UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Dissertation

Object-Oriented Specification of Distributed Systems

Author:

Ulrike Lechner

Advisors:

Prof. Christian Lengauer, Ph. D., and **Prof. Dr. Martin Wirsing**

June 23, 1997

Abstract

Since object-oriented languages model worlds of interacting, independent objects, admitting concurrency in such languages seems to be straightforward. But the integration of concurrency and object-oriented structuring concepts, especially inheritance, induces a new object model and calls for new formal techniques and a new design method. We aim at the design of safety-critical systems in a high-level object-oriented language, whose correctness can only be ensured by the formal techniques of refinement and verification.

Our work is based on the object-oriented concurrent specification language Maude. Maude's distinctive features are its abstract and simple object model and its abstract synchronization and communication mechanisms.

We develop concepts for designing, structuring and reusing object-oriented specifications in Maude, and techniques for verifying and for refining these specifications.

In verification, we provide abstraction techniques and develop notions of invariants appropriate for Maude specifications. In refinement, we develop concepts and techniques appropriate for the refinement of classes, of communication and synchronization patterns and of specifications.

A focal point is the reuse of Maude specifications. We develop a set of concepts for structuring and reuse: states as classes, inheritance, subconfigurations and message algebras. We gain a high degree of code reusability and demonstrate this with examples. We establish the connection between the reuse constructs of Maude and relations between classes of models, and characterize the classes of properties that can be inherited via the different reuse concepts.

Maude is our main specification language. We employ algebraic and coalgebraic specifications and the modal μ-calculus, for covering facets of concurrent objects and levels of abstraction not represented in Maude. We instantiate our object model and our structuring and reuse concepts at various levels of abstraction.

The bounded buffer and a specification of an airport are the running examples. All specifications have been implemented in CafeOBJ.

Acknowledgments

I am indebted to Professor Christian Lengauer and Professor Martin Wirsing. They initiated the topic of my thesis, encouraged and guided my work with patience, circumspection and with interest. They have been not only advisors, but also partners in fruitful discussions and coauthors in various publications. Special thanks to Professor Christian Lengauer for his particular indefatigability in correcting my English.

I would like to thank Prof. Niels Schwartz for his help with all the formal aspects on the way to a Ph.D.

Jose Meseguer's work on Maude and the Rewriting Logic has been seminal for my thesis. I am obliged to him for his support during is entire development.

Natalija Krivokapič and Nils Ellmenreich helped me by proofreading parts of this thesis. Friederike Nickl has contributed not only as a coauthor. She has played a vital and very helpful role in my professional career since my first undergraduate term.

Alexander Knapp and Toshimi Sawada helped me in my struggle with CafeOBJ. Tiziana Margaria-Steffen and Bernhard Steffen as well as Gerald Lüttgen, Stephan Merz and Susanne Graf provided valuable feedback.

The meetings with Uwe Nestmann, Martin Steffen and Terry Stroup as well as with Benjamin Pierce and, in particular, with Grit Denker at various occasions were both encouraging and fruitful.

I am also indebted to all system administrators for providing me with an excellent computing environment. Here I would like to mention Martin Griebl, Nils Ellmenreich, Andreas Stübinger, Sven Anders and Max Vogl as well as Detlef Menzel and Klaus Schießl.

Sonja Wildbihler helped me with her excellent management of the library.

However, during the time it takes to write a thesis, friends are invaluable for advice and, more importantly, for distraction and encouragement.

Barbara's charming visits were highlights during the time it took to write the thesis.

Special thanks to Professor Winfried Hahn and Natalija Krivokapič whose encouragement has been decisive for this thesis.

Thank you to my colleagues Nils Ellmenreich, Sergei Gorlatch, Christoph Herrmann, Christoph Wedler and, in particular, Martin Griebl, as well as Johanna Bucur and Ulrike Peiker for a good working climate.

Christine and Stefan Gastinger, Georg Heßmann, Andrea Häusler, Ingeborg Meissner, Bernhard Reus, Robert Stabl, Wolf-Dieter Tiedemann, Kirsten Zbil-Bielefeld and Eric Zbil encouraged me. I enjoyed numerous discussions at dinners with Susanne Buchwieser, Sandra Burg, Dagmar Felix, Dorit Huber, Nana Krivokapič and Verena Lauren.

The Go games with John O'Donnell, Nils Ellmenreich, Dirk Koschützki and Bernhard Steffen were a challenging diversion. I enjoyed the Skat games with Prof. Siegfried Graf, Andreas Hagerer, Prof. Winfried Hahn, Prof. Christian Lengauer and Prof. Niels Schwartz.

Many thanks to Sandra Burg, Silvia Hurt, Natalija Krivokapić and the substitutes Nils Ellmenreich and Ahmed Mian Syed for excellent dinners and entertaining games.

Thank you for your friendship during all these years.

Volker Braun, Achim Dannecker, Claudia Gsottberger, Dirk Koschützki, Tiziana Margaria-Steffen and Bernhard Steffen gave me the opportunity to visit CeBIT'97, which turned out to be decisive for my professional future. Thank you.

This thesis would not have been possible without my family. The support by Christine and Stefan and, in particular, by my parents has been invaluable.

I am indebted to the DFG for funding. Travel support has been granted by the DAAD, through its ARC program and by the DFG.

Passau, June 23, 1997 Ulrike Lechner

Contents

Chapter 1

Introduction

Computer science has been creating a universe within science. The components of this universe, the computers and the languages are man-made. This gives us the freedom to change and customize this universe to our needs.

With the systems they model, programmers create their own worlds within this universe. These worlds are abstract and, often, very complex. The science of programming imposes a new, abstract and structured way of thinking. The freedom in the design, the creativity that it takes to create such a new world, and the complexity of these programs demand a programming method to support programmers with adequate means to express their ideas and to offer them a discipline of programming.

Concepts that make a programming paradigm adequate for the design of complex programs are an abstract way of reasoning, the support of modular programming and the reusability of code. These concepts have proven to be crucial for the development of complex software systems. But, for safety-critical systems, programming method has to be complemented with formal methods to express and verify program properties precisely.

The object-oriented paradigm has proven adequate for the design of large systems and its popularity both in academia and industry has been increasing over the past years.

1.1 The Object-Oriented Paradigm

The object is the center of this paradigm. An object is an autonomous entity: it consists of its data describing the state, and methods, i.e., operations to access and manipulate the data. The most important concept in object orientation is *encapsulation*: the state of an object can be accessed and manipulated only through the methods. Objects are instances of classes. A class is an abstraction of a number of objects. A program is a collection of classes which models a system as a collection of communicating objects. Characteristic for the object-oriented paradigm are also the means for reuse. Classes can be derived from existing classes in a process of incremental modification and enhancement. Object-oriented design methods support programmers from the first informal steps to the final program [Boo91, BR95, CY91a, CY91b, JCJÖ92, RBP$^+$91, SM92, WBWW90].

Simula [BDMN73] is considered to be the first object-oriented language. The most popular languages nowadays in industry are: Smalltalk [GR83, Sch96], Eiffel [Mey92], C++ [Str86] and Java [AG96].

Let us have a closer look at *the* object-oriented language Smalltalk [GR83]. Smalltalk is a homogeneous object-oriented language, i.e., every construct in the language is an object. It has few but very powerful concepts: objects, method invocation, classes and inheritance. The expressiveness is obtained through the large library in which Smalltalk's control constructs and standard data types are implemented. Smalltalk is a reflective language. This means that it is possible to access and modify the object model, the execution model and even the whole programming environment within Smalltalk.

1.2 Motivation and Scope

Smalltalk and other object-oriented languages like C++ and Eiffel provide little or no support for concurrency, although concurrency seems to fit very naturally into the paradigm of object orientation. However it turns out that concurrent object orientation differs very much from sequential object orientation. The reason is that, in order to integrate concurrency, the object model has to be changed. This demands that the way objects communicate, the reuse concepts and the programming and design method be revisited.

We develop the kernel of an object-oriented concurrent programming paradigm. Our work is based on the object-oriented concurrent specification language Maude, developed by J. Meseguer [Mes92b, Mes93a, Mes93b, Mes96, MW92] as a notation for his Rewriting Logic [CM93, Mes90a, Mes92b, Mes96, MMO95, MOM94, MOM96]. We develop concepts for reuse and enrich Meseguer's Maude with these new constructs. We develop refinement techniques to refine Maude specifications and verification techniques for Maude specifications. Our framework comprises also a programming method. For Maude specifications we adopt the notation implemented in the CafeOBJ system [FN96, Sch97].

Our goal is the structured development of complex systems in an object-oriented style. Our approach combines the advantages of both object orientation and formal specification languages.

- Object orientation profits from the use of formal methods. Requirements for object-oriented models are phrased in an informal or semi-formal way and object-oriented programs are typically validated, not verified. Informal descriptions of requirements can be ambiguous, inconsistent and incomplete. Only requirements phrased in a formal language, e.g., a specification language, can be checked and validated and only verification techniques can guarantee properties of programs. Moreover, the presence of concurrency is the source of many errors in programs and thus formal methods are essential to ensure the correctness.

- Object orientation has advantages for the use of formal methods, since object orientation helps to structure a specification. Specifications of complex systems are

> typically complex as well. Thus object-oriented structuring and reuse concepts assist programmers in mastering this complexity.

Our approach is aimed at the design of complex systems in an object-oriented language. Maude specifications can play two different roles in the design of such a system. The *first role* of a Maude specification is to be an executable prototype, which allows to validate the specification and to detect conceptual errors at an early stage of the design process. The *second, for us more important role* is to be a formal description of a system and to be thus the basis for verification and refinement. Maude is an abstract language and Maude's object model is the one of a high-level language. Although object orientation in the fields of programming languages and data bases have much in common and although Maude has been successfully used in the field of data bases [DG93, MQ93] we restrict ourselves to programming languages.

Our main contribution is in the field of specification languages, but our results have implications on the design of object-oriented programming languages as well. Our specification language describes computational progress—similar to a programming language. Maude plays a *third role* as a notation to reason about the constructs appropriate for object-oriented concurrent languages at various levels of abstraction.

At the abstract level of Maude, we can separate properties from implementation details. In our approach, we go even one step further and abstract also from typical object-oriented language constructs. This keeps our object-oriented language and its semantics simple. Furthermore it makes the development of a language, its semantics, refinement and verification technique and a programming method feasible. We work at the level of a specification language, but the concepts are so concrete that they can be applied at the level of a programming language.

We develop the kernel of an object-oriented concurrent programming paradigm. However, we remain in the world of formal specifications. We have omitted several issues relevant for making our approach acceptable to object-oriented programmer: an implementation of Maude, an object-oriented analysis and design method and case studies to demonstrate the expressiveness. Here we would like to refer to other work on case studies, implementations and semi-formal specifications. Let us give here a brief survey on complementing work.

Work on rewriting systems, which can be used as Maude interpreters, is in progress. Sequential interpreters are [BKK$^+$96b, Eke96, FD97]. Strategies to improve the performance of term-rewriting systems are developed in [BKK96a, CM96]. The Rewrite Rule Machine [AGL$^+$92, LMOMR94], a massively parallel computer, is designed for executing Maude specifications. The speed of the sequential interpreters allows to execute specifications of reasonable size whereas the results of the simulation of the Rewrite Rule Machine promise that large specifications can be executed at a very high speed [AGL$^+$92]. We have checked our Maude specifications with the CafeOBJ system [Sch97, FN96].

In our work we develop design concepts but we cannot claim to offer an object-oriented analysis and design method. More experience and more and larger case studies would be necessary to develop such a design method. Moreover, we remain within the world of formal specifications and do not deal with the semi-formal, often graphical notations of object-oriented analysis and design methods, as they are typical in object orientation. To relate our formal approach to specification with the informal graphical notations common in object orientation, we refer to work combining informal graphical notation and formal specification [Nic94, NW93, WK96]. In [WK96] the informal graphical notation of the design method OOSE [JCJÖ92] is enriched by formal assertions to enable the (partially) automatic translation of graphical notation into a specification.

Maude as a specification language is a general-purpose language. We specify the bounded buffer in several different versions, we specify a communication protocol and use the example of an airport based on [Sal93] to demonstrate techniques for a large specification. [Mes96] gives an excellent survey of the work based on the specification language Maude. To demonstrate that Maude is expressive and a general-purpose language, let us mention some examples of Maude specifications. Trees and algorithms on trees have been specified in [Har95]. Maude has been used to give semantics to another object-oriented concurrent language, TROLL light [DG93]. CCS [Mes92b, Mil89], the π-calculus [FMQ96, MPW92, Vir96], an actor language [Tal96], Petri Nets [Mes92b] and image processing algorithms [Mes92b] have been specified in Maude. Maude has been used to implement a data base [MQ93]. The expressiveness of Maude and the rewriting logic is demonstrated also by its reflectiveness, i.e., the possibility to access and manipulate the execution model of Maude. The rewriting calculus and Maude have been extended in [ÖM96, WK97] to deal with real time. Reflection and strategies for Maude have been specified in Maude in [CM96].

1.3 From Sequential to Concurrent Object-Oriented Languages

The step from a sequential to a concurrent object-oriented language seems to be small: concurrency in a collection of communicating objects seems to be a very straightforward concept. However, it induces severe changes of the object model and, thus, changes of all the concepts that are essential for object-oriented languages: the communication mechanisms and the concepts for reuse.

Let us illustrate this in more detail, using the example of the bounded buffer, the classical example of a concurrent object-oriented program [AFK$^+$93, Frø92, MCR93, MY93, Mes93b, Nie93]. A bounded buffer is described as follows. Elements can be stored only in a non-full bounded buffer. Elements can be retrieved only from a non-empty bounded buffer. In a concurrent setting, there might be several users which concurrently store and retrieve elements from a single buffer.

Let us discuss which protocols are appropriate for communication between users and bounded buffers. In a sequential language, the users ask whether the bounded buffer is full or empty, and store and retrieve their elements accordingly. In a concurrent setting, this protocol can only be used if atomicity of the first request for the status, and the second store or retrieve request can be ensured. Achieving this atomicity would require complicated synchronization mechanisms and an object model much more complicated than the one of sequential languages. *The simplest protocol for the implementation of communication between objects is to let an object decide on the basis of its internal state whether it accepts a message or not.* This is implemented by the so-called *synchronization code*. The synchronization code inside an object determines whether an object accepts a message or not. We argue that this is the protocol that suits the object-oriented paradigm best, since it advocates objects as independent entities. Furthermore, it is a protocol suitable for specifications since it is abstract. In our object model, an object contains data describing its state, methods to access and modify the data and synchronization code to determine when an object accepts a message.

1.4 Object-Oriented Specification of Distributed Systems

There are numerous ways [Nie91] to combine concepts from object orientation and sequential and concurrent specification languages. We make a number of design decisions when creating the object model for our object-oriented concurrent paradigm which we explain and motivate in this section.

1.4.1 Object Model

The example of a bounded buffer illustrates how communication between objects can be implemented. In our model objects decide whether they accept a message or not. This is a very simple, abstract and object-oriented protocol.

We choose *asynchronous message passing* as our communication mechanism. The reason is that it suits the concept of objects as autonomous entities. Thus, objects communicate via messages and, when an object accepts a message, a method is executed and performs the state change. An answer is sent as a message. The socalled *synchronization code* is part of a class and determines whether an object accepts a message.

In the way we have described the object model, we presume that we have a state-based specification language.

