99] Ralf Holtkamp. *Ein Rahmenwerk für die Fertigungslenkung*. PhD thesis, Heinz Nixdorf Institute, HNI-Verlagsschriftenreihe, Band 51, Paderborn, Germany, 1999. (in German).

[HPSS87] David Harel, Amir Pnueli, Jeanette P. Schmidt, and Rivi Sherman. On the Formal Semantics of Statecharts. In *Second IEEE Symposium on Logic in Computer Science, Ithaca, NY, USA, June 1987*, pages 54–64. IEEE Computer Society Press, 1987.

[HR00] Michael R.A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.

[IEE87] IEEE, The Institute of Electrical and Electronics Engineers. Software Engineering Standards, 1987.

[IHJ$^+$03] Anders Ivner, Jonas Högström, Simon Johnston, David Knox, and Pete Rivett. Response to the UML2.0 OCL RfP, Version 1.6 (Submitters: Boldsoft, Rational, IONA, Adaptive Ltd., et al.). OMG Document ad/03-01-07, January 2003. ftp://ftp.omg.org/pub/docs/ad/03-01-07.pdf (last visited on December 11th, 2003).

[ISO96] ISO International Standards Organization. Information Technology – Programming Languages, their Environments and System Software Interfaces – Vienna Development Method – Specification Language – Part 1: Base language. International Standard ISO/IEC 13817-1, December 1996.

[ISO02] ISO International Standards Organization. Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics. International Standard ISO/IEC 13568, July 2002.

[JEJ02] Yan Jin, Robert Esser, and Jörn W. Janneck. Describing the Syntax and Semantics of UML Statecharts in a Heterogeneous Modelling Environment. In M. Hegarty, B. Meyer, and N.H. Narayanan, editors, *Diagrams 2002 – Second International Conference on Theory and Application of Diagrams, April 2002, Callaway Gardens, GA, USA*, volume 2317 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2002.

[Jen91] Kurt Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis. In K. Jensen and G. Rozenberg, editors, *High-level Petri Nets, Theory and Application*, pages 44–119. Springer, 1991.

[JHC02] Jean-Marc Jézéquel, Heinrich Hußmann, and Stephen Cook, editors. *UML 2002 – The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference. Dresden, Germany, September/October 2002*, volume 2460 of *Lecture Notes in Computer Science*. Springer, 2002.

[JMM$^+$99] Wil Janssen, Radu Mateescu, Sjouke Mauw, Peter Fennema, and Petra van der Stappen. Model Checking for Managers. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 1999, and Toulouse, France, September 1999*, volume 1680 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 1999.

[JRB99] Ivar Jacobson, James Rumbaugh, and Grady Booch. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, 1999.

[KH02] Stuart Kent and John Howse. Constraint Trees. In Clark and Warmer [CW02], pages 228–249.

[KMR02] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model Checking Timed UML State Machines and Collaborations. In W. Damm and E.-R. Olderog, editors, *7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002), Oldenburg, September 2002*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–416. Springer, 2002.

[Koe67] Arthur Koestler. *The Ghost in the Machine*. PAN Books, London, UK, 1967.

[KP92] Yonit Kesten and Amir Pnueli. Timed and Hybrid Statecharts and their Textual Representation. In J. Vytopil, editor, *Second International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 1992), Nijmegen, The Netherlands, January 1992*, volume 571 of *Lecture Notes in Computer Science*, pages 591–619. Springer, 1992.

[KTW02] Christiane Kiesner, Gabriele Taentzer, and Jessica Winkelmann. VisualOCL: A Visual Notation of the Object Constraint Language. Technical Report 23, Computer Science Department of the Technical University of Berlin, Berlin, Germany, 2002.

[Kus01] Sabine Kuske. A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In Gogolla and Kobryn [GK01], pages 241–256.

[KW00] Anneke Kleppe and Jos Warmer. Extending OCL to Include Actions. In Evans et al. [EKS00], pages 440–450.

[KW01] Anneke Kleppe and Jos Warmer. Unification of Static and Dynamic Semantics of UML: a Study in Redefining the Semantics of the UML Using the pUML OO Meta Modelling Approach, 2001. http://www.klasse.nl/english/uml/uml-semantics.html (last visited on December 11th, 2003).

[KW02] Anneke Kleppe and Jos Warmer. The Semantics of the OCL Action Clause. In Clark and Warmer [CW02], pages 213–227.

[Kwo00] Gihwon Kwon. Rewrite Rules and Operational Semantics for Model Checking UML Statecharts. In Evans et al. [EKS00], pages 528–540.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam00] Axel van Lamsweerde. Formal Specification: a Roadmap. In A. Finkelstein, editor, *22nd International Conference on Software Engineering (ICSE 2000), Future of Software Engineering Track, June 2000, Limerick, Ireland*, pages 147–159. ACM Press, 2000.

[LC96] Karl R.P.H. Leung and Daniel K.C. Chan. Extending Statecharts with Duration. In *20th Annual International Computer Software and Application Conference (COMPSAC'96), Seoul, South Korea, August 1996*, pages 246–251. IEEE Computer Society Press, 1996.

[Lev97] Francesca Levi. *Verification of Temporal and Real-Time Properties of Statecharts*. PhD thesis, Dipartimento di Informatica, Universita di Pisa, Pisa, Italy, 1997.

[LMM99a] Diego Latella, István Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML Statechart Diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[LMM99b] Diego Latella, István Majzik, and Mieke Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99), Florence, Italy, February 1999*, pages 331–347. Kluwer Academic Publishers, 1999.

[LP99] Johan Lilius and Ivan Paltor. Formalising UML State Machines for Model Checking. In France and Rumpe [FR99], pages 430–445.

[LQV01] Luigi Lavazza, Gabriele Quaroni, and Matteo Venturelli. Combining UML and Formal Notations for Modelling Real-Time Systems. In V. Gruhn, editor, *Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9), Vienna, Austria, September 2001*, pages 196–206. ACM Press, 2001.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[MC99] Luis Mandel and María V. Cengarle. On the Expressive Power of OCL. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – Formal Methods. World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999*, volume 1708 of *Lecture Notes in Computer Science*, pages 854–874. Springer, 1999.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall International Editions, 2nd edition, 1997.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[MP90] Zohar Manna and Amir Pnueli. A Hierarchy of Temporal Properties. In *9th Annual ACM Symposium on Principles of Distributed Computing, Quebec City, Quebec, Canada, August 1990*, pages 377–410. ACM Press, 1990.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Specification*. Springer, 1992.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems. Safety*. Springer, 1995.

[MSP96] Andrea Maggiolo-Schettini and Adriano Peron. Retiming Techniques for Statecharts. In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'96), 4th International Symposium, Uppsala, Sweden, September 1996*, volume 1135 of *Lecture Notes in Computer Science*, pages 55–71. Springer, 1996.

[Mül96] Wolfgang Müller. *Executable Graphics for VHDL-Based Systems Design*. PhD thesis, Department of Mathematics and Computer Science, Universität-GH Paderborn, Paderborn, Germany, 1996.

[OMG] OMG, Object Management Group. http://www.omg.org.

[OMG99] OMG Analysis and Design Platform Task Force. White Paper on the Profile Mechanism, Version 1.0. OMG Document ad/99-04-07, April 1999. ftp://ftp.omg.org/pub/docs/ad/99-04-07.pdf (last visited on December 11th, 2003).

[OMG00a] OMG, Object Management Group. UML 2.0 OCL Request For Proposal. OMG Document ad/00-09-03, September 2000. ftp://ftp.omg.org/pub/docs/ad/00-09-03.pdf (last visited on December 11th, 2003).

[OMG00b] OMG, Object Management Group. UML Profile for CORBA Specification. OMG Document ptc/00-10-01, October 2000. ftp://ftp.omg.org/pub/docs/ptc/00-10-01.pdf (last visited on December 11th, 2003).

[OMG01] OMG, Object Management Group. Common Warehouse Metamodel (CWM) Specification. OMG Documents formal/01-10-01 (main specification) and formal/01-10-27 (extensions), October 2001. ftp://ftp.omg.org/pub/docs/formal/01-10-01.pdf (last visited on December 11th, 2003).

[OMG02] OMG, Object Management Group. Meta Object Facility Specification. OMG Doucument formal/02-04-03, April 2002. ftp://ftp.omg.org/pub/docs/formal/02-04-03.pdf (last visited on December 11th, 2003).

[OMG03a] OMG Analysis and Design Platform Task Force. UML 2.0 OCL RFP – Recommendation Vote Status, May 2003. http://www.omg.org/techprocess/meetings/schedule/UML_2.0_OCL_RFP.html (last visited on December 11th, 2003).

[OMG03b] OMG, Object Management Group. UML 2.0 OCL Final Adopted Specification. OMG Document ptc/03-10-14, October 2003. ftp://ftp.omg.org/pub/docs/ptc/03-10-14.pdf (last visited on December 11th, 2003).

[OMG03c] OMG, Object Management Group. UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/03-03-02, April 2003. ftp://ftp.omg.org/pub/docs/ptc/03-03-02.pdf (last visited on December 11th, 2003).

[OMG03d] OMG, Object Management Group. Unified Modeling Language 1.5 Specification. OMG Document formal/03-03-01, March 2003. ftp://ftp.omg.org/pub/docs/formal/03-03-01.pdf (last visited on December 11th, 2003).

[OMG03e] OMG, Object Management Group. Unified Modeling Language: Infrastructure, Version 2.0. Adopted Specification, OMG Document ad/03-03-01, July 2003. ftp://ftp.omg.org/pub/docs/ad/03-03-01.pdf (last visited on December 11th, 2003).

[OMG03f] OMG, Object Management Group. Unified Modeling Language: Superstructure, Version 2.0. Final Adopted Specification, OMG Document ptc/03-08-02, August 2003. ftp://ftp.omg.org/pub/docs/ptc/-03-08-02.pdf (last visited on December 11th, 2003).

[Pad00] Peter Padawitz. Swinging UML – How to Make Class Diagrams and State Machines Amenable to Constraint Solving and Proving. In Evans et al. [EKS00], pages 162–177.