The two concepts, asynchronous message passing and synchronization code, make our objects independent, autonomous entities. On the other hand, concurrency gives rise to a new dimension of problems as, e.g., the occurrence of deadlocks. One way to cope with these problems is to introduce another, more abstract way of communication, *synchronous implicit communication*. A number of objects can synchronize and exchange data to compute their internal states. At the level of a specification language, such an implicit communication is appropriate: we do not care about how the communication is implemented, who triggers the communication and who answers, and how many objects participate. We are only interested in the resulting state. With such powerful and abstract communication mechanisms, we have to keep the number of actions small and make reasoning about the properties feasible.

1.4.2 Concepts for Reuse

Reuse and, in particular, incremental modification by inheritance of classes is a distinguishing feature of object-oriented languages. The term “inheritance anomaly” [MY93] refers to the problems of combining concurrency and inheritance and the necessity to re-implement large portions of code when trying to reuse classes with inheritance. These problems are caused by the presence of synchronization code and inherent to it, no matter how the synchronization code is implemented. We would like to have concepts for reuse which, in our approach, means not necessarily exclusively inheritance. Thus, we develop a set of new structuring and reuse concepts to complement the notion of inheritance: states as classes, subconfiguration and message algebra.

At the abstract, property-oriented level of a specification language, *reuse of code is inheritance of properties*. We use the term “inheritance” as a generalization of all reuse constructs. At the abstract, property-oriented level of a specification language, the properties, and not (parts of) the code should be inherited. This is a main conceptual difference between reuse in specification on the one hand and in programming languages on the other hand.

1.4.3 Language Philosophy

We use a *heterogeneous specification language*. Homogeneous languages certainly have advantages: they are smaller and their semantics is simpler. For the comprehensibility of specifications it is crucial that the properties one would like to express suit the paradigm of the specification language. For the specification of a complex, possibly heterogeneous system, one has to choose between a large monolithic specification language and a heterogeneous specification using several small languages adequate for different parts of the specification. We give the pragmatism of a heterogeneous language preference over the elegance of a homogeneous language. Data types are specified with an algebraic specification language and behavior of objects as a state transition system. Moreover, also from the object-oriented point of view, our language is a heterogeneous object-oriented language, since we support data types other than objects.

We distinguish between different phases in the design of an object-oriented system. In each phase, we are interested in a different set of properties.

- The first phase comprises the implementation of the classes and the class hierarchy. Here we are interested in the properties of classes, i.e., the properties of the data and the operations on the data inside the objects, as well as the communication and synchronization of objects. In particular, we are interested in the properties that can be inherited via the reuse relations provided by the language.

- In the second phase, a system is modeled by objects that are instances of the class library implemented in the first phase. Thus we consider a collection of objects with their behavior and their properties. In this case we are interested in all kinds of properties of distributed systems as, e.g., reachability of states, absence of deadlocks or occurrence of deadlock.

Corresponding to the two phases, we have three levels of specifications: the intra-object level, the inter-object level and the link between these two levels.

- At the *intra-object* level, object-oriented specification is about the specification of the data that make up the state of an object and the properties of the operations on the data. Since we abstract from the implementation of the methods and since we have serialized objects and do not deal with all the problems of intra-object concurrency, only the properties of the data are relevant. Thus, the specification language we use at this level is an algebraic specification language.

- At the *behavioral* level, the methods and their properties are specified. This level provides the encapsulation of the implementation of the state and the methods. The methods and their properties are only observable but not visible or accessible like at the intra-object level. Here, we use the formalism of behavioral specifications of [Jac96a, Jac96b, Jac96c].

- At the *inter-object* level, the communication and synchronization between objects and the computational progress of a system is described. This level is covered by Maude.

1.4.4 Formal Methods

A specification language alone is not of much use unless it is accompanied by a method for *refining specifications to programs*. We specify which transitions may take place and the synchronization code specifies when an object *may* accept a message. These are also properties expressed by programming languages. Refinement is feasible, since our specification style describes computational progress in an operational way, similarly to a programming language.

For a specification language which has only a certain kind of properties that are directly expressible, there have to be *means to verify other properties*. Moreover, the specification language is operational and, thus, we would like to have methods to reason on a more property-oriented level.

1.5 Object Orientation, Concurrency and Specification

We develop an algebraic approach to object-oriented concurrent specification. We employ the concepts of objects, classes and messages and of structuring and reuse from the object-oriented paradigm. We employ traditional techniques from algebraic specification and complement them with concepts of concurrent specifications. We look for abstraction mechanisms to model encapsulation, for reuse concepts and for relations between specifications and programs.

1.5.1 Algebraic Specifications

A specification is a property-oriented description which abstracts from implementation details. Specifications are the basis for formal proofs of properties of systems.

In [Wir95], specification languages are classified into model-oriented, property-oriented and type-oriented languages. They are characterized by their underlying logic, their constructs supporting an underlying paradigm and their structuring mechanisms. Algebraic specification languages as, e.g., ASL [Wir87], OBJ3 [GKK$^+$88] and OS [Bre91] belong to the class of property-oriented specification languages. In an algebraic specification language, a software module is described by giving a signature consisting of sorts and operation symbols as a static interface. Equational axioms specify the characteristic properties of the operations in algebraic specifications. The property that is expressible within algebraic specifications is equality.

Specification languages support different programming paradigms: OS [Bre91] and OOSpectrum [WNL95] are algebraic, object-oriented specification languages, CIP-L [Bau85], LARCH [GHW85] and COLD [Jon89] support the imperative programming paradigm, Spectrum [BFG$^+$93] the functional paradigm.

The relation between a specification and a program is established by refinement. In the process of stepwise refinement, the property-oriented descriptions are replaced by more and more concrete implementations. In this process, gradually more information on the representation of data types and algorithms is added. The result of such a process of stepwise refinement is a program. The initial and the loose approach provide a formal basis for refinement from a specification to a program [ONS93, SJE92, Wir90, Wir95].

To make refinement feasible, i.e., to relate a specification to a concrete program, there have to be means of abstraction. Behavioral specification abstracts from some of the properties [Hen89]. While an algebraic specification gives the characteristic properties of a data type, a behavioral specification gives only the observable properties. Recent developments in behavioral specification make it possible to consider not only the algebraic but also the concurrent and object-oriented paradigm in a uniform way [BHW95, HS95, Jac96a].

1.5.2 Specification of Concurrent Systems

There is a variety of languages modeling concurrent systems as, e.g., the process algebras CCS [Mil89], CSP [Hoa85] and π-calculus [MPW92, FMQ96]. Common to all these languages is that they have an operational semantics and describe transition systems.

The process algebra used in object orientation is the π-calculus [Jon93c, MPW93, NS96, PT97]. Collections of objects have mobile communication structures, i.e., object identifiers are passed between objects and, thus, which objects "know" each other and are thus able to communicate, varies during run time. Only the π-calculus can express these mobile communication partners.

There is a spectrum of abstract process semantics which provide abstraction from properties of transition systems [vG90a]. Bisimulation is the finest abstraction in this spectrum: processes are identified when they cannot be distinguished by observation.

The μ-calculus [Dam93, Koz83, Sti92], a branching-time logic [Krö87], is a property-oriented way of reasoning about concurrent systems. The μ-calculus characterizes bisimilar processes. Weaker logics characterize coarser equivalences between processes [vG90a]. Model checking is the technique of verifying that temporal logic formulas are satisfied in a transition system [CES86, SCK$^+$95]. In abstract interpretation [CC78, LGS$^+$95, Ste93, SKV96] a property phrased in the μ-calculus is verified for an "abstract" transition system, which is (bi)similar to the "concrete" transition system one is interested in. Since bisimilar processes satisfy the same set of properties, one can infer that the property of the abstract transition system holds. This approach can be applied to coarser relations and subclasses of μ-formulas as well [LGS$^+$95].

One way of refinement in a concurrent specification language is action refinement [vG89], where an action is replaced by a number of actions. In general, such a refinement preserves hardly any properties [vG90a]. [Lar93] presents a property-preserving refinement of modal transition systems.

Explicit concurrency is error-prone and hard to control. Thus, one approach to refinement is to exploit the implicit concurrency in a specification and refine it to an explicitly parallel program [Len82, LH82]. Basically this works best for very regular structures with bounds which can be determined statically [LH86, HL87]. We adapt some of the criteria developed for this kind of refinement to establish our refinement relations.

1.5.3 Relation between Object Orientation and Specification Languages

The formal basis for the description of properties of data types with algebraic specification and of behavior with transition systems seems to be quite different. But when one looks more closely at the very basics of algebras and transition systems the underlying principles are compatible.

The semantics of algebraic specifications are algebras, the semantics of concurrent languages transition systems. Algebras and transition systems (or coalgebras) are dual concepts [HR95, Jac96c, Rei95, Wir90]. Algebraic specifications describe characteristic properties of data types, while coalgebraic specifications describe observable properties of data types. Algebraic specifications construct data types while coalgebraic specifications observe data types. Thus, we use algebras to describe the properties at the intra-object level following [Bre91, Wir90] and transition systems to model the behavior of objects at the behavioral and the inter-object level. Here we follow [HR95, Jac96c, Mes92a, Rei95]. Techniques of behavioral specifications [BH93, BHW95, Hen89, HS95] are the link between the two different concepts of algebraic and coalgebraic specifications.

1.6 Contents

In Chap. 2, we give an introduction to our specification language, Maude. First, we explain the basic features as the object model, communication and concurrency in Maude. We develop a set of reuse constructs for Maude and discuss the problems of reusing object-oriented concurrent code. Our reuse constructs are states as classes, inheritance, subconfigurations and message algebras. The example with which we demonstrate both the basic language constructs of Maude as well as the reuse concepts is the bounded buffer, a classical example in concurrent object-oriented languages. We develop a programming method for structuring complex specifications in Maude and illustrate with it the role of various ways to structure class declarations and to model concurrency, communication and synchronization in Maude. We demonstrate how our reuse concepts can be used to model a large and complicated specifications. We give guidelines for the use of the language concepts. For specifications we use not only Maude but also two other specification formalisms: algebraic and coalgebraic specifications. We explain both briefly.

Chap. 3 is an introduction to the formal background of Maude. Our approach is an algebraic approach to object-oriented concurrent specifications. Thus we give a brief introduction to order-sorted algebras and models as well as the mappings: order-sorted signature morphisms, homomorphisms and specification morphisms. Order-sortedness induces several problems and thus we require our specifications to have the property of coherence. We explain with examples why this property is a necessity. Furthermore we give properties that the signature morphisms have to have in order to preserve the properties of signatures. We extend the notion of order-sorted specification with a relation. We discuss the role of partiality in Maude specifications and demonstrate how we use order-sortedness to deal with specifications. We compare and relate the two dual specification formalisms, the algebraic and coalgebraic specification formalism and give Maude an algebraic and a coalgebraic interpretation.

In Chap. 4, we develop verification techniques appropriate for Maude specifications. We choose the μ-calculus to be our notation for properties of Maude specifications. We use a framework of abstract interpretation [CC78, LGS+95] to verify properties for Maude specifications. This framework is combined with algebraic techniques to be suitable for Maude. As an example we verify a mutual exclusion property for Maude and thus our focus is here on the inter-object level. We use the notion of abstract interpretation to reason about the properties that are inherited by our reuse constructs. We use formula schemata to describe properties of our objects and discuss which of the properties expressed as formula schemata can be inherited.

Chap. 5 deals with refinement techniques for object-oriented concurrent specifications. We develop three different approaches for refinement.

1. In the refinement of a class, i.e., the refinement of the implementation of a class, an abstract, property-oriented implementation is refined to a concrete executable

implementation.

2. In the refinement of a communication pattern, we refine abstract communication patterns to communication patterns, which are used in object-oriented programming languages. We demonstrate this with a refinement of a positive acknowledge and retransmission protocol.

3. For the refinement of a complete system we develop a new notion of behavioral refinement, which is sort-independent, and where the refinement relation depends of the properties to be preserved by the refinement.

Chap. 6 concludes our work with a brief review of the language, the formal techniques and the programming method and our design decisions.

An appendix contains some specification code and several proofs.

Mathematical notation

Our mathematical notation follows Dijkstra [DS90]. Quantification over a dummy variable x is written $(Q\,x\ :\ R(x)\ :\ P(x))$. Q is the quantifier, R a predicate in x representing the range of the dummy and P a term that depends on x. E.g. $(\forall x : x \in X \wedge \phi(x) : \psi(x))$ is the universal quantification of the predicate $\psi(x)$, where the domain of x is X and $\phi(x)$ is a condition on the dummy x. We abbreviate the set comprehension $(\cup x : x \in X \wedge \phi(x) : \{f(x)\})$ by the more traditional $\{f(x) \mid x \in X \wedge \phi(x)\}$.

Formal logical deductions are written:

$$
\begin{array}{ll}
 & \mathit{formula}_1 \\
\text{op} & \quad \{ \text{ comment explaining the validity of this relation } \} \\
 & \mathit{formula}_2
\end{array}
$$

Chapter 2

Object-Oriented Specifications in Maude

In the object-oriented paradigm, the world is modeled as a collection of autonomous communicating entities, the objects. With object-oriented specifications of distributed systems in Maude, we follow the object-oriented paradigm: a program or specification consists of a collection of communicating objects and Maude provides the necessary encapsulation and concepts to structure and reuse specifications. Maude is a specification language, thus Maude specifications are abstract, property-oriented descriptions of a world of communicating entities.

We think of Maude specifications as to be used in the development of complex concurrent systems. The object model of Maude for high-level applications, i.e., the objects are the concurrent entities. Thus, the typical application field of Maude specifications is in the development of high-level applications as opposed to low-level programs. Herein, Maude specifications can play several different roles.

The first role is to be a first prototype, which can be used to validate informal specifications as they are typical in object-oriented design methods. Such a prototype is helpful for detecting conceptual errors or shortcomings at an early state in the design process, where it is both relatively cheap and easy to correct them. Here, it is crucial that Maude specifications are executable [Eke96], that they are so abstract that complex communication and synchronization patterns can be described concisely and that the language provides the reuse and structuring concepts typical for object orientation.

The second role of a Maude specification is to be the formal basis for the refinement of a specification to a program or for a verification of a program. In the object-oriented design philosophy, a program is a collection of classes, in which ideally each class is compact and easy to understand. This facilitates the verification of the behavior of classes. But, when all the classes are being put together and objects are created as their instances, the behavior of a collection of objects has to be verified. The more concurrency an object-oriented language admits, the more important formal methods become. Here, it is crucial that Maude specifications deal with both the behavior of single classes and the behavior of a collection of objects. We develop accordingly refinement and verification techniques in Chap. 4 and Chap. 5.

Maude plays for us a third role. We use Maude specifications as a notation for reasoning about the object model and for constructs appropriate for object-oriented concurrent languages in general.

Particular to Maude is its simple object model; Maude abstracts from the methods and Maude specifications describe only the states and state changes of objects. The global state consists of the local states of the objects and the messages pending to be processed. The behavior of objects is specified in transition rules, which express which transitions *may* happen. Particular to Maude are the synchronization and communication mechanisms: objects communicate via messages and a number of objects may synchronize in order to perform a state change in which they exchange the data to compute the resulting state. We call the communication via messages *asynchronous explicit* communication, the communication and synchronization of a number of objects *implicit synchronous* communication.