[Par95] David L. Parnas. Teaching Programming as Engineering. In J.P. Bowen and M.G. Hinchey, editors, *The Z Formal Specification Notation, 9th International Conference of Z Users (ZUM'95), Limerick, Ireland, September 1995*, volume 967 of *Lecture Notes in Computer Science*, pages 471–481. Springer, 1995.

[Pnu80] Amir Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1980.

[PS91] Amir Pnueli and Michal Shalev. What is in a Step: On the Semantics of Statecharts. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 244–264. Springer, 1991.

[PS97] Jan Philipps and Peter Scholz. Compositional Specification of Embedded Systems with Statecharts. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development. 7th International Joint Conference CAAP/FASE, Lille, France, April 1997*, volume 1214 of *Lecture Notes in Computer Science*, pages 637–651. Springer, 1997.

[PU97] Carsta Petersohn and Luis Urbina. A Timed Semantics for the STATEMATE Implementation of Statecharts. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *4th International Symposium of Formal Methods Europe (FME'97): Industrial Applications and Strengthened Foundations of Formal Methods, Graz, Austria, September 1997*, volume 1313 of *Lecture Notes in Computer Science*, pages 553–572. Springer, 1997.

[Qui01] Julia Quintanilla de Simsek. *Ein Verifikationsansatz für eine netzbasierte Modellierungsmethode für Fertigungssysteme*. PhD thesis, Heinz Nixdorf Institute, HNI-Verlagsschriftenreihe, Band 87, Paderborn, Germany, 2001. (in German).

[RACH00] Gianna Reggio, Egidio Astesiano, Christine Choppy, and Heinrich Hußmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In T. Maibaum, editor, *Third International Conference on Fundamental Approaches to Software Engineering (FASE 2000). Part of the European Joint Conferences on the Theory and Practice of Software (ETAPS 2000), Berlin, Germany, March 2000*, volume 1783 of *Lecture Notes in Computer Science*. Springer, 2000.

[RBP$^+$91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International Editions, 1991.

[RG98] Mark Richters and Martin Gogolla. On Formalizing the UML Object Constraint Language OCL. In T.W. Ling, S. Ram, and M.L. Lee, editors, *17th International Conference on Conceptual Modeling (ER'98), Singapore, November 1998*, volume 1507 of *Lecture Notes in Computer Science*, pages 449–464. Springer, 1998.

[RG99] Mark Richters and Martin Gogolla. A Metamodel for OCL. In France and Rumpe [FR99], pages 156–171.

[Ric01] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Bremen, Germany, 2001.

[RJB98] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[RK97] Jürgen Ruf and Thomas Kropf. Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals. In E. Cerny and D.K. Probst, editors, *Correct Hardware Design and Verification Methods (CHARME'97), 9th IFIP WG 10.5 Advanced Research Working Conference, Montreal, Canada, October 1997*, pages 146–166. Chapman and Hall, 1997.

[RK99] Jürgen Ruf and Thomas Kropf. Modeling and Checking Networks of Communicating Real-Time Systems. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME'99), 10th IFIP WG 10.5 Advanced Research Working Conference, Bad Herrenalb, Germany, September 1999*, pages 265–279. Springer, 1999.

[RM99] Sita Ramakrishnan and John McGregor. Extending OCL to Support Temporal Operators. In *21st International Conference on Software Engineering (ICSE 99), Workshop on Testing Distributed Component-Based Systems, Los Angeles, CA, USA*, May 1999.

[RM00] Sita Ramakrishnan and John McGregor. Modelling and Testing OO Distributed Systems with Temporal Logic Formalisms. In M.H. Hamza, editor, *18th IASTED International Conference on Applied Informatics (AI'2000), Innsbruck, Austria, February 2000*. ACTA Press, Calgary, Canada, 2000.

[RS01] Bernhard Rumpe and Robert Sandner. UML – Unified Modeling Language im Einsatz. Teil 3. UML-RT für echtzeitkritische und eingebettete Systeme. *at – Automatisierungstechnik, Reihe Theorie für den Anwender, 11/2001*, 2001. (in German).

[RT01] Ella E. Roubtsova and W.J. Toetenel. Specification of Real-Time Properties in UML. In *22nd IEEE Real-Time Systems Symposium (RTSS), Work-In-Progress Section, London, UK*, December 2001.

[Ruf00] Jürgen Ruf. *Techniken zur Modellierung und Verifikation von Echtzeitsystemen*. PhD thesis, Universität Karlsruhe, Karlsruhe, Germany, March 2000. (in German).

[Ruf01] Jürgen Ruf. RAVEN: Real-Time Analyzing and Verification Environment. *Journal on Universal Computer Science (J.UCS), Springer*, 7(1):89–104, February 2001.

[Ruf02] Jürgen Ruf. Formal Verification of Timing Properties of a Holonic Material Transport System. Technical Report WSI-2002-03, Wilhelm-Schickard Institute, University of Tübingen, Tübingen, Germany, 2002.