The specification language Maude has been designed by J. Meseguer [MW92, Mes93b, Mes93a, Mes96]. We take Meseguer's basic principles: the object model, inheritance, the synchronization and communication mechanism and transition rules to specify the behavior. We add to this constructs for structuring and reusing specifications and develop a specification method. Furthermore we complement Maude's abstract object model, which covers only state and state changes with specifications that deal with the intra-object level, i.e., that specify the properties of methods. We choose the notation of the CafeOBJ system [Sch97, FN96]. The rewriting calculus that we use to compute the transitions computes a labeled transition system, in contrast to Meseguer's unlabeled transition systems.

Based on the main notation, Maude, its object model, communication and synchronization mechanisms, we develop our own notion of object-oriented specification.

So, *What is object-oriented specification?* There are several answers to this question. Which one chooses depends on the specification language and, in our view, also on the level of granularity. We distinguish three levels of specification:

- At the *intra-object* level, object-oriented specification gives the properties of states and the methods of a single class. At this level the original question can be stated more precisely as *What is a class?* or *What are the properties of objects belonging to a class?*

- A *behavioral specification* describes the properties of a class which are observable from outside. This level considers an object to be an entity encapsulating the state and its implementation and allowing access to the state only via methods. Thus, here we are interested in *How do objects of a class behave?*

- At the *inter-object* level, we specify synchronization and communication between objects. At this level, the original question can be stated more precisely as: *How do objects communicate and synchronize?*

All these questions cover different aspects of object orientation, for which we use different specification approaches. Maude is the main notation we use and provides us with our framework. Specifications in Maude deal with the inter-object level of object-oriented worlds, i.e., with communication and synchronization between objects. With algebraic object-oriented specifications we specify the properties of a class. Behavioral specifications specify the observable behavior of a class.

The structuring and reuse concepts make the object-oriented paradigm superior to other programming paradigms. We develop new structuring and reuse concepts and show their generality and orthogonality. Moreover, we give a method of how to structure specifications in Maude.

This chapter is organized as follows. Sect. 2.1 is an introduction to Maude. We introduce new reuse concepts for Maude in Sect. 2.2 and apply them in Sect. 2.3 to demonstrate their power. In Sect. 2.4, we develop a specification method appropriate for large Maude specifications. Algebraic and coalgebraic specifications of single classes are provided in Sect. 2.5. Sect. 2.6 contains an extensive survey of other object-oriented concurrent languages in specification and programming.

2.1 Introduction to Maude

The specification language subsumes two different language paradigms: one in which the basic data types are given in equational (algebraic) specifications, and one in which state transitions between configurations are specified in transition rules. The conceptual difference between these two worlds is described in Sect. 2.1.2. The algebraic part is OBJ3 [GKK+88]; it forms the subset of the language used to specify the properties of data types in a purely algebraic way. The object-oriented state-modeling part is specified by an operational (state transition) semantics given by so-called *transition rules*.

Note that our notation does not follow Maude's syntax as presented in Meseguer's papers [MW92, Mes93b, Mes93a, Mes96]. We use the syntax of CafeOBJ [Sch97, FN96].

2.1.1 Object-Oriented Specification in Maude

The nucleus of each object-oriented language is the object. In Maude, we have a particularly simple object model: objects in Maude are state-based, and we specify the state changes. Let us introduce our favorite object, the bounded buffer, which will accompany us throughout the thesis. A bounded buffer is modeled by an object of class `BdBuffer`. Our object has object identifier `B1`. An object of class `BdBuffer` stores its capacity in an attribute called `max`, the number of elements that have been put into a buffer is stored in the attribute `in` and the number of elements taken out from a buffer in the attribute `out`. The elements a bounded buffer "buffers" are stored as a list in attribute `cont`. Maude's notation for such a bounded buffer, with a capacity of three, which already had stored four elements and from which three elements have been taken, and which contains one element, `E2`, at present is:

```
< B1 :  BdBuffer | max = 3, in = 4, out = 3, cont = E2 >
```

Objects communicate via messages. Messages trigger state transitions of objects. A message consists of a message name and parameters. Typically the object to which it is addressed is one of those parameters. To improve the readability of the specification, we use mixfix notation for the messages. In the example of the bounded buffer, a message to which we refer as `put` consists of a message name, `put`, a parameter `B`, which will later turn out to be the addressee and a parameter `E`, which is the element being put into the buffer:

```
(put E into B)
```

Transition rules[1] specify how objects react to messages. In the example, a bounded buffer *may* react to a `put` message if it is not full, i.e., if the value of `in` minus the value of `out` (denoted by `sd(I,O)`) is less than the value of `max`. A transition rule may have a label (`P` in this example):

```
crl [P]: (put E into B)
         < B : BdBuffer | cont = C, in = I,
                          out = O, max = M >
         =>  < B : BdBuffer | cont = E C, in = I + 1,
                              out = O, max = M >
         if sd(I,O) < M .
```

The keyword which denotes a conditional transition rule, i.e., a transition rule with a Boolean condition, is `crl`. The keyword `rl` indicates an (unconditional) transition rule. An example is the specification of the reaction of a bounded buffer to a message `get`:

```
rl [G]: (get B replyto R)
        < B : BdBuffer | cont = C E, in = I,
                         out = O, max = M >
        => < B : BdBuffer | cont = C, in = I,
                            out = O + 1, max = M >
           (to R answer to get is E) .
```

The left-hand side and the precondition stated at the end determine when a state transition *may* happen. If the pattern at the left-hand side of a rule matches the current state, the object involved changes its state according to the rule. New messages may be produced according to the rule. In our example `P`, a message with message name `put` and a bounded buffer have to exist and the object identifier in the message and in the bounded buffer have to be identical in order to enable the bounded buffer to accept the message `put`. The value of `in` is increased and the element `E`, which is a parameter to the `put` message, is added to the list of contents. In this rule no answer message is created. In rule `G`, which models the reaction of a bounded buffer to a `get` message, an answer message `(to R answer to get is E)` is created as part of the global state.

[1] Meseguer calls these rules rewriting rules.

The *global state* of an object-oriented system is called a *configuration*. A configuration is a multiset (or bag) of objects and messages. We require that object identifiers are unique within a configuration. A configuration that contains two bounded buffers and two messages with message name `put` is depicted in Fig. 2.1.

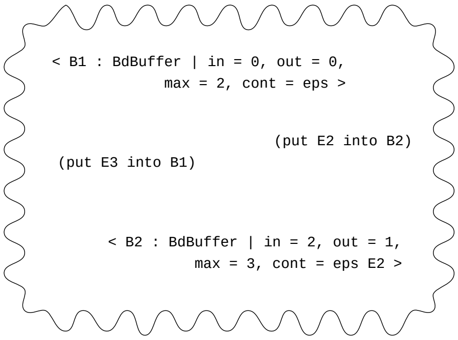

Figure 2.1: Configuration with two bounded buffers and two messages

In this configuration, three transitions are possible: (1) `B1` may accept a `put` while `B2` is idle, (2) `B2` may accept a `put` while `B1` is idle or (3) `B1` and `B2` may both accept a `put`. Thus, we obtain three possible successor states. This is depicted in Fig. 2.2. The application of transition rules is governed by a rewriting calculus given in Sect. 2.1.3.

The complete specification of a bounded buffer comprises all the specification fragments we have explained up to now. This specification and slightly modified versions of it, will be presented throughout this work. Let us first give the specification of the bounded buffer, and explain it in detail afterwards:

```
module BD-BUFFER {
  import {
    protecting (NAT)
    protecting (LIST)
    protecting (EXT-ACZ-CONFIGURATION)
  }

  signature {
    class BdBuffer {
      in   : Nat
      out  : Nat
      max  : NzNat
      cont : List
    }

    class Proto {
      class : ClassId
```

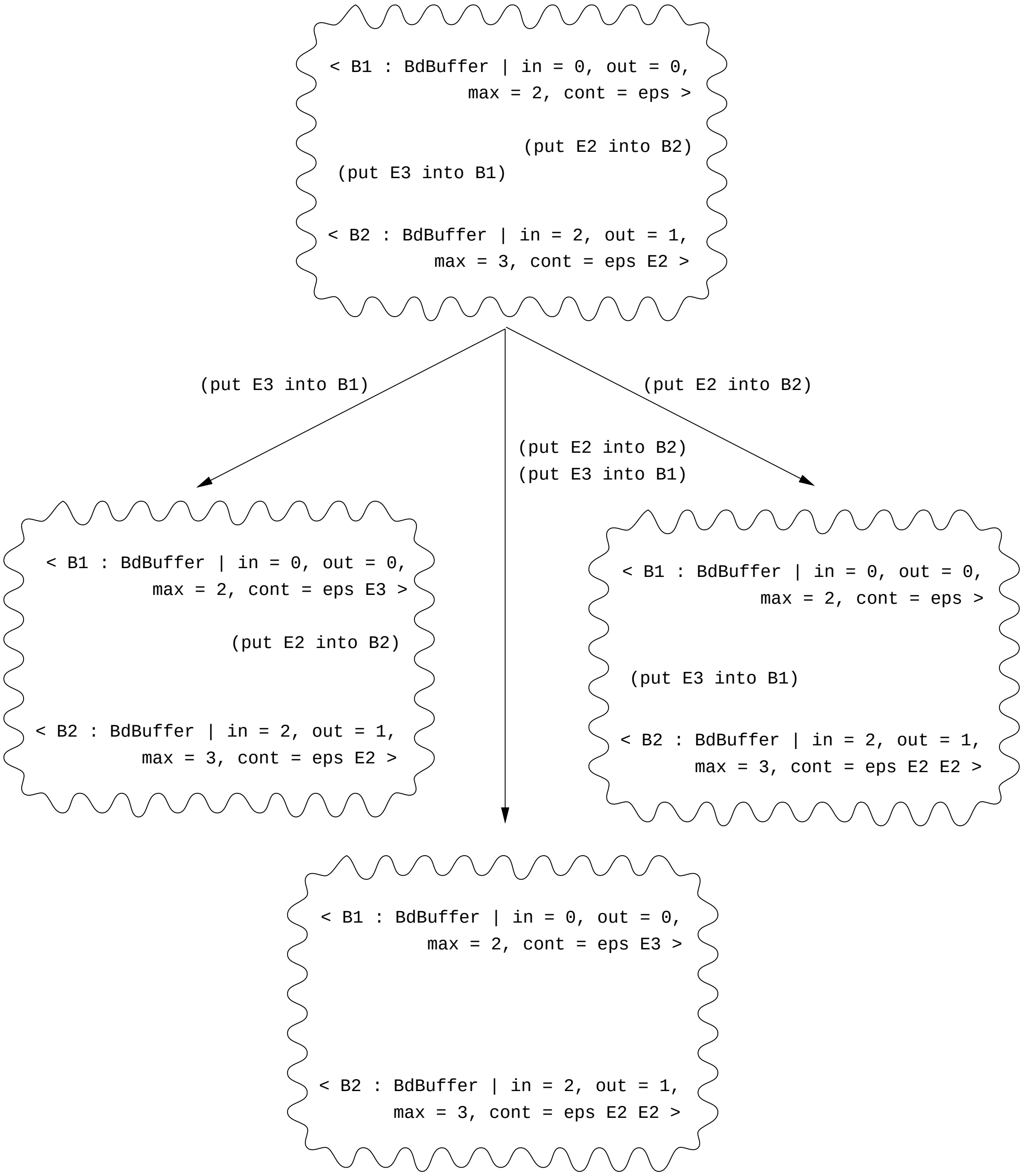

Figure 2.2: State changes

```
      next  : ObjectId
    }

     op new BdBuffer with _ replyto _ : NzNat ObjectId -> Message
     op to _ the new BdBuffer is _    : ObjectId ObjectId -> Message
     op get _ replyto _               : ObjectId ObjectId -> Message
     op to _ answer to get is _       : ObjectId Elem -> Message
     op put _ into _                  : Elem ObjectId -> Message
   }

  axioms {
    vars B R U P : ObjectId
    var  E       : Elem
    var  C       : List
    vars I O     : Nat
    var  M       : NzNat
    var  ATTS    : Attributes

  crl [P]: (put E into B)
           < B : BdBuffer | cont = C, in = I,
                            out = O, max = M, ATTS >
           =>  < B : BdBuffer | cont = E C, in = I + 1,
                                out = O, max = M, ATTS >
           if sd(I,O) < M .


   rl [G]: (get B replyto R)
           < B : BdBuffer | cont = C E, out = O, max = M, ATTS >
           => < B : BdBuffer | cont = C, out = O + 1, max = M, ATTS >
              (to R answer to get is E)  .


   rl [N]: (new BdBuffer with M replyto U)
           < P : Proto | class = BdBuffer, next = B >
           => < P : Proto | class = BdBuffer, next = incoid(B) >
              < B : BdBuffer | cont = eps, in = 0, out = 0, max = M >
              (to U the new BdBuffer is B)  .
 }
}
```

Specifications consist of three parts: (1) import, (2) signature and (3) axioms. Specification `BD-BUFFER` imports specification `NAT`, which specifies the natural numbers and a particular sort, `NzNat`, the non-zero natural numbers. Specification `LIST` contains a specification of lists and comprises the two sorts `Elem` (Elements) and `List`. `eps` denotes the empty list. We have an overloaded binary function symbol _ _, denoted by a space, for concatenation of elements and lists: `E L` denotes a list constructed from first an element `E` and a then a list `L`, whereas `L E` is a list consisting of a list `L` and an element `E`. In specification `(EXT-ACZ-CONFIGURATION)` all the basic data types are defined. This includes, e.g., object identifiers (`ObjectId`), messages (`Message`) and configurations (`ACZ-Configuration`).

Specification `LIST` and `EXT-ACZ-CONFIGURATION` are given in App. A.1, which contains also a specification `LIST-ENRICHED` in which several functions on lists are provided.

Specifications `BD-BUFFER` contains two class declarations. Class `BdBuffer` has four attributes: `in`, `out`, `max` and `cont`. A bounded buffer may react to two messages: `put` and `get`. `Put` stores an element in the buffer, `get` removes the first element being stored in the buffer and sends it to a "receiver". The transition rule with rule label `P` says that an object of class `BdBuffer` can react to a `put` message only if the actual number `sd(I,O)` of objects being stored, is smaller than the capacity `max` of the bounded buffer. Sending a `get` message not only triggers a state change of buffer `B` but also initiates an answer message.

The second class of this specification, class `Proto`, is responsible for creating objects of class `BdBuffer`. While rules `P` and `G` model the standard reaction of a buffer to `put` and `get` messages, rule `N`, which models the creation of a bounded buffer, is particular to Maude. To create a new buffer, a message `new` is sent to a proto-object (of class `Proto`) which is responsible for creating new objects of class `BdBuffer`. Message `new` triggers the creation of a new object. Hereby, default values are assigned to attributes `in`, `out` and `cont`. The value of `max` is determined by a parameter of message `new`. The proto-object changes (increases) the value of parameter `next` by `incoid`, and we assume that this is done such that the same object identifier is never created twice. For each class, we have (at least) one proto-object. The existence of more than one proto-object for a class in the configurations enables us to create several instances of this class in parallel.

Generally speaking, transition rules specify *explicit, asynchronous communication* via message passing: if a message is part of a configuration, a state transition may happen and new (answer) messages waiting to be processed in subsequent state transitions may be created as part of the resulting configuration (in the specification given above only one new message is generated). The transition rules specify not only the behavior but also the equivalent of the synchronization code of other languages: the pattern given at the left-hand side involves not only the presence of objects and messages but also certain properties like the equality of object identifiers, values of attributes and parameters of the messages. (We could also specify more than one object at the left-hand side of a transition rule and specify a synchronous state transition of several objects.)

Maude is a very powerful and abstract language. Omitting an object at the right-hand side of a rule, which is present at the left-hand side, deletes this object. Omitting attributes at the right-hand side of a rule deletes these attributes as well. This is an advantage when specifying changes in the global and local states which involve the dynamic creation of objects and dynamic changes of the classes and thus the attributes of objects. However, it forces us to write down much more syntax than needed to describe the necessary changes. Note that we do not apply the notational convention of Meseguer's Maude [Mes93a, MW92], where we are allowed to omit attributes at the right-hand side of a rule whose values are not changed. Later we will use a variable, typically named `ATTS`, of sort `Attributes` to collect all the attributes and their values which are not needed and, thus, not explicitly mentioned in a rule. This variable collects also all attributes, which are particular to all classes that are subclasses of the class for which the rule is given.

To complete the specification of the bounded buffer, we give a specification of a sender and receiver which communicate with the bounded buffer:

```
module SRB {
  import {
    protecting (BD-BUFFER)
  }

  signature {
    class Sender {
      sendq  : List
      buffer : ObjectId
    }

    class Receiver {
      incoming : List
      buffer   : ObjectId
    }

    op new Sender with _ and _ replyto _ : List ObjectId ObjectId
                                                        -> Message
    op new Receiver with _ replyto _     : ObjectId ObjectId -> Message
    op to _ the new Sender is _          : ObjectId ObjectId -> Message
    op to _ the new Receiver is _        : ObjectId ObjectId -> Message
  }

  axioms {
    vars  S R B U : ObjectId
    vars  C L     : List
    vars  I O     : Nat
    var   E       : Elem
    var   ATTS    : Attributes

    rl [S1]: < S : Sender | sendq = L E, buffer = B, ATTS >
          => < S : Sender | sendq = L, buffer = B, ATTS >
             (put E into B) .
```

```
    rl [R1]: < R : Receiver | ATTS >
             => < R : Receiver | ATTS >
                (get B replyto R) .

    rl [R2]: (to U answer to get is E)
             < R : Receiver | incoming = L, buffer = B, ATTS >
             => < R : Receiver | incoming = E L, buffer = B, ATTS >
                (get B replyto R) .

    rl [NR]: (new Receiver with B replyto U)
             < P : Proto | class = Receiver, next = R >
             => < P : Proto | class = Receiver, next = incoid(R) >
                < R : Receiver | incoming = eps, buffer = B >
                (to U the new Receiver is R) .

    rl [NS]: (new Sender with L and B replyto U)
             < P : Proto | class = Sender, next = S >
             => < P : Proto |  class = Sender, next = incoid(S) >
                < S : Sender | sendq = L, buffer = B >
                (to U the new Sender is S) .
 }
}
```

In specification `SRB` (for Sender, Receiver, Buffer), we specify the behavior of two classes, a sender and a receiver, such which interact with the bounded buffer of specification `BD-BUFFER`. Rules `S1` and `R1` model the autonomous behavior of the sender and the receiver. Such rules are particular to Maude. An object may decide to perform an autonomous state change, i.e., a state change that is not triggered by a message. E.g., it is the sender's autonomous decision whether to send a `put` message to a bounded buffer. A receiver may decide autonomously (rule `R1`) or triggered by an answer (rule `R2`) to send a `get` message to the buffer.

Note that sender and receiver store the object identifier of their buffer in an attribute, while the parameters of the `put` and `get` message carry the object identifier to which the answer or acknowledgment message is sent. The transition rules provide enormous flexibility in specifying how objects and messages synchronize, how they have to fit together to make a transition happen and also how the final state of the transition is computed.

Up to now, we have not used the full power of Maude's communication and synchronization mechanism. Transition rules may specify a synchronization between several objects and messages. Accordingly, a transition rule may contain several objects at the left- and right-hand side of a transition rule. Let us give, as an example, a specification of a buffer, sender and receiver:

```
module SYNC-SRB {
  import {
    protecting (NAT)
    protecting (LIST)
    protecting (EXT-ACZ-CONFIGURATION)
   }

  signature {
    class BdBuffer {
      in   : Nat
      out  : Nat
      max  : NzNat
      cont : List
    }

    class Sender {
      sendq  : List
      buffer : ObjectId
    }

    class Receiver {
      incoming : List
      buffer   : ObjectId
    }

    op get _      : ObjectId -> Message
    op put into _ : ObjectId -> Message
   }

  axioms {
    vars S R B U : ObjectId
    var  E       : Elem
    vars C L L'  : List
    vars I O     : Nat
    var  M       : NzNat
    vars Buffer-ATTS Sender-ATTS Receiver-ATTS : Attributes

  crl [P]: (put into B)
           < S : Sender | sendq = L E, buffer = B,
                          Sender-ATTS >
           < B : BdBuffer | cont = C, in = I, out = O, max = M,
                            Buffer-ATTS >
           => < B : BdBuffer | cont = E C, in = I + 1, out = O, max = M,
```

```
                              Buffer-ATTS >
                < S : Sender | sendq = L, buffer = B,
                            Sender-ATTS >
             if sd(I,O) < M .

  rl [G]:    (get B)
             < R : Receiver | incoming = L, buffer = B,
                            Receiver-ATTS >
             < B : BdBuffer | cont = C E, in = I, out = O, max = M,
                            Buffer-ATTS >
             => < B : BdBuffer | cont = C, in = I, out = O + 1, max = M,
                              Buffer-ATTS >
                < R : Receiver | incoming = E L, buffer = B,
                              Receiver-ATTS > .

  crl [PG]: (put into B)
            (get B)
            < S : Sender | sendq = L E, buffer = B,
                         Sender-ATTS >
            < R : Receiver | incoming = L', buffer = B,
                           Receiver-ATTS >
            < B : BdBuffer | cont = C, in = I,
                           out = O, max = M, Buffer-ATTS >
            => < B : BdBuffer | cont = E C, in = I + 1,
                             out = O + 1, max = M, Buffer-ATTS >
               < S : Sender | sendq = L, buffer = B,
                            Sender-ATTS >
               < R : Receiver | incoming = E L', buffer = B,
                             Receiver-ATTS >
            if sd(I,O) < M .
  }
}
```

Rule `P` specifies a synchronous `put` between a sender and a bounded buffer. The action is triggered by a message `put` that carries only one parameter, namely the object identifier of the buffer. The pattern at the left-hand side of the transition rule ensures that a sender puts the elements into a specific buffer, namely the buffer whose object identifier the sender has stored.

Analogously, rule `G` models a synchronous version of `get`. Here, the receiver and the bounded buffer synchronize, the oldest element is removed from the attribute `cont` of the buffer and is stored in the attribute `incoming` of the buffer.

In contrast to specification `SRB`, specification `SYNC-SRB` ensures that the elements being put into a buffer actually arrive at the buffer. This makes the acknowledgment and the answer messages superfluous.

Rule PG specifies a concurrent put and get. Provided the buffer is neither full nor empty, a bounded buffer, a receiver and a sender can synchronize and then perform a transition in which an element is being put by the sender into the buffer and an element is taken from the buffer and stored by the receiver. This is an example of a rule involving synchronization between more than two objects. Note that the pattern at the left-hand side together with the precondition specify that the buffer is neither empty nor full.

The behavior specified in SRB is particular to Maude and typical for a Maude specification. The rules describe inter-object synchronization, involving up to three objects (in rule RG) and a joint state transition with a joint computation of the final state of the objects involved.

2.1.2 Equational Specification

The specification language Maude comprises two different concepts of specification: algebraic and object-oriented specification. In this section, we deal with the algebraic style, i.e., with equational specifications and the differences to the object-oriented specifications with state transitions. As an example, we use the basic data type of lists:

```
module LIST {

  signature {
    [Elem]
    [List]

    op eps : -> List
    op _ _ : Elem List -> List
    op _ _ : List Elem -> List
  }

  axioms {
    vars E E1 E2 : Elem
    var  L : List

    eq [Eq1]:  E2 eps = eps E2 .
    eq [Eq2]:  E1 (L E2)  = (E1 L) E2 .
  }
}
```

Let us explain the specification. Two sorts, Elem and and List, are declared. Specification LIST contains three operations: eps constructs a list, and the overloaded function symbol _ _ denotes construction of a list from an element and a list as described earlier. The specification contains two equations specifying equality of lists. These two equations are unconditional (keyword eq). Conditional equations begin with the keyword ceq.

This specification does not contain subsorts. For an equational specification with subsorts, see specification `LIST-WITH-SUBSORTING` in Sect. 3.2.

The characteristic properties of the operations are given by equations. Equality is an equivalence relation. From `a = b` and `b = c` we can infer that `b = a` and `a = c`. State transitions are not commutative: from `a => b` we certainly do not want to be able to infer `b => a`. Assume we have a configuration `c` and two possible transitions `c => d1` and `c => d2`. If we modeled transitions as equations, we could infer by the transitivity of equality that `d1 = d2`.

2.1.3 Rewriting Calculus

The transition rules are applied by a rewriting calculus to configurations. We employ in our work two different calculi: the original one by Meseguer and our calculus. Let us give the original one first.

There are four rules in Meseguer's rewriting calculus:
Reflexivity. For each $[\,t\,] \in T^E(\Sigma, X)$:

$$\frac{}{[\,t\,] \to [\,t\,]}$$

Transitivity.

$$\frac{[\,t_1\,] \to [\,t_2\,],[\,t_2\,] \to [\,t_3\,]}{[\,t_1\,] \to [\,t_3\,]}$$

Congruence. For each $(f : s_1 \times \cdots \times s_n \to s) \in F$ and $t_i \in T^E(\Sigma, X)_{s_i}$

$$\frac{[\,t_1\,] \to [\,t_1'\,] \;\ldots\; [\,t_n\,] \to [\,t_n'\,]}{[\,f(t_1, \ldots , t_n)\,] \to [\,f(t_1', \ldots , t_n')\,]}$$

Replacement. For each rewrite rule $r : [\,t(x_1, \ldots , x_n)\,] \to [\,t'(x_1, \ldots , x_n)\,]$ in T:

$$\frac{[\,w_1\,] \to [\,w_1'\,] \;\ldots\; [\,w_n\,] \to [\,w_n'\,]}{[\,t(\overline{w}/\overline{x})\,] \to [\,t'(\overline{w}'/\overline{x})\,]}$$

The replacement rule allows simultaneous rewrites. Note that transition rules can only be applied concurrently when their redexes are disjoint.

This calculus forms a categorical structure [Mes92a] with states as objects and transitions as morphisms. The transition system described by this calculus is very abstract: it abstracts from the way state transitions are computed. It is irrelevant how many "single steps" are necessary to compute a state transition, and which messages are consumed in these transitions. However, we would like to have a more concrete view of the systems we specify. First of all, we would like to know which actions take place in a transition. Thus, our rewriting calculus defines a labeled transition system. We choose as labels the messages that are consumed. They model the "actions" we are interested in. In order to observe single transitions, we do not have a transitivity rule in our calculus.

Our rewriting calculus, given below in five rules, defines Maude's semantics in the form of a *labeled transition system*. In the following, let m, m' denote messages, a_i attribute names, v_i and w_i values, o_i object identifiers, C_i, C_i', D_i and D_i' class identifiers, $atts_i$ sets of pairs of attributes together with their variables, and σ a substitution. Let $\{a_i = w_i\}$ be a set of attributes and $\{\texttt{<}\,\sigma(o_i) : D_i \mid \{a_i = v_i\}\texttt{>}\}$ a set of objects and $\{m\}$ a multiset of messages.

A transition

$$\begin{array}{ll} \{m\}[\sigma] & \\ \{\texttt{<}\,\sigma(o_i) : D_i \mid \{a_i = v_i\}[\sigma],\ atts_i\,\texttt{>}\} & \\ \xrightarrow{\{m\}[\sigma]} \ \{\texttt{<}\,\sigma(o_i) : D_i' \mid \{a_i = w_i\}[\sigma],\ atts_i\,\texttt{>}\} & \\ \qquad\quad \{m'\}[\sigma] & \end{array} \tag{Inst}$$

is possible if T contains a transition rule (in which all attributes of classes C_i together with their values are stated)

$$\begin{array}{ll} [R] & \{m\} \\ & \{\texttt{<}\,o_i : C_i \mid \{a_i = v_i\}\,\texttt{>}\} \\ & \texttt{=>}\ \ \{\texttt{<}\,o_i : C_i' \mid \{a_i = w_i\}\,\texttt{>}\} \\ & \qquad \{m'\} \end{array}$$

and a substitution

$$\sigma : Vars \to T(\Sigma, X),$$

where

$$D_i \leq C_i$$

and

$$D_i' = \begin{cases} D_i, & \text{if } C_i = C_i' \\ C_i' & \text{else} \end{cases}$$

Let us explain this rule. A transition rule in T is instantiated, such that all variables of the rule are substituted according to a substitution σ. The classes of the objects of the configuration to which the rule is applied are subclasses of the objects of the rule. Since an object of a subclass may have more attributes than the object of the superclass, we introduce $atts_i$ to match the additional attributes of the subclass. Values of those attributes are not changed in the transition. The values v_i are changed to w_i according to the rule. For simplicity we assume that no objects are created or deleted by the transition rule. We make two simplifying assumptions for the case that objects change their classes: the class of the object at the right-hand side of the rule becomes the class of the (instance) object, and classes between which class changes are possible have the same attributes.

As a notational convention, we may omit attributes at the left- and right-hand side of a rule whose values are not needed in the transition, and additionally all attributes at the right-hand side whose values are not changed.

In the case of a conditional transition rule of the form:

$$m'_1\, o'_{i_1} \ldots o'_{i_n} \texttt{ => } o'_{j_1} \ldots o'_{j_m}\, m'_2 \ldots m'_n \texttt{ if } p_1 \wedge \ldots \wedge p_k$$

with (unconditional) equations or (conditional) transitions $p_1, \ldots, p_k$, we require additionally that all $p_i[\sigma]$ are derivable.

We need two more rules: (Emb) embeds the left-hand and the right-hand side of a transition into a configuration, containing objects and messages not changed by the transition, and (Equ) makes the transition relation compatible with equations. Let $c, d, c', d', c_1, c_2, d_1, d_2$ and h be configurations. Let $\{m\}, \{m_1\}, \{m_2\}$ be multisets of messages and let $=_E$ denote equality modulo equations in the set E:

$$c\ h \xrightarrow{\{m\}} d\ h \text{ if } c \xrightarrow{\{m\}} d \tag{Emb}$$

$$c' \xrightarrow{\{m'\}} d' \text{ if } c \xrightarrow{\{m\}} d \text{ and } c =_E c', d =_E d', \{m\} =_E \{m'\} \tag{Equ}$$

In the presence of subconfigurations, we have to introduce one more rule, (Sub), to the rewriting calculus which specifies the application of transition rules to subconfigurations (see Sect. 2.2.3). Let $\texttt{<}\ o : C \mid a : S,\ atts\ \texttt{>}$ be an object containing a subconfiguration S stored under the attribute name a, and let $atts$ denote all other attributes with their values of o apart from a:

$$\texttt{<}\ o\!:\!C \mid a\!:\!S,\ atts\ \texttt{>} \xrightarrow{\{m\}} \texttt{<}\ o\!:\!C \mid a = S',\ atts\ \texttt{>} \text{ if } S \xrightarrow{\{m\}} S' \tag{Sub}$$

These rules allow to apply one transition rule to a configuration. Thus, they establish an interleaving semantics. To model true concurrency we have to provide a rule that allows to apply more transition rules in parallel.

$$c_1\ c_2 \xrightarrow{\{m_1\ \ m_2\}} d_1\ d_2 \text{ if } c_1 \xrightarrow{\{m_1\}} d_1 \text{ and } c_2 \xrightarrow{\{m_2\}} d_2 \tag{Par}$$

The label of a parallel composition of two transition systems is the set of the labels of the two transitions.