[RvTdR01] Ella E. Roubtsova, Jan van Katwijk, W.J. Toetenel, and Ruud C.M. de Rooij. Real-Time Systems: Specification of Properties in UML. In *7th Annual Conference of the Advanced School for Computing and Imaging (ASCI 2001), Het Heijderbos, Heijen, The Netherlands, May/June 2001*, pages 188–195, 2001.

[Sch96] Uta Schneider. *Ein formales Modell und eine Klassifikation für die Fertigungssteuerung – Ein Beitrag zur Systematisierung der Fertigungssteuerung*. PhD thesis, Heinz Nixdorf Institute, HNI-Verlagsschriftenreihe, Band 16, Paderborn, Germany, 1996. (in German).

[Sel99] Bran Selic. Turning clockwise: Using UML in the real-time domain. *Communications of the ACM*, 42(10):46–54, October 1999.

[SF99] Neelam Soundarajan and Stephen Fridella. Modeling Exceptional Behavior. In France and Rumpe [FR99], pages 691–705.

[Sim00] Anthony J.H. Simons. On the Compositional Properties of UML Statechart Diagrams. In *Electronic Workshops in Computing: Rigorous Object-Oriented Methods 2000*. British Computer Society, 2000.

[SKM01] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model Checking UML State Machines and Collaborations. In S.D. Stoller and W. Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier, Amsterdam, The Netherlands, 2001.

[SR98] Bran Selic and James Rumbaugh. Using UML for Modeling Complex Real-Time Systems. White Paper, 1998. http://www.rational.com/media/whitepapers/umlrt.pdf (last visited on December 11th, 2003).

[SS00] Markus Stumptner and Michael Schrefl. Behavior Consistent Inheritance in UML. In A.H.F. Laender et al., editors, *19th International Conference on Conceptual Modeling (ER 2000), Salt Lake City, UT, USA, October 2000*, volume 1920 of *Lecture Notes in Computer Science*, pages 527–542. Springer, 2000.

[SS01] Shane Sendall and Alfred Strohmeier. Specifying Concurrent System Behavior and Timing Constraints Using OCL and UML. In Gogolla and Kobryn [GK01], pages 391–405.

[SS02a] Michael Schrefl and Markus Stumptner. Behavior consistent specialization of object life cycles. *ACM Transactions of Software Engineering and Methodology (ACM TOSEM)*, 11(1):92–148, January 2002.

[SS02b] Shane Sendall and Alfred Strohmeier. Using OCL and UML to Specify System Behavior. In Clark and Warmer [CW02], pages 250–279.

[SWB03] Perdita Stevens, Jon Whittle, and Grady Booch, editors. *UML 2003 – The Unified Modeling Language. Modeling Languages and Applications. 6th International Conference. San Francisco, CA, USA. October 2003*, volume 2863 of *Lecture Notes in Computer Science*. Springer, 2003.

[War97] Jos Warmer. OCL Parser, Version 0.3, 1997. http://www-4.ibm.com/software/ad/library/standards/-ocl-download.html (last visited on December 11th, 2003).

[WHS94] Engelbert Westkämper, Michael Höpf, and Christoph Schaeffer. Holonic Manufacturing Systems (HMS) – Test Case 5. In *Proceedings of Holonic Manufacturing Systems, Lake Tahoe, CA, USA*, February 1994.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[Wit99] Gunnar Wittich. *Ein problemorientierter Ansatz zum Nachweis von Realzeiteigenschaften eingebetteter Systeme*. PhD thesis, Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany, 1999. (in German).

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language – Getting Your Models Ready for MDA*. Object Technology Series. Addison-Wesley, 2nd edition, 2003.

[Zab03] Henning Zabel. Verfahren zur Codegenerierung für eine laufzeitoptimierte Analyse von Fertigungsplanungsmodellen durch Modelchecking. Master's thesis, Universität Paderborn, Paderborn, Germany, September 2003. (in German).

[ZG02] Paul Ziemann and Martin Gogolla. An Extension of OCL with Temporal Logic. In J. Jürjens, M.V. Cengarle, E.B. Fernandez, B. Rumpe, and R. Sandner, editors, *Critical Systems Development with UML – Proceedings of the UML'02 Workshop*, pages 53–62. Technische Universität München, Institut für Informatik, Munich, Germany, 2002.

[ZG03] Paul Ziemann and Martin Gogolla. An OCL Extension for Formulating Temporal Constraints. Technical Report 1/03, Fachbereich Mathematik und Informatik, University of Bremen, Bremen, Germany, July 2003.

Appendix A

Timed Finite State Machines for PPNs

A timed finite state machine fsm in the context of our formal MFERT definition is a tuple

$$\langle P, S, Tr, S_{Lab}, GE, AE, Tr_{Lab}, s_0 \rangle$$

where

- P is a set of atomic propositions.
- S is a set of states.
- $Tr \subseteq S \times S$ is a state transition relation, such that every state has a successor state: $\forall s \in S : \exists d \in S : (s,d) \in Tr$.
- $S_{Lab} : S \to \mathcal{P}(P)$ is a state labeling function.
- GE is a set of guard expressions. We here assume that a language L_{GE} exists with well-defined syntax and semantics for the elements of GE and that for all $g \in GE : Type(g) = Bool$.
- AE is a set of action expressions. We here assume that a language L_{AE} exists with a well-defined syntax and semantics for the elements of AE.
- $Tr_{Lab} : Tr \to \mathcal{P}(GE) \times \mathcal{P}(\mathbb{N}) \times \mathcal{P}(AE)$ is a transition labeling function that defines a set of condition expressions, a set of delay times, and a set of action expressions for each transition $tr \in Tr$.
- $s_0 \in S$ is the initial state of the finite state machine.