In contrast to Meseguer's calculus, our own calculus has neither a reflexivity rule nor a transitivity rule. Rule (Emb) is weaker than the replacement rule in the original calculus; the replacement rule could be obtained by (Emb), (Equ), and a transitivity rule.

The rewriting calculus gives us an interpretation of our specifications. A specification describes (a class of) transition systems (see Chap. 3). We require for our transition rules that the variables at the right-hand side of the rules and the precondition are instantiated at the left-hand side of the transition rules. The reason for this is that an uninstantiated variable of a type with an infinite domain of values would lead to infinitely branching transition systems. With the requirement that all variables be instantiated, the final state of a transition is the result of a computation not the result of an arbitrary choice. In a finite configuration and with a finite number of equations and transition rules, only finitely varying applications of transition rules are possible. Thus, the transition systems described by our Maude specifications are finitely branching. The condition on the class identifier ensures that transition rules describing state and class changes of objects are monotonic since the objects change class such that they keep the least possible sorts.

2.2 Object-Oriented Structuring Concepts

The most common and popular means to structure the code of object-oriented programs is inheritance. Inheritance and concurrency are two paradigms which are difficult to combine in a satisfactory way. In [MY93] the necessity of reprogramming a large portion of inherited code is called the *inheritance anomaly*. In [Mes93b] it is claimed that the main reason for the inheritance anomaly is the presence of *synchronization code*, the code which determines which method calls can be accepted depending on the state of the object. The classical example of the use and the necessity of synchronization code is the bounded buffer which allows its get method to be invoked only if it is not empty [Frø92, MY93, Mes93b].

It is claimed that Maude overcomes the inheritance anomaly: in [Mes93b], Meseguer demonstrates how to use Maude's inheritance mechanism and modules with various ways of reuse to specify concurrent systems in a structured and modular way.

We agree with [Mes93b] in that the inheritance anomaly is caused by the *presence of synchronization code*, not by the way this code is structured and implemented. But, in resolving the inheritance anomaly, we develop a new idea: Maude's inheritance mechanism, which allows only to add new attributes and new possibilities of state changes, is not sufficient. We introduce two new concepts: a *subconfiguration* and an *algebra of messages*. They allow us to reuse code not only by enhancing the possible state transitions but also by restricting them and by composing actions out of basic actions in an object-oriented way. We are interested in the *reusability of code*—and this means for us *not* necessarily only inheritance.

Specifications are property-oriented descriptions and can be used as formal basis for verification and validation. When designing structuring and reuse concepts, we have properties in mind that can be "inherited" or are preserved by the structuring concepts. For inheritability of properties by the reuse constructs, see Sect. 4.4.

In this section, we develop concepts to structure and reuse object-oriented concurrent specifications or programs. We begin with states as classes to structure the specification of a single class, we continue with inheritance and subconfigurations and the concept of message algebras. We introduce these concepts first. Then we apply them to the specification of a bounded buffer to demonstrate how they overcome the inheritance anomaly.

2.2.1 States as Classes

The state of an object plays a decisive role in the way the object reacts to a message sent to it—most fundamentally, whether it reacts at all! A buffer accepts a get message only if it is not empty and a put message only it is not full. The states and the transitions between the states are depicted in a state transition diagram in Fig. 2.3.

Compared to other, more conventional languages, the transition rules in Maude describe a very dynamic behavior: objects may not only change the values of attributes, they may be created, and may disappear and they may change class. We take advantage of this and model a bounded buffer such that each of the three states, empty, full, and neither full nor empty, is implemented by a separate class.

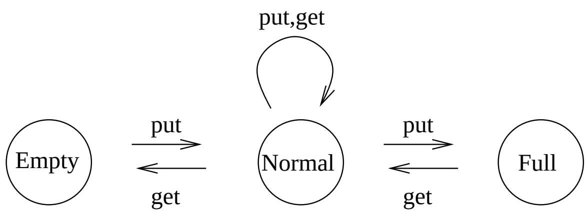

Figure 2.3: States of the bounded buffer

```
module BD-BUFFER-WITH-STATES {
  import {
    protecting (LIST-ENRICHED)
    protecting (EXT-ACZ-CONFIGURATION)
  }

  signature {
    class BdBuffer {}

    class EmptyBdBuffer [BdBuffer] {
      max : NzNat
    }

    class FullBdBuffer [BdBuffer] {
      cont : List
    }

    class NormalBdBuffer [BdBuffer] {
      cont : List
      max  : NzNat
    }

    class Proto {
      class : ClassId
      next  : ObjectId
    }

     op new BdBuffer with _ replyto _ : NzNat ObjectId -> Message
     op to _ the new BdBuffer is _    : ObjectId ObjectId -> Message
     op get _ replyto _               : ObjectId ObjectId -> Message
     op to _ answer to get is _       : ObjectId Elem -> Message
     op put _ into _                  : Elem ObjectId -> Message
   }
```

```
axioms {
  vars B U  : ObjectId
  var  C    : List
  var  E E' : Elem
  var  M    : NzNat
  var  ATTS : Attributes

rl [P1] : (put E into B)
          < B : EmptyBdBuffer | max = M, ATTS >
          => < B : NormalBdBuffer | cont = E eps, max = M, ATTS > .

crl [P2]: (put E into B)
          < B : NormalBdBuffer | cont = C, max = M, ATTS >
          => < B : NormalBdBuffer | cont = E C, max = M, ATTS >
          if length(C) + 1 < M .

crl [P3]: (put E into B)
          < B : NormalBdBuffer | cont = C, max = M, ATTS >
          => < B : FullBdBuffer | cont = E C, ATTS >
          if (length(C) + 1 == M) .

rl [G1]:  (get B replyto U)
          < B : FullBdBuffer | cont = C E, ATTS >
          => < B : NormalBdBuffer | cont = C, max = length(C) + 1,
                                    ATTS >
             (to U answer to get is E) .

rl [G2]:  (get B replyto U)
          < B : NormalBdBuffer | cont = C E' E, ATTS >
          => < B : NormalBdBuffer | cont = C E', ATTS >
             (to U answer to get is E)

rl [G3]:  (get B replyto U)
          < B : NormalBdBuffer | cont = eps E, max = M, ATTS >
          => < B : EmptyBdBuffer | max = M, ATTS >
             (to U answer to get is E) .

crl [N]:  (new BdBuffer with M replyto U)
          < P : Proto | class = BdBuffer, next = B >
          => < P : Proto | class = BdBuffer, next = incoid(B) >
             < B : EmptyBdBuffer | max = M >
             (to U the new BdBuffer is B)
             if  M > 1 .
```

```
  }
}
```

In general, there are two different ways of modeling states. In one model, states are represented by attributes; the values of attributes are used to determine whether or not a message is accepted. This model can also be expressed in other specification languages. In the other model, states are represented by classes; the class determines whether or not an object accepts a message. We argue that this model is more object-oriented—and typical for Maude because it makes better use of Maude's concept of class inheritance and Maude's convenient provisions for letting an object change its class membership (see Sect. 2.2.1). A state change is modeled by making the object change its class. This model requires (at least) one class per state.

In the specification above, we have three classes that model states. While the class and state determine whether an object reacts, the values of attributes determine only how an object reacts. A bounded buffer in state `FullBdBuffer` accepts a `get`, not a `put` method and the value of attribute `cont` determines the value of the answer message.

The advantage of the states-as-classes approach is that the state descriptions are smaller: an empty buffer does not need an attribute to store contents, a full buffer does not need to store the maximal number of elements, provided it is able to compute the length of the contents. The states-as-classes approach not only avoids unnecessary ballast in the object model, it also makes reasoning about the consistency of the state of objects even simpler (see Sect. 4.2).

Note that a bounded buffer in this specification changes not only its class, but also the description of the state, i.e., its attributes, dynamically. This justifies the explicit notation of the attributes as described earlier.

2.2.2 Inheritance Relation

In Maude a heir class inherits all attributes, all equations and all transition rules from its ancestor class. The inheritance relation is realized via a subsort relation:

```
class Heir [A1 ... An] {
  atts
}
```

This declaration declares class `Heir` to be a subclass of classes `A1` ... `An`. Thus, we allow multiple inheritance. The names of the attributes of classes `A1` ... `An` have to be disjoint when they are not inherited to these classes from common superclasses. Attributes of classes `A1` ... `An`, which are inherited from a common superclass, are identified in the heir class. We may declare some attributes `atts` to be new to class `Heir`.

Maude's inheritance relation allows only to add new transitions to the transitions possible for the heir. It does not provide a means of redefining the behavior of an object or disabling transitions.

2.2.3 Subconfigurations

With subconfigurations, we can structure our configuration by permitting an object to contain configurations. We would like to have modest control about which objects and messages are part of a subconfiguration. Thus, we introduce a new sort, `Subconfiguration`, which is a subsort of sort `ACZ-Configuration`, the sort of the global state. We impose no restriction on which messages may be part of a subconfiguration. Thus, sort `Message`, the sort of messages, is a subsort of sort `Subconfiguration`.

The implementation of this subconfiguration concept is given below in specification `SUBCONFIGURATION`.

```
module SUBCONFIGURATION {
  import {
    protecting (ACZ-CONFIGURATION)
  }

  signature {
  [Message < Subconfiguration < ACZ-Configuration]
  }
}
```

We specify explicitly which messages may pass from a configuration into a subconfiguration or vice versa and, thus, we need not impose restrictions on the messages in the subconfiguration construct.

An example of a class declaration using the subconfiguration construct is:

```
[BdBuffer Flag < Subconfiguration]

class HBdBuffer {
  conf : Subconfiguration
}
```

Objects of class `HBdBuffer` have an attribute `conf`, which is a subconfiguration that contains only objects belonging to class `BdBuffer` or class `Flag`.

2.2.4 Message Algebras

Maude is an object-oriented language which admits, compared to other object-oriented languages, very fine-grained concurrency. Thus, it becomes necessary to apply reuse not only to classes but also for the messages and the transition rules.

Process algebras provide language constructs to compose processes from single actions. We adopt this principle and compose messages with message combinators. We specify the semantics of the message combinators with Maude's transition rules. A transition rule models a single atomic transition. A transition that is triggered by a composed message is composed from the transitions triggered by (uncomposed) messages, and is an atomic transition as well. For the specification of the composition of messages we use the reachability relation, denoted by the operator `==>`, which is specified in module `RWL`.

The message combinators and their semantics are specified in Maude. This demonstrates reflection in Maude: Maude and its execution model are subject to a Maude specification.

In the following specification `MSG-ALGEBRA` we implement message combinators that we use later in Sect. 2.3:

```
module MSG-ALGEBRA {
  import {
    protecting (ACZ-CONFIGURATION)
    protecting (RWL)
  }

  signature {
    op _ +  _  : Message Message -> Message
    op _ ;  _  : Message Message -> Message
    op _ ;; _  : Message Message -> Message
    op _ |  _  : Message Message -> Message
    op _ || _  : Message Message -> Message
   }

  axioms {
    vars m n m1 m2 n1 n2   : Message
    vars c d c1 c2 d1 d2 h : ACZ-Configuration

  crl [C]: (m + n) c => d
          if m c ==> d  or
             n c ==> d .

  crl [S1]: (m1 ; m2) c => d
           if  m1 c   ==> h  and
               m2 h   ==> d .

  crl [S2]: (m1 ;; m2) c => d (n1 ;; n2)
           if m1 c  ==> n1 h and
              m2 h  ==> n2 d .

  crl [P1]: (m1 | m2) c1 c2 => d1 d2
           if m1 c1 ==> d1 and
              m2 c2 ==> d2 .
```

```
  crl [P2]: (m1 || m2) c1 c2 => d1 d2 (n1 || n2)
            if m1 c1 ==> n1 d1 and
               m2 c2 ==> n2 d2 .
  }
}
```

Specification MSG-ALGEBRA contains three message combinators: choice (+), sequential composition (; and ;;) and parallel composition (| and ||). We have two versions of sequential and parallel composition: one which composes the answer messages following the structure of the input message (;; and ||) and one which does not pass this structure on to the next configuration (; and |).

Generally, we have the freedom to specify any kind of sequential or parallel composition or choice in a message algebra. We have chosen the combinators given above only because we need them for the specifications in the next section.

2.3 An Example of Reuse: Specifications of a Bounded Buffer

In this section, we extend the specification of the bounded buffer given in Sect. 2.1 with additional messages, as originally suggested in [MY93] and partly also in [Mes93b]. With these extensions we demonstrate that Maude's inheritance relation, the message algebra and the concept of subconfiguration are powerful enough to overcome the inheritance anomaly. The first extension (by a message `last`) uses only Maude's inheritance mechanism, the second (by a message `get2`) the message algebra, and for the third extension (by a message `gget`) we need the message algebra and subconfigurations. To prove that our reuse constructs are orthogonal, we specify a buffer which accepts `last`, `get2` and `gget`. The class and module hierarchy we present in this section are depicted in Fig. 2.4.

2.3.1 Message: `last`

One extension used in [MY93] to demonstrate how synchronization code requires the redefinition of existing code is the addition of a method `last` which returns the most recent element put into the buffer:

```
module BD-BUFFER-X {
  import {
    protecting (BD-BUFFER)
  }

  signature {
    class XBdBuffer [BdBuffer] {
    }
```

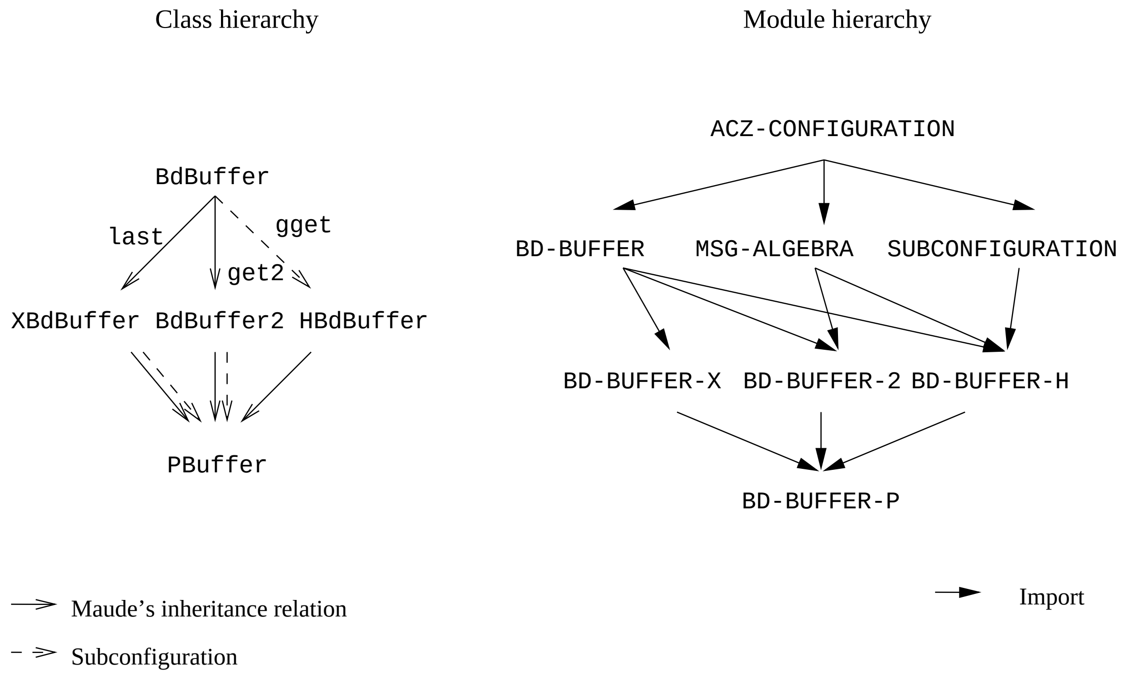

Figure 2.4: Extensions of class `BdBuffer` and module `BD-BUFFER`

```
  op last _ replyto _          : ObjectId ObjectId -> Message
  op to _ answer to last is _ : ObjectId Elem -> Message
  }

  axioms {
    vars B U  : ObjectId
    var  E    : Elem
    var  C    : List
    vars O    : Nat
    vars ATTS : Attributes

  rl [L]: (last B replyto U)
          < B : XBdBuffer | cont = E C, out = O, ATTS >
          => < B : XBdBuffer | cont = C, out = O + 1, ATTS >
             (to U answer to last is E) .
  }
}
```

Since `XBdBuffer` is a subclass of `BdBuffer`, an `XBdBuffer` inherits all attributes and all transition rules from `BdBuffer`. Thus, an object of class `XBdBuffer` is capable of all state transitions which an object of class `BdBuffer` is capable of in the same context. The transition rule `L` defines the behavior of an `XBdBuffer` when accepting a message `last`. When adding this new behavior to an existing specification, we do not have to redefine or alter any piece of existing code and, thus, the inheritance anomaly does not apply.

2.3.2 Message: `get2`

Say, we would like to extend the specification of class `BdBuffer` such that a buffer accepts an additional message, `(get2 B replyto U)`, which sends two elements of buffer `B` to an object `U`. In [Mes93b], such a bounded buffer is specified according to the fragment of a specification given below:

```
module BD-BUFFER-2-PRELIM {
  import {
    extending (BD-BUFFER)
  }

  signature {
    class BdBuffer2 [BdBuffer] {
    }

    op get2 _ replyto _                : ObjectId ObjectId -> Message
    op to _ answer to get2 is _ and _ : ObjectId Elem Elem -> Message
  }

  axioms {
    vars B U  : ObjectId
    vars E F  : Elem
    var  C    : List
    vars I O  : Nat
    var  M    : NzNat
    var  ATTS : Attributes

  rl [get]: (get2 B replyto U)
             < B : BdBuffer2 | cont = C F E, out = O, ATTS >
             => < B : BdBuffer2 | cont = C, out = O + 2, ATTS >
                (to U answer to get2 is F and E) .
  }
}
```

This solution is not "suffering" from the inheritance anomaly but has one drawback: it does not reuse the specification of message `get`. Our solution uses `MSG-ALGEBRA` to derive a composed message `get2` from the implementation of `get`:

```
module BD-BUFFER-2 {
  import {
    protecting (BD-BUFFER)
    protecting (MSG-ALGEBRA)
  }

  signature {
    class BdBuffer2 [BdBuffer] {
    }

    op get2 _ replyto _               : ObjectId ObjectId  -> Message
    op to _ answer to get2 is _ and _ : ObjectId Elem Elem -> Message
  }

  axioms {
    vars B U  : ObjectId
    vars E F  : Elem
    var  ATTS : Attributes

  eq [E1]: (get2 B replyto U)
           < B : BdBuffer2 | ATTS >
           = < B : BdBuffer2 | ATTS >
             ((get B replyto U);;(get B replyto U)) .

  eq [E2]: ((to U answer to get is E);;(to U answer to get is F))
           = (to U answer to get2 is F and E) .
  }
}
```

We allow the “transformation” of a `get2` message to two `get` messages in sequence only if `get2` is addressed to a buffer of class `BdBuffer2` and, thus, we do not extend the set of messages a `BdBuffer` object accepts. The rewriting calculus and the algebra of messages allow us to process a `get2` message in one step, since we have an equational “transformation” of `get2` to `get` messages which does not require a computation step and since the algebra of messages allows us to build an atomic state transition `get2`.

Assume that we would like to implement `get2` in a more conventional language, with methods encapsulated in objects and guards for the methods [AFK+93, Frø92, HSJ+94, MY93]. This implementation of `get2` would apply `get` twice and its guard would have to make sure that `get2` is only invoked if the buffer contains (at least) two elements. The synchronization code of `get2` would have to be either derived from the synchronization code of the `get` method invoked twice by `get2` or written “by hand”. Both might be hard, although, in our example, both is rather trivial. In our approach, the message algebra ensures that a `get2` method may only be invoked if both invocations of `get` can be executed in sequence. This replaces the synchronization code of the message `get2` and facilitates the reuse of methods.

2.3.3 Message: `gget`

Our last modification is to make the bounded buffer history-sensitive: a new message, `gget`, is only accepted if the latest message was a `put` message. Adding a message with a history-sensitive behavior involves a change of the behavior of all messages: `put` has to set a flag, all other messages have to reset it:

```
module BD-BUFFER-H {
  import {
    protecting (BD-BUFFER)
    protecting (MSG-ALGEBRA)
    protecting (SUBCONFIGURATION)
  }

  signature {
    class Flag {
      buffer : ObjectId
    }
    class FSet [Flag] { }
    class FUnset [Flag] { }

    [Flag BdBuffer < Subconfiguration]
    class HBdBuffer {
      conf : Subconfiguration
    }

    op flag-set _            : ObjectId -> Message
    op flag-unset _          : ObjectId -> Message
    op flag-reset _          : ObjectId -> Message
    op gget _ replyto _      : ObjectId ObjectId -> Message
  }

  axioms {
     vars B U F S : ObjectId
     var  C       : Subconfiguration
     var  E       : Elem
     vars B-ATTS F-ATTS H-ATTS : Attributes

  rl [S]: (flag-set F)   < F : Flag | F-ATTS >  => < F : FSet   | F-ATTS > .
  rl [U]: (flag-unset F) < F : Flag | F-ATTS >  => < F : FUnset | F-ATTS > .
```

```
  rl [R]: (flag-reset F) < F : FSet | F-ATTS >  => < F : FUnset | F-ATTS > .


  eq [E1]: (gget B replyto U)
         < B : HBdBuffer | (conf = < B : BdBuffer | B-ATTS >
                                   < F : Flag | buffer = B, F-ATTS >
                                   C),
                           H-ATTS >
          = < B : HBdBuffer | (conf = ((get B replyto U) | (flag-reset F))
                                       < B : BdBuffer | B-ATTS >
                                       < F : Flag | buffer = B, F-ATTS >
                                       C),
                              H-ATTS > .

  eq [E2]: < B : HBdBuffer | (conf = (to U answer to get is E) C), H-ATTS >
        = < B : HBdBuffer | (conf = C), H-ATTS >
            (to U answer to get is E) .

  eq [E3]: (put E into B)
         < B : HBdBuffer | (conf = (< B : BdBuffer | B-ATTS >
                                    < F : Flag | buffer = B, F-ATTS >)
                                    C),
                            H-ATTS  >
       = < B : HBdBuffer | (conf = ((put E into B) | (flag-set F))
                                   < B : BdBuffer | B-ATTS >
                                   < F : Flag | buffer = B, F-ATTS >
                                   C)  ,
                           H-ATTS > .

  eq [E4]: (get B replyto U)
         < B : HBdBuffer | (conf = < B : BdBuffer | B-ATTS >
                                   < F : Flag | buffer = B, F-ATTS >
                                   C),
                           H-ATTS >
        = < B : HBdBuffer | (conf = ((get B replyto U) | (flag-unset F))
                                    < B : BdBuffer | B-ATTS >
                                    < F : Flag | buffer = B, F-ATTS >
                                    C ),
                            H-ATTS > .
  }
}
```

A history-sensitive buffer HBdBuffer encapsulates, in a subconfiguration, a buffer and a flag which indicates whether the last message accepted was a put.

We model the states of the flag, according to the states-as-classes approach, as classes and not, as usual, as attributes. In doing so, we are able to refine the state of class Flag and, thus, its ability to process messages by introducing more subclasses. A flag accepts three messages and, while the state (and class) is irrelevant for a flag-set and flag-unset message to be accepted, a flag-reset message is only accepted if the actual state of the flag is FSet.

The messages addressed to an object HBdBuffer are transformed by equations to a composed message inside the subconfiguration. This composed message consists of one message addressed to the BdBuffer and one message addressed to Flag. The combinator | ensures that the message responsible for the manipulation of the buffer and the message triggering the state change of the flag are processed in parallel. In equation E1 a message gget can be transformed into a parallel combination of a get and a flag-reset message at any time. The state of the flag is only relevant for processing the composed message consisting of a flag-reset and a get message.

The two messages put and get migrate—like the message gget—into a subconfiguration and are transformed into a put, respectively a get message, and a message addressed to the flag. A put message is transformed into a composed message, consisting of a put and a flag-set message, a get into a get and an flag-unset message. The migration of answer messages from a subconfiguration into a configuration is also modeled by an equation.

The variable C of type subconfiguration in the equation matches messages which have already passed from the overall configuration into the subconfiguration. The use of configuration variables also enhances the reusability of the specification BD-BUFFER-H: if in a reusing specification more than two objects are contained in an HBdBuffer then the variable C can match these objects in the application of the transition rules by the rewriting calculus.

2.3.4 Another Buffer

In the previous sections, we have given methods and language constructs for the reuse of specifications. Each method covers a particular situation of reuse. But it remains to demonstrate that these three techniques of reuse fit and can be used together.

We specify a bounded buffer, PBuffer, which is capable of processing all the three messages: last, get2 and gget.

```
module BD-BUFFER-P {
  import {
    protecting (BD-BUFFER-X)
    protecting (BD-BUFFER-2)
    protecting (BD-BUFFER-H)
  }
```

```
signature {
  class PBufferI [BdBuffer2] {
  }
  [PBufferI < XBdBuffer]

  class PBuffer [HBdBuffer] {
  }

 op (_,_)                        : ObjectId ObjectId -> ObjectId

 op new PBuffer with _ replyto _ : NzNat ObjectId    -> Message
 op to _ the new PBuffer is _    : ObjectId ObjectId -> Message
 }

axioms {
  vars B U P F : ObjectId
  vars E E'    : Elem
  var  M       : NzNat
  var  C       : Subconfiguration
  vars ATTS B-ATTS F-ATTS : Attributes


eq [E1]: (last B replyto U)
         < B : PBuffer | (conf = C), ATTS >
       = < B : PBuffer |
                (conf = C ((last B replyto U) | (flag-unset B))), ATTS > .

eq [E2]: < B : PBuffer | (conf = C (to U answer to last is E)), ATTS >
         = < B : PBuffer | (conf = C), ATTS >
          (to U answer to last is E) .

eq [E3]: (get2 B replyto U)
         < B : PBuffer | (conf = C), ATTS >
       = < B : PBuffer |
                (conf = C ((get2 B replyto U) | (flag-unset B))), ATTS > .

eq [E4]: < B : PBuffer |
                (conf = C (to U answer to get2 is E' and E)), ATTS >
         = < B : PBuffer | (conf = C), ATTS >
           (to U answer to get2 is E' and E) .

rl [N]: (new PBuffer with M replyto U)
        < P : Proto | class = PBuffer, next = (B,F) >
```

```
          => < P : Proto | class = PBuffer, next = (incoid(B), incoid(F)) >
             < B : PBuffer | (conf = (< B : PBufferI | cont = eps, in = 0,
                                                       out = 0, max = M >
                                    < F : FUnset | buffer = B >)) >
             (to U the new PBuffer is B) .
  }
}
```

We have two subclass relations which inherit their behavior from ancestor classes. The subclass definition of `PBufferI` ensures that `put`, `get` and `last` can be processed by `PBufferI`. Furthermore, it ensures that a `get2` method can actually be converted into a sequence of two `get` messages. The subclass definition of `PBuffer` ensures that all messages which may migrate into an `HBdBuffer`, namely `put`, `get` and `gget`, may migrate into a `PBuffer` as well. Equations `E1` to `E4` specify the migration of the messages `last` and `get2` and the answer message into and out of the subconfiguration. Rule `N` ensures that the buffer contained in the subconfiguration of `PBuffer` is of class `PBufferI`.

Note that again no changes of the reused specification are necessary. In fact, this specification demonstrates that our three concepts of reuse can be used together.

2.3.5 Buffers with Synchronous Communication

The transition rules of Maude offer a very powerful communication mechanism which we have not used up to now in our model of bounded buffers: a rewrite rule can employ more than one object and one message at its left-hand and right-hand side. Such a rule specifies joint atomic state transitions by all the objects involved. As an example, we use the rule `G` which models a synchronous `get` from specification `SYNC-SRB` in Sect. 2.1.1. To distinguish the implementation of the synchronous `get` message from the asynchronous `get` message, we give it the message name `sync-get` and an additional parameter, the object identifier of the receiver. Both implementations are included in specification `SYNC-SRB-MSG`. Let us give the specification first and discuss the two versions of the implementation of `sync-get`, which it offers, afterwards.

```
module SYNC-SRB-MSG {
  import {
    protecting (NAT)
    protecting (LIST)
    protecting (EXT-ACZ-CONFIGURATION)
    protecting (MSG-ALGEBRA)
    protecting (BD-BUFFER)
   }

  signature {
    class Receiver {
```

```
      incoming : List
      buffer   : ObjectId
    }

    op sync-get _ replyto _ : ObjectId ObjectId -> Message
    op h _                  : ObjectId -> Message
   }

  axioms {
    vars R B U : ObjectId
    var  E     : Elem
    vars C L   : List
    vars I O   : Nat
    var  M     : NzNat
    vars Buffer-ATTS, Receiver-ATTS : Attributes

                                                    -- first implementation
  rl [SYNC-GET-RULE]:
     (sync-get B replyto R)
     < R : Receiver | incoming = L, buffer = B, Receiver-ATTS >
     < B : BdBuffer | cont = C E, out = O, Buffer-ATTS >
     => < B : BdBuffer | cont = C, out = O + 1, Buffer-ATTS >
        < R : Receiver | incoming = E L, buffer = B, Receiver-ATTS > .

                                                    -- second implementation
  rl [H]:
     (h R)
     (to R answer to get is E)
     < R : Receiver | buffer = B, incoming = L, Receiver-ATTS >
     => < R : Receiver | buffer = B, incoming = E L, Receiver-ATTS > .

  eq [SYNC-GET-COMPOSED]:
     (sync-get B replyto R) = ((get B replyto R) ; h(R)) .
  }
}
```

Is a synchronous rule like `SYNC-GET-RULE` really appropriate for reuse? Of course, this depends on the kind of reuse, but synchronous rules cannot be inherited easily when encapsulating one of the participants—say, the buffer—in a subconfiguration. The rule relies very much on the structure of the configuration—namely, that both user and buffer are part of the same configuration—and on the shape of the objects.

We propose a specification which uses the message algebra to specify a joint atomic state transition between buffer and receiver. Transition rule `H` specifies that a user requires two messages, an `answer` message and a help message `h`, to store an element retrieved from the buffer. Equation `SYNC-GET-COMPOSED` specifies that a synchronous `get` is a sequential composition of a `get` and an `h` message. The sequential composition ensures that the `h` message is always processed after the `get` message and, since the `h` and the `answer` message can only be processed in one joint synchronous transition, the `answer` message is also processed.