A.1 Help Functions

For technical reasons, we additionally define partial mappings of the transition labeling function Tr_{Lab} by the help functions listed in Table A.1.[1] Annotations of the form $Tr_{Lab}|_{(+,-,-)}$ are projections on tuple elements. A tuple element is taken iff a '$+$' is specified at its corresponding index position.

[1] $Cons_{AE}$ and $Prod_{AE}$ are defined in Subsection "Consumption and Production Actions" below.

Table A.1: Help Functions for Transitions $t \in Tr$

$Conditions(t)$	$\overset{def}{=} Tr_{Lab}(t)\|_{(+,-,-)} \subseteq \mathcal{P}(GE)$
$Delay(t)$	$\overset{def}{=} Tr_{Lab}(t)\|_{(-,+,-)} \subseteq \mathbb{N}$
$Actions(t)$	$\overset{def}{=} Tr_{Lab}(t)\|_{(-,-,+)} \subseteq \mathcal{P}(AE)$
$NextState(t)$	$\overset{def}{=} t\|_{(-,+)} \in S$
$ConsumeActions(t)$	$\overset{def}{=} \{ae \in AE \mid ae \in Actions(t) \cap Cons_{AE}\}$
$ProduceActions(t)$	$\overset{def}{=} \{ae \in AE \mid ae \in Actions(t) \cap Prod_{AE}\}$
$Count : AE \rightarrow \mathbb{N}_0$	$\forall ae \in AE : Count(ae)$ is the number of production elements that are consumed or produced when executing ae.

A.2 Operational Semantics

The operational semantics of finite state machines and their timed variants is usually defined by runs, i.e., sequences of states over time, based on some given rules when transitions are applicable and may fire. Sequence elements of these runs are usually specified with respect to elapsed time and condition evaluation. But note that *we do not provide a particular execution semantics for timed FSMs* here and simply assume that such a well-defined operational semantics exists for the FSM defined above. We just assume that the following properties hold:

1. L_{GE} defines, among others, guard expressions that query the current status of PENs without side effects.

2. L_{AE} defines expressions to reserve production elements in preceding PENs for later consumption as well as expressions to reserve sufficient space for production elements to be placed in succeeding PENs.

3. Evaluation of guard expressions may only affect local variables or adjacent PENs.

4. Execution of action expressions may only manipulate local variables or adjacent PENs.

5. Reserving production elements for consumption in a preceding PEN (or space for production in a succeeding PEN, respectively) by evaluation of a guard expression and the actual consumption (production) by means of execution of an action expression must not be interfered, i.e., once a reservation is acknowledged by a PEN, the requesting PPN immediately has to execute an action that consumes (produces) the corresponding production elements.

A.3 Consumption and Production Actions

We are particularly interested in production element flow and often need to talk about manipulations of PPNs with respect to their adjacent PENs, i.e., consumption and production of elements in PENs. We define the following sets:

- Let $Cons_{AE} \subseteq AE$ be the set of action expressions that represent consumption of production elements, i.e., actions that eliminate production elements from preceding PENs.
- Let $Prod_{AE} \subseteq AE$ be the set of action expressions that represent production of new elements, i.e., actions that add production elements to succeeding PENs.

A.4 Restrictions

Note that the following restrictions on variable types must hold for guard and action expressions.

- Free variables of guard expressions must be of a type that is defined in an adjacent PEN.

  $Type(Var(g)) \subseteq \{C(pe) \in DT \mid \exists pe \in PE : (pe, pp) \in E \vee (pp, pe) \in E, with\ pp = {M_{FSM}}^{-1}(fsm)\}$.
- Data Types of consumption actions must be defined in preceding PENs.

  $\forall ca \in Cons_{AE} : Type(ca) = void \wedge Type(Var(ca)) \subseteq \{C(pe) \mid \exists pe \in PE : (pe, pp) \in E, pp = {M_{FSM}}^{-1}(fsm)\}$.
- Data Types of production actions must be defined in succeeding PENs.

  $\forall pa \in Prod_{AE} : Type(pa) = void \wedge Type(Var(pa)) \subseteq \{C(pe) \mid \exists pe \in PE : (pp, pe) \in E, pp = {M_{FSM}}^{-1}(fsm)\}$.
- We further restrict the actions that may appear in transitions $t \in Tr$. We do not allow both consumption and production actions in the same transition.