One can imagine that the other messages of the various buffers can be specified in a synchronous version using this technique. Moreover, this specification of a synchronous communication can also be inherited to heirs which are encapsulated in a subconfiguration.

2.3.6 How to Circumvent the Inheritance Anomaly

In [MY93], several kinds of synchronization code are investigated. The inheritance anomaly occurs because no kind of synchronization code supports all types of modification when reusing a class declaration which contains synchronization code. Maude's equivalent to "synchronization code" is the pattern to be matched at the left-hand side of a transition rule. It consists of one or more messages, their parameters and the internal state of the object(s). Thus, each transition rule specifies a "synchronization constraint", more precisely an "enabled set", i.e., a condition under which a message may be accepted, individually for each message. Since each rule specifies such a pattern, adding new "enabled conditions" by adding new rules is the kind of modification of existing code which can be expressed straight-forwardly by Maude's inheritance relation (like `get2` and `last`).

For other kinds of modification of the behavior of reused code, we provide different mechanisms. With encapsulation in subconfigurations we are able to restrict the ability of objects to react to messages. The algebra of messages supports the specification of complex systems with a large number of objects and a complex control flow in the reuse of specifications with "simple" transition rules.

The reason why we are able to reuse code in many ways is that we provide different kinds of synchronization or control code: for each particular type of modification there is one particular construct for reuse.

One of the advantages of using equations to model the migration of messages into and out of subconfigurations is that we do not add state transitions or actions to the rewrite system. The migration would be some special kind of state transition and could be modeled by an internal action (like τ in CCS), but this would cause the same problems with compositionality and verification as it does in CCS.

Maude's inheritance relation together with the concept of a subconfiguration and an algebra of messages make it possible to reuse specifications. Our work on a verification technique for Maude specifications [WNL95] and Chap. 4 also demonstrate that asynchronous message passing is better than synchronous message passing with respect to the inheritability of properties of specifications, a necessity for modular verification. The use of equations in modeling the migration of messages into and out of configurations keeps the number of transitions small. This helps to make the verification of properties of the behavior of single objects or configurations feasible. The use of equations contributes to a very abstract level of specification, where the structure of the state is of less importance, the focus of the specification lies on synchronization and communication of the objects, and one has few but powerful messages.

The degree of reusability of our specifications is far higher than one would expect for such a simple language in the presence of the inheritance anomaly. This suggests that the basic design concepts of Maude, especially the object model, the communication mechanism and the transition rules for the specification of the behavior, are more appropriate for a structured design of specifications than the design concepts of more conventional languages with synchronous communication, explicit synchronization code and code of methods encapsulated in objects.

Together with our work on verification techniques for specifications in Maude [WNL95], our picture of a sensible object-oriented specification language for the design of complex, concurrent systems is becoming more and more precise: Maude's object model, Maude's communication mechanism, a message algebra, subconfigurations and, maybe, also a module concept for programming in the large. In our approach we use a specification language, Maude. But, of course, our reuse mechanisms can be modeled by language constructs and become part of an object-oriented concurrent specification or programming language.

2.4 How to Specify in Maude

To cope with a large specification or program, one needs not only concepts for structuring and reuse, but also a programming method appropriate for the language and the concepts. We have developed a powerful set of structuring and reuse constructs for Maude: states as classes, inheritance, subconfigurations, message algebras and modules. In this section we develop some guidelines for how to apply these constructs in the development of a large specification.

The object-oriented paradigm comprises object-oriented analysis and design methods [Boo91, RBP+91, BR95, JCJÖ92, SM92]. They provide programmers with guidelines and informal or semi-formal methods to describe the program and its properties. Naturally, the choice of the programming language has to have some influence on the design method. These methods aim at a quite conventional object model and a sequential programming language. Thus any of the analysis and design methods has to be adapted to Maude and in particular to concurrency, the asynchronous explicit communication by messages, the synchronous implicit communication, the way state transitions are modeled and the structuring and reuse concepts. Particularly the way the computational progress is specified in Maude, the transition rules, demand a new programming method. In traditional object-oriented languages, the methods are encapsulated in classes and, thus, the classes are the main means to structure the code of a specification. In Maude, (synchronous) transition rules cannot be assigned to and encapsulated inside a class. This stresses the need for a programming method. However, our goal is to demonstrate how our structuring and reuse concepts are applied in the structured development of a large system and to develop some guidelines and not to develop a new analysis and design method.

As an example we use the specification of an airport. Our specification is based on [Sal93], where several versions were developed. Each of these versions has more than 3000 lines of code. We present here a fragment of the specification of an airport to illustrate the design concepts of our specification. We give additional parts of this specification in App. A.2. However this specification still is not a complete model of an airport, since we provide only the specifications necessary to interpret the modules given in this section.

In this section we start at a point in the design process where we have already identified which entities are modeled by objects and with which messages these objects communicate. In our example, we have objects modeling planes, passengers, cargo, crew members, clerks, tower, runway, take-off and landing controls, ground controls.

We focus in this section on the use of the structuring and reuse concepts. We proceed as follows. We begin with structuring in the small, namely with the structuring of a class by states as classes, and continue with the structuring concepts at the level of classes, inheritance and subconfigurations. We discuss the role of composed messages and asynchronous, synchronous and composed transition rules in a structured specification. Finally, we continue with reuse in the large of modules.

2.4.1 How to Use States as Classes

In a concurrent setting, the specification of the behavior of a single class is typically more complicated than in a sequential setting. The reason for this is the synchronization code, the code that determines whether an object reacts to a message. Although, up to now, all examples of Maude specifications are quite small, they demonstrate already the need for structuring concepts. The analysis of the inheritance anomaly in Sect. 2.3 shows the difficulties in organizing, implementing and reusing this synchronization code, and it demonstrates that there is a strong need for modularization even at a granularity smaller than a class.

The specification of the behavior of a plane is an example of a class which has several states and which is involved in so many different transition rules that modularization is important in order to keep the specification readable. The life cycle of a plane consists of several states: `onGround`, `takeoff`, `flying`, `landing` and, again, `onGround`, where state `onGround` consists itself of four states: `disembarking`, `maintenance`, `waiting`, `embarking`. The life cycle is depicted in Fig. 2.5.

Specification `PLANES` gives part of the class declarations of planes and their states:

```
module PLANES {
  import {
    protecting (ACZ-CONFIGURATION)
    protecting (NAT)
    protecting (LIST)
    protecting (TIME)
    protecting (SUBCONFIGURATION)
  }
```

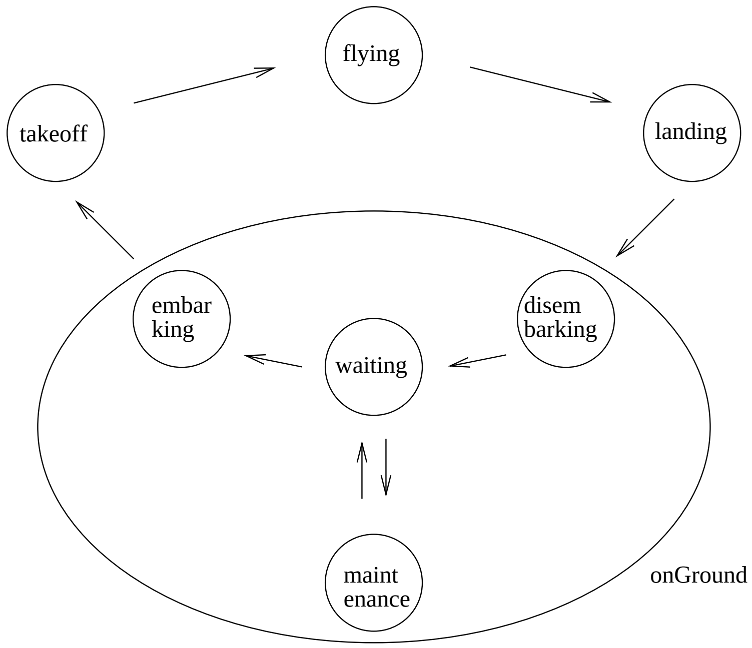

Figure 2.5: States of a plane

```
signature {
  [StateofPlane < Subconfiguration]
  [Cockpit Engine < StateofPlane]

  class Plane {
    destination  : ObjectId
    schedule     : Time
    state        : StateofPlane
    tower        : ObjectId
  }

  class FlyingPl [Plane] {
    speed  : Nat
    height : Nat
  }

  class onGroundPl [Plane]{
  }

  class TakeoffPl [onGroundPl] {
    runway : ObjectId
```

```
    }
  }
}
```

In each of the states, a plane reacts differently to messages, and also has different attributes. There is a couple of attributes common to all these states; they are declared for class `Plane`. E.g., common to all planes is that they store the destination and their schedule. In contrast a state in class `TakeoffPl` stores information different to class and state `Flying`. Specification `SYNCHRONOUS-TAKEOFF` gives a transition rule specifying the takeoff of a plane with a change of the state of a plane:

```
module SYNCHRONOUS-TAKEOFF {
  import {
    protecting (PLANES)
    protecting (PARTS-OF-AIRPORTS)
    protecting (LIST-ENRICHED)
  }

  signature {
    op  minspeed : -> Nat
    op  takeoff of _ at _ with _ : ObjectId ObjectId ObjectId -> Message
    op (_,_) :  ObjectId ObjectId -> Elem
  }

  axioms {
    vars P R T S : ObjectId
    vars FL NL   : List
    vars Plane-ATTS Runway-ATTS Tower-ATTS : Attributes

  crl [takeoff]:
        ((takeoff of P at R with T)
        (< P : TakeoffPl | runway = R,
                           destination = S, tower = T, Plane-ATTS >
        (< R : Runway | Runway-ATTS >
        < T : Tower | free = FL, nfree = (R,P) NL, Tower-ATTS >)))
        =>  (< P : FlyingPl | speed = minspeed,
                              destination = S, tower = T, Plane-ATTS >
            (< R : Runway | Runway-ATTS >
             < T : Tower | free = FL R, nfree = NL, Tower-ATTS >))
          if not(iselem(R,F)) .
  }
}
```

The transition rule for take-off models the change of the plane's state by a change of the plane's class. The attributes that are new in the class at the right-hand side of the rule have to be given values. The attributes that are particular to the class at the left-hand side of a rule do not exist at the right-hand side any more.

The rule `Takeoff` models the take-off of a plane. It is a synchronous action, involving the tower, the runways and a plane. The plane changes its state. In the transition from a plane in class `TakeoffPl` to `FlyingPl`, the attribute `runway` is only part of state `TakeoffPl` while the attributes `speed` and `direction` are set at the right-hand side of the rule.

The states-as-classes approach has several advantages. The first, in this example obvious advantage is that the specification becomes more modular. In each state, only the necessary attributes are present and, thus, the size of the state does not become unnecessarily large. The second advantage is that the state changes are modeled not only smoothly but very efficiently. In typical implementations, an object contains a link to its code. When an object changes its state, only the link to the code has to be moved to implement the change of the behavior.

To explain the third advantage, we have to preview material of Chap. 3. Methods are operations on data types that are defined for a class and all its subclasses. Only inherited methods are defined for the ancestor class(es) as well. Thus, methods are partial operations. Order-sorted algebras avoid partiality: in object-oriented sequential languages, the functionality of the ancestor class is a subset of the functionality of the heir classes and the carrier set for the ancestor is a subset of the carrier set of the heir. In object-oriented concurrent languages, the behavior of an object, i.e., the messages it accepts and the methods it executes, varies much more than in sequential languages, since the objects are more autonomous. I.e., a plane in state Takeoff must provide operations which are not needed in the other states and, thus, should not be implemented for all states, and a plane in state take-off accepts messages it refuses in other states and vice versa. If a plane belonged to only one class during its lifetime, this would yield highly partial operations on the state. Moreover, since the synchronization code prevents messages from being accepted, there is no need for implementing all operations for all states. The operations on the data must only be implemented, if the message that triggers these operations to be executed is accepted. In order to keep a specification small and comprehensible, one should not implement operations for states in which they will—due to the synchronization code—never be executed. This yields, compared to sequential object-oriented languages, a higher amount of partiality of operations. To avoid the problems that arise from partial functions, we take advantage of the fact that the number of states is reasonably small, and thus we can model them in classes. If whether a message is accepted or not depends only on the message name and the object identifier the message is addressed to, then the number of states is maximally 2^n, where n is the (typically small) number of messages or methods of an object. With the states-as-classes concept we obtain total functions again. This gives us guidelines for how to choose the states as classes and how to implement the states:

- *Choose the classes such that the operations on the state are total, i.e., the operations* *necessary to perform the state change are defined when an object accepts a message.*

- Provided that whether an object accepts a message depends on the message name, *choose the states as classes such that the state and class determines alone whether an object accepts a message or not. The values of the attributes determine how the object reacts.*

- *Keep the state of objects as small as possible and avoid unnecessary attributes. Model states, which require different attributes, as classes.*

2.4.2 How to Use Inheritance

Inheritance is the most popular form of reuse in object-oriented languages. Maude's inheritance relation prescribes the inheritance of all attributes, all equations and all transition rules of the ancestor to the heir. Thus, in order to develop a class hierarchy, one has to find components of the states, i.e., attributes, equalities and state changes that are common to several classes.

The states-as-classes approach is a method for modularizing the specification according to the states of an object. But the states-as-classes approach is not the only way in which classes can be modularized. A class can comprise several specializations, which differ in their state and in their behavior. For instance, a plane can be either a passenger plane or a cargo plane. Both planes have different states: one has to have a cabin with passengers and a small cargo hold, one has to have only a large cargo hold. In the example of the airport, states as classes and specializations are orthogonal: the instances of planes belong to classes that inherit from one of the two specializations and from one of the states. The class hierarchy used to model the behavior of planes is depicted in Fig. 2.6. At the lowest level we refrain from giving all possible subclasses.

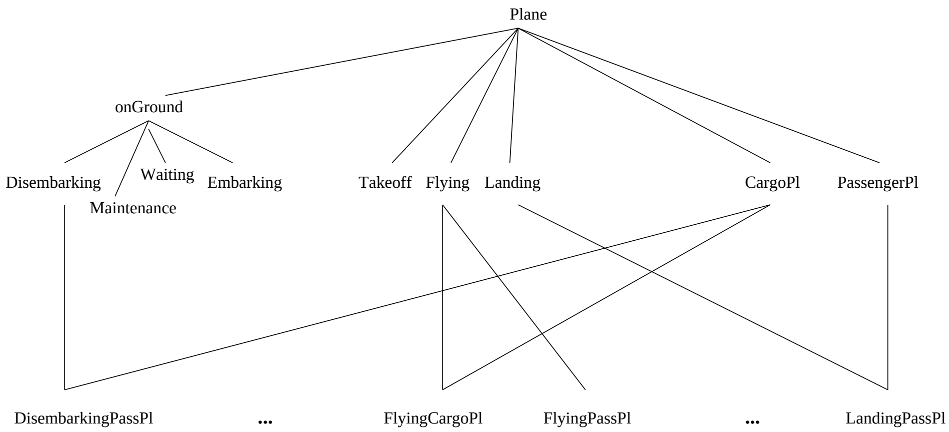

Figure 2.6: Class hierarchy for modeling planes

In our notion of inheritance, heirs inherit the whole state and all transition rules from ancestors. Thus, to establish a class hierarchy based on inheritance, one has to find components of the state, equations and transitions that are common to several classes.