  $\forall t \in Tr : \neg(ConsumeActions(t) \neq \varnothing\ and\ ProduceActions(t) \neq \varnothing)$

A.5 Mapping to I/O-Interval Structures

A mapping of timed FSMs to I/O-Interval Structures can be easily performed for the components P, S, Tr, Tr_{Lab}, S_{Lab}, and s_0. As we only make some basic assumptions about the languages L_{GE} and L_{AE} for guard and action expressions, a mapping of the components GE and AE cannot be given here. Instead, we make use of the mapping of Timed State Diagrams (see Section 2.4.3) to I/O-Interval Structures, in particular for transition annotations (events, guards, actions, and timing information) and synchronous event communication.

Appendix B

OCL Metalevel Operations for Classifiers

In this appendix, a list of some sample additional operations defined for the metaclass `Classifier` is provided, taken from [OMG03d, Section 2.5.3.8]. Note that some OCL expressions are adjusted to be compliant to the adopted OCL 2.0 specification [OMG03b].

- The operation `allFeatures()` results in a set containing all features of the classifier itself and all its inherited features.

```
allFeatures() : Set(Feature) =
    self.feature
    ->union(self.parent.oclAsType(Classifier).allFeatures())
```

- The operation `allOperations()` results in a set containing all operations of the classifier itself and all its inherited operations.

```
allOperations() : Set(Operation) =
    self.allFeatures()
     ->select(f:Feature | f.oclIsKindOf(Operation))
```

- The operation `allAttributes()` results in a set containing all attributes of the classifier itself and all its inherited attributes.

```
allAttributes() : Set(Attribute) =
    self.allFeatures()->select(f:Feature | f.oclIsKindOf(Attribute))
```

- The operation `associations()` results in a set containing all associations of the classifier itself.

```
associations() : Set(Association) =
    self.association.association->asSet()
```

- The operation `allAssociations()` results in a set containing all associations of the classifier itself and all its inherited associations.

```
allAssociations() : Set(Association) =
    self.associations()
    ->union(self.parent.oclAsType(Classifier).allAssociations())
```

- The operation `oppositeAssociationEnds()` results in a set of all association ends that are opposite to the classifier.

```
oppositeAssociationEnds() : Set(AssociationEnd) =
    self.associations()
    ->select(a:Association |
             a.connection->select(ae:AssociationEnd |
                                  ae.participant = self).size() = 1)
    ->collect(a:Association |
              a.connection->select(ae:AssociationEnd |
                                   ae.participant <> self))
    ->union(self.associations()
            ->select(a:Association |
                     a.connection->select(ae:AssociationEnd |
                                          ae.participant = self).size() > 1)
            ->collect(a:Association | a.connection))
```

- The operation `allOppositeAssociationEnds()` results in a set of all association ends opposite to the classifier, including the inherited ones.

```
allOppositeAssociationEnds() : Set(AssociationEnd) =
    self.oppositeAssociationEnds()
    ->union(self.parent.allOppositeAssociationEnds())
```

Appendix C

Structural Constraints for MFERT Models

Similar to an approach that uses a UML Profile to restrict real-time system designs with UML Class Diagrams for validation [AdSSL$^+$01], we here restrict UML Class Diagrams for validation of MFERT as follows.

C.1 ProductionDataType

A Production Data Type defines a tuple[1] of data types. The constraints for `ProductionDataType` are:

1. All attributes must be of a kind of data type.

```
self.allAttributes()->forAll(attr:Attribute |
                             attr.type.oclIsKindOf(DataType))
```

2. Only query operations are allowed for Production Data Types. Constructor operations, i. e., operations that have a name that is equal to the type name, are excluded.

```
self.allOperations()->forAll(op:Operation |
  op.name <> self.name implies op.isQuery = true)
```

3. Production Data Types are passive classes.

```
self.isActive = false
```

4. Production Data Types may only inherit from other Production Data Types.

```
self.allParents()->forAll(g:GeneralizableElement |
  g.stereotype.name->includes('ProductionDataType'))
```

5. A Production Data Type may not have an association among itself.

[1]Note that the package UML::Foundation::Core declares attributes of classes as ordered.

```
self.associations()
  ->select(a:Association |
          a.connection->select(ae:AssociationEnd |
                               ae.participant = self)
                      ->size() > 1)
  ->isEmpty()
```

6. A Production Data Type may only aggregate or be composed of data types or Production Data Types.

```
self.associations()
  ->select(a:Association |
          a.connection->includes(ae:AssociationEnd |
             (ae.aggregation = AggregationKind::aggregate or
              ae.aggregation = AggregationKind::composite)
             and ae.participant = self))
  -- now get all AssociationEnds from selected Associations
  ->collect(a:Association | a.connection)
  ->forAll(ae:AssociationEnd |
      ae.participant.stereotype.name->includes('ProductionDataType')
      or ae.participant.oclIsKindOf(DataType))
```

C.2 ElementList

The `ElementList` stereotype represents a parameterized interface that provides certain operations dedicated to manage lists with elements of a certain Production Data Type. The elements of such lists must all be of the same type which is given as a parameter to `ElementList` and restricted to be a Production Data Type. The constraints of `ElementList` are:

1. This constraint specifies the operations that must at least be provided by classes that are compliant to the `ElementList` interface. We implicitly assume that additional appropriate constructors are available and that the usual FIFO semantics are defined for the operations.

```
let operationNames : Set(Name) =
Set{'getElementType','addElement','getElement','deleteElement'}
in
self.allOperations().name->includesAll(operationNames)
```

2. The parameter must be a Production Data Type.

```
self.typedParameter->size() = 1 and
self.typedParameter.stereotype.name->includes('ProductionDataType')
```

3. Each Element List belongs to at most one `ProductionElementNode`.

```
context ProductionElementNode inv:
  ProductionElementNode.allInstances
  ->forAll(x,y:ProductionElementNode |
          (x <> y implies x.inputSequence <> y.inputSequence) and
          (x <> y implies x.outputSequence <> y.outputSequence) )
```

C.3 MFERTNode

`MFERTNode` is the abstract superclass of `ProductionProcessNode` and `ProductionElementNode`. The constraints of `MFERTNode` are:

1. MFERT nodes are abstract.

   ```
   self.isAbstract = true
   ```

2. MFERT nodes may only inherit from other MFERT nodes.

   ```
   self.allParents()->forAll(g:GeneralizableElement |
     g.stereotype.name->includesAll(self.stereotype.name)
   ```

3. Associations between two MFERT nodes are necessarily modeled using `ElementFlow` associations.

   ```
   MFERTNode.allInstances->forAll(m,n : MFERTNode |
     m <> n implies
       m.associationEnds()
       ->intersection(n.oppositeAssociationEnds())
       ->collect(ae:AssociationEnd | ae.association)
       ->forAll(a:Association |
                a.stereotype.name->includes('ElementFlow'))
   ```

4. There is at most one relationship between each pair of MFERT nodes. It might be a generalization or an association. The latter case is already partially handled. If the relationship is a generalization, the participating MFERT nodes must be of the same subclass, i. e. either Production Process Nodes or Production Element Nodes.

   ```
   MFERTNode.allInstances->forAll(m,n : MFERTNode |
    m <> n implies
    (
      ( m.associationEnds()
        ->intersection(n.oppositeAssociationEnds())->size() <= 1
        and m.allParents()->excludes(n)
        and n.allParents()->excludes(m)
      )
    xor
      (
        (m.allParents()->includes(n) or n.allParents()->includes(m))
        and m.type = n.type
        and m.associationEnds()
            ->intersection(n.oppositeAssociationEnds())->isEmpty()
      )
    )
   ```

5. An MFERT node may not have an association among itself.

```
self.associations()
  ->select(a:Association |
          a.connection->select(ae:AssociationEnd |
                               ae.participant = self)->size() > 1)
  ->isEmpty()
```

6. In MFERT designs, we do not allow aggregation and composition of MFERT nodes.

```
self.associations()
  ->select(a:Association |
          a.connection->includes(ae:AssociationEnd |
                   (ae.aggregation = AggregationKind::aggregate
                    or ae.aggregation = AggregationKind::composite)
                   and ae.participant = self))
  ->isEmpty()
```

C.4 ProductionProcessNode

Production Process Nodes are subclasses of MFERTNodes. They consume from and send production elements to Production Element Nodes. The constraints of `ProductionProcessNode` are:

1. Each Production Process Node has its own thread of control.

```
self.isActive = true
```

C.5 ProductionElementNode

Production Element Nodes are subclasses of MFERT nodes. They store production elements for further processing by subsequent Production Process Nodes. The tagged value `elementType` determines the Production Data Type of the production elements that can be stored. Two lists with production elements are managed by a Production Element Node (one is for incoming, the other for outgoing production elements). The tagged value `time` is used to specify a cyclic interval for shifting of elements between the two lists. The tagged values `inputCapacity` and `outputCapacity` specify the maximal capacity of the lists. The constraints of `ProductionElementNode` are:

1. Production Element Nodes are passive.

```
self.isActive = false
```

2. The two Element Lists are storing instances of the type that is specified by the tagged value `elementType`:

```
self.inputList.getElementType().oclIsTypeOf(self.elementType)) and
self.outputList.getElementType().oclIsTypeOf(self.elementType))
```

3. The value of the tagged values `time`, `inputCapacity`, and `outputCapacity` must be non-negative.

```
self.time > 0 and self.inputCapacity > 0 and self.outputCapacity > 0
```

C.6 ElementFlow

`ElementFlow` represents a restricted association between MFERT nodes. For brevity reasons, the tagged value source is set to the classifier that is identified via the participant association of the first element in the ordered list of association ends (determined by metaclass `AssociationEnd`. The tagged value target is set to the classifier that is identified via the participant association of the second element in the ordered list of association ends. The tagged value `type` identifies a Production Data Type. Only instances of this data type may be transferred between the connected MFERT nodes from the source towards the target end. The constraints of `ElementFlow` are:

1. Element Flow associations are only allowed between two concrete MFERT nodes:

```
self.connection->size() = 2 and
self.connection.participant
  ->forAll(c:Classifier | c.stereotype.name
                             ->includes('ProductionProcessNode')
                          or c.stereotype.name
                             ->includes('ProductionElementNode'))
```

2. The two tagged values `source` and `target` are equal to the two classifiers that are determined by the two association ends of the Element Flow:

```
self.source = self.connection->at(1).participant and
self.target = self.connection->at(2).participant
```

3. `ElementFlow` associations are only allowed between concrete subclasses of MFERT nodes of different types, i. e., between Production Process Nodes and Production Element Nodes:

```
  (self.source.stereotype.name->includes('ProductionElementNode')
   and
   self.target.stereotype.name->includes('ProductionProcessNode'))
xor
  (self.source.stereotype.name->includes('ProductionProcessNode')
   and
   self.target.stereotype.name->includes('ProductionElementNode'))
```

4. Navigation along `ElementFlow` associations is always possible in both directions, i. e., attribute `isNavigable` is true, but only for directly involved classifiers, i. e., `visibility` is protected. We restrict multiplicity of association ends to 1, as an `ElementFlow` association shall indicate a relationship between two instances of MFERT nodes. The `targetScope` is the instance level (this is the default and does not need to be fixed), and an `ordering` does not need to be specified, as only one target end exists. `ElementFlow` associations neither specify aggregation nor composition relationships, so `aggregation` is 'none'. Qualifying attributes are not considered for Element Flows. The following OCL formula summarizes these restrictions:

```
self.connection->forAll(ae:AssociationEnd |
      ae.isNavigable   = true
  and ae.multiplicity  = 1
  and ae.visibility    = VisibilityKind::protected
  and ae.aggregation   = AggregationKind::none
  and ae.qualifier->isEmpty())
```

5. Each `ElementFlow` association is associated with a Production Data Type which is represented by the tagged value `type`. That tagged value must reference to the same type as specified by the participating Production Element Node:

```
(self.source.stereotype.name
   ->includes('ProductionElementNode')
 implies self.type = self.source.elementType)
and
(self.target.stereotype.name
   ->includes('ProductionElementNode')
 implies self.type = self.target.elementType)
```

Appendix D

Property Specification Patterns with OCL

Table D.1: OCL Expressions for Existence Pattern (Assumptions as in Table 3.2)

`P` becomes true . . .	
. . . globally	`init: self@post()->forAll(g | g->includes(P))`
. . . before `R`	`init: self@post()->forAll(g | g->startsWith( Sequence{not R, P} ))`
. . . after `Q`	`inv: self.oclInConf(Q) implies self@post()->forAll(g | g->includes(P))`
. . . between `Q` and `R`	`inv: self.oclInConf(Q) implies` `self@post()->forAll(g | g->startsWith( Sequence{not R, P} ))`
. . . after `Q` until `R`	`inv: oclInConf(Q) implies` `self@post()->forAll(g | g->startsWith( Sequence{not R, P} ))`

Table D.2: OCL Expressions for Universality Pattern (Assumptions as in Table 3.2)

`P` is true . . .	
. . . globally	`inv: self.oclInConf(P)`
. . . before `R`	`init: self@post()->forAll(g | g->startsWith( Sequence{P, R} ))`
. . . after `Q`	`inv: self.oclInConf(Q) implies` `self@post()->forAll(g | g->forAll(conf | conf = P))`
. . . between `Q` and `R`	`inv: self.oclInConf(Q) implies` `self@post()->forAll(g | g->startsWith( Sequence{P, R} ))`
. . . after `Q` until `R`	`inv: self.oclInConf(Q) implies` `not self@post()->exists(g |` `g->startsWith(Sequence{not R, not P and not R}))`

Table D.3: OCL Expressions for Precedence Pattern (Assumptions as in Table 3.2)

S precedes P . . .	
. . . globally	`init: not self@post()->exists(g | g->startsWith( Sequence{not S, P} ))`
. . . before R	`init: self@post()->forAll(g | g->startsWith( Sequence{not P, S or P} ))`
. . . after Q	`inv: self.oclInConf(Q) implies` `self@post()->forAll(g | g->startsWith( Sequence{Q, not P, S} ))`
. . . between Q and R	`inv: self.oclInConf(Q) implies` `self@post()->forAll(g | g->startsWith( Sequence{not P, S or R} ))`
. . . after Q until R	`inv: self.oclInConf(Q) implies` `not self@post()->exists(g | g->startsWith( Sequence{not S and not R, P} ))`

Table D.4: OCL Expressions for Response Pattern (Assumptions as in Table 3.2)

S responds to P . . .	
. . . globally	`inv: self.oclInConf(P) implies self@post()->forAll(g | g->includes(S))`
. . . before R	`inv: self.oclInConf(P) implies` `self@post()->forAll(g | g->startsWith( Sequence{not R, S} ))`
. . . after Q	`inv: self.oclInConf(Q) implies` `self@post()->forAll(g | g->includes( Sequence{P, true[0,'inf'], S})`
. . . between Q and R	`inv: self.oclInConf(Q) implies self@post()->forAll(g |` `g->includes( Sequence{P, not R [0,'inf'], S, not R [0,'inf'], R} ))`
. . . after Q until R	`inv: self.oclInConf(Q) implies self@post()->forAll(g |` `g->startsWith( Sequence{not R, P, not R[0,'inf'], S} ))`