We summarize the important issues in using inheritance in three rules:

- *Try to separate states as classes and specializations in the class hierarchy.*
- *Try to find common components of the states, common equalities and common transitions to identify superclasses.*
- *Do not make the class hierarchy deep, each class has to model a reasonable unit.*

2.4.3 How to Use Subconfigurations

In Sect. 2.2.3 subconfigurations were used as an instrument to control which messages come into contact with an object being encapsulated.

In general, subconfigurations restrict the ability of objects to react to messages that are part of the global state. In reuse subconfigurations are therefore the dual concept to Maude's inheritance. Thus, in a class hierarchy, we have after inheritance as the first reuse relation a second relation, the subconfiguration.

Let us illustrate its use in the airport specification, and in particular in the class hierarchy modeling planes. A plane might be faulty. Say, a defect destroys the ability to react to messages. With the inheritance relation we have gradually built up the ability to react to messages. With a subconfiguration relation that is dual to the inheritance relation, we reduce the ability to react to messages. Say, the defects are relevant only in the classes modeling states and independent from the specialization. Thus each class modeling a state has a dual state modeling a faulty plane in this state. We obtain a class hierarchy, parallel to the inheritance relation. We use subconfigurations to derive from the states as classes the states modeling faulty planes. Between the classes modeling faulty planes there is again an inheritance relation. Fig. 2.7 depicts the part of the class hierarchy modeling states as classes with the parallel class hierarchy modeling the faulty planes. Specification `FAULTY-PLANES` gives the implementation of part of the class hierarchy of faulty planes:

```
module FAULTY-PLANES {
  import {
    protecting (PLANES)
    protecting (SUBCONFIGURATION)
    protecting (PARTS-OF-AIRPORTS)
  }

  signature {
    [onGroundPl < Subcf-FaultyonGround < Subconfiguration]
    class FaultyonGround {
      plane : Subcf-FaultyonGround
    }
```

```
    [DisembarkingPl < Subcf-FaultyDisembarking < Subconfiguration]
    class FaultyDisembarking [FaultyonGround] {
    }

    [Subcf-FaultyDisembarkingPassPl <  Subcf-FaultyDisembarking]
    class FaultyDisembarkingPassPl [FaultyDisembarking] {
    }

    [Subcf-FaultyDisembarkingCargoPl < Subcf-FaultyDisembarking]
    class FaultyDisembarkingCargoPl [FaultyDisembarking] {
    }
  }
}
```

Let us explain the class hierarchy in detail. Class `FaultyonGround` is inferred from class `onGround`. An object of class `FaultyonGround` is not able to react to all messages an object of class `onGround` is able to react to. Technically this is realized by encapsulating an object of class `onGround` in a subconfiguration in attribute `plane` of class `FaultyonGround`.

A plane in any of the four subclasses of `onGround` suffers from the defects as well. Thus, for every of the four subclasses, we introduce a corresponding class modeling faulty planes. E.g., for the class `Disembarking`, we introduce the class `FaultyDisembarking`. The inheritance relation between `onGround` and its four heir classes is mirrored in the class hierarchy of the classes `FaultyonGround` and its four subclasses. Thus, a faulty class, e.g., `FaultyDisembarking` can perform at least all transitions of its superclass, `FaultyonGround`, while `FaultyonGround` reduces the ability to react to messages `onGround` reacts to and `FaultyDisembarking` reduces the ability to react to messages `Disembarking` reacts to.

This construction is iterated to obtain the inheritance and subconfiguration relations at the next level of the class hierarchy. In Fig. 2.7 we refrain from depicting all these subclass relations and restrict ourselves to picture the relation for disembarking planes.

Subconfigurations can play a second role as a means to structure the global state: they help to provide a more comprehensible and more faithful specification. The state of a system modeled in Maude is an unordered collection of objects and messages. This often does not match the human intuition about the system modeled. In the “real world”, there are clusters of communicating entities. Communication between these clusters happens less often than communication inside the clusters. Moreover, the clusters are really separate from each other and do not interfere whereas, in a Maude specification, all objects that are part of a configuration can communicate with each other. Thus, subconfigurations model that a group of objects communicate only via certain messages with its environment and the other way round, that the environment cannot interfere with the communication of such a group of objects.

Let us continue with the airport specification. Up to now, we have only considered the

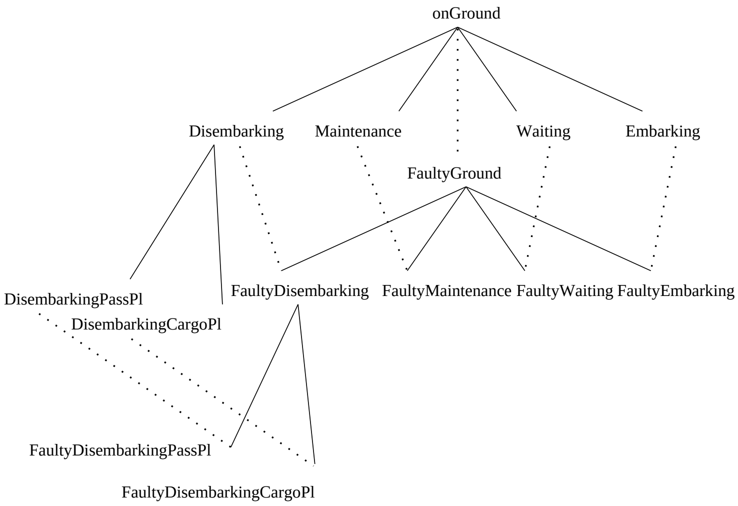

Figure 2.7: Class and subclass hierarchy for modeling planes

class hierarchy of planes. To demonstrate at which points we use subconfigurations we give a larger portion of the class hierarchy of the airport specification.

In the airport specification, each airport is a cluster of objects. The objects being part of an airport communicate with each other. Only very few messages are addressed to entities not part of an airport, and only very few components of an airport have the ability to send messages out of an airport. On the other hand, also only relatively few messages from outside the airport have the ability to influence parts of an airport.

Thus, each airport "contains" in a subconfiguration all its institutions and the planes and the passengers. Subconfigurations can be nested and, thus, several objects in a subconfigurations can, again, contain subconfigurations. E.g., each plane is itself a cluster which contains in several subconfigurations the cargo, the passengers, the crew, the engine.

Airports are not the only "top-level" subconfigurations. For instance, we use subconfigurations to model the air corridors along which planes fly between the airports. An airport itself contains several subconfigurations: planes comprise a cockpit and an engine. A cockpit contains a pilot, a copilot and a display. A passenger plane transports a cabin, which consists of passengers and crew members, and a cargo hold, while a cargo plane transports a cargo hold alone:

```
module PLANE-COCKPIT {
  import {
    extending (PLANES)
    extending (PARTS-OF-AIRPORTS)
  }

  signature {
```

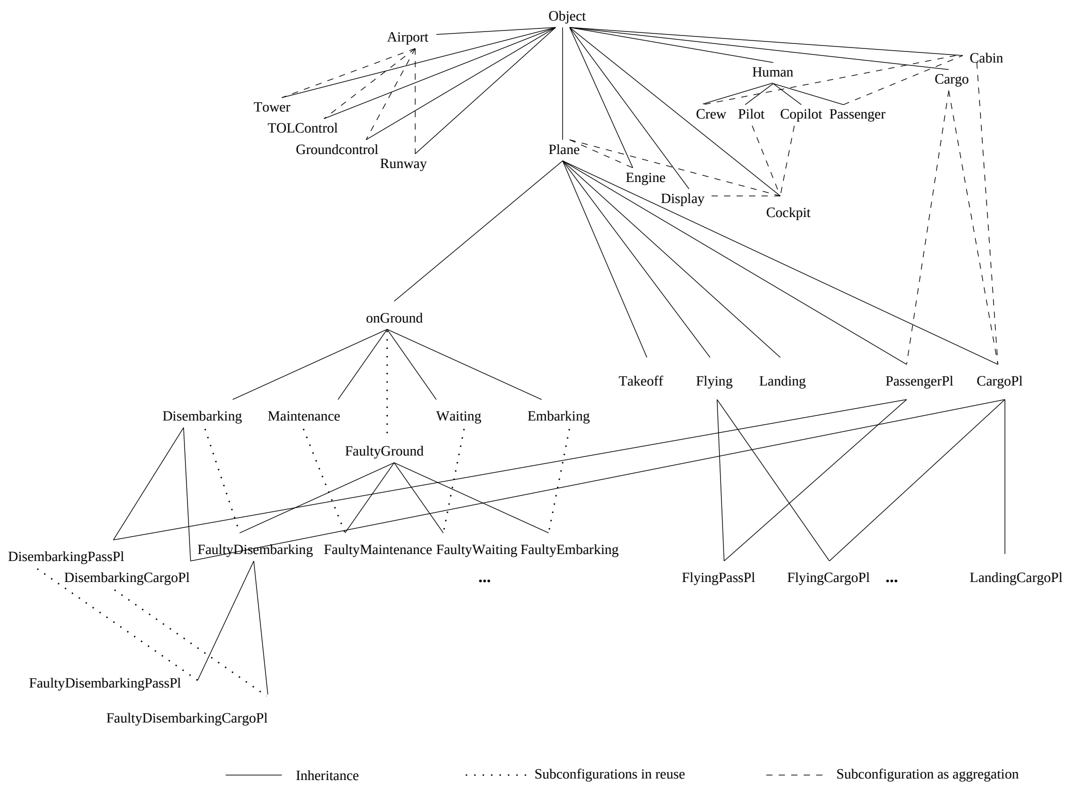

Figure 2.8: Class hierarchy for modeling an airport

```
    [Subcf-Cockpit]
    class Cockpit {
      state : Subcf-Cockpit
    }

    [Subcf-PassengerPl]
    class PassengerPl [Plane] {
      transporting : Subcf-PassengerPl
    }

    [Subcf-Cabin]
    class Cabin {
      cont : Subcf-Cabin
    }

    [Subcf-CargoPl]
    class CargoPl [Plane] {
      transporting : Subcf-CargoPl
    }

    class Display {
      speed : Nat
    }

    class Pilot {}

    [Pilot Copilot Display Message  < Subcf-Cockpit < Subconfiguration]
    [Cabin Cargo Message <  Subcf-PassengerPl < Subconfiguration]
    [Passenger Crew Message <  Subcf-Cabin < Subconfiguration]
    [Cargo Message < Subcf-CargoPl < Subconfiguration]
    }
  }
}
```

The use of subconfigurations not only helps to get a more faithful and comprehensible model. An additional benefit of using subconfigurations is that the object models and the transition rules actually become simpler.

After describing the class hierarchy in our airport example, let us discuss the impact of the use of subconfigurations on transition rules. The use of subconfigurations has assets and drawbacks. The main drawback is that subconfigurations induce transitions or equations of rules to model the migration of messages and objects into and out of the subconfigurations. The asset of the use of subconfigurations is that the object model and the patterns used in the transition rules become smaller and simpler, since subconfigurations are a shield against an environment interfering with the communication of a group of objects. This makes the specification actually easier, since the patterns of the left-hand might become less complicated.

Let us compare two different implementations of a situation in a cockpit of a plane. A cockpit contains two people, the pilot and the copilot, each modeled by an object. A cockpit contains an object `Display`; each device of the display is modeled by an attribute of the object `Display`. The pilot asks the copilot to tell him a value of a display. To answer this request the copilot has to look at the value of the display. This is implemented as a synchronous implicit communication between copilot and display. The copilot memorizes the speed in his attribute `speed`.

In the first implementation, which does not take advantage of subconfigurations, i.e., when pilot, copilot and the display are just "floating around" in a configuration, the pattern at the left-hand side of the rule has to ensure that pilot, copilot and display belong together, e.g., to the same plane. Thus, the copilot has to store the object identifier of its pilot and the display in order to be able to sent him messages. The messages have to carry the identifier of the objects they are addressed to:

```
module MESSAGE-IN-GLOBAL-STATE {
  import {
    protecting (PLANE-COCKPIT)
  }

  signature {
    op _ to copilot _ what speed : ObjectId ObjectId -> Message
    op to pilot _ _              : ObjectId Nat -> Message
  }

  axioms {
    vars P1 P2 D : ObjectId
    var  X       : Nat
    vars Display-ATTS Copilot-ATTS : Attributes

  rl  [what-speed]
      (P1 to copilot P2 what speed)
      < P2 : Copilot | display = D, pilot = P1, Copilot-ATTS >
      < D : Display | speed = X, Display-ATTS >
      => < P2 : Copilot | display = D, pilot = P1, Copilot-ATTS >
         < D : Display | speed = X, Display-ATTS >
         (to pilot P1 X) .
  }
}
```

In a specification modeling the cockpit as a subconfiguration, the objects modeling copilot and display and the messages do not have to store or carry the object identifiers to whom they are addressed; there is only one object that may react to a `to copilot` message in the subconfiguration and `to copilot` messages do not migrate into or out of subconfiguration `Cockpit`. Thus, the transition can be implemented in a simpler transition rule:

```
module MESSAGE-IN-COCKPIT {
  import {
    protecting (PLANE-COCKPIT)
  }

  signature {
    op to copilot what speed : -> Message
    op to pilot _ : Nat -> Message
  }

  axioms {
    vars P1 P2 D : ObjectId
    vars X Y     : Nat
    vars Display-ATTS Copilot-ATTS : Attributes

  rl [what-speed]:
     (to copilot what speed)
     < P2 : Copilot | speed = Y, Copilot-ATTS >
     < D : Display | speed = X, Display-ATTS >
     => < P2 : Copilot | speed = X, Copilot-ATTS >
        < D : Display | speed = X, Display-ATTS >
        (to pilot X) .
  }
}
```

The cockpit of a plane is an example of a subconfiguration which is quite static. Subconfigurations are dynamic: they may exist only temporarily and objects as well as messages may migrate into and out of them. An example of such a migration is the flight of a plane from one airport to its destination airport:

```
module FLIGHT {
  import {
    protecting (PLANES)
    protecting (PARTS-OF-AIRPORTS)
    protecting (MSG-ALGEBRA)
    protecting (SYNCHRONOUS-TAKEOFF)
  }
```

```
 signature {
   [ Plane Tower Runway < Subcf-Airport < Subconfiguration ]
   class Airport {
     state : Subcf-Airport
   }

   class Corridor {
     dir    : ObjectId
     planes : Subconfiguration
   }

   op cor : ObjectId ObjectId -> ObjectId    -- corridor from _  to _
   op por : ObjectId ObjectId -> Elem        -- plane on runway

   op flight of _ from _ to _ : ObjectId ObjectId ObjectId -> Message
   op _ flying from _ to _    : ObjectId ObjectId ObjectId -> Message
   op _ landing at _          : ObjectId ObjectId -> Message
 }

 axioms {
   vars P T R MS : ObjectId
   vars F RL     : List
   var  PL       : Subconfiguration
   var  C        : Subcf-Airport
   vars A1 A2 MUC SFA : ObjectId
   vars Tower-ATTS Airp-ATTS Corr-ATTS Plane-ATTS Runway-ATTS : Attributes


eq [flight]:
   (flight of P from A1 to A2)
   < MUC : Airport |
          (state = < T : Tower | nfree = (por(R,P) RL):List,
                                 Tower-ATTS >
                   C),
          Airp-ATTS >
   = ((takeoff of P at MUC with R) ;
     ((P flying from A1 to A2) ; (P landing at A2)))
     < MUC : Airport |
            (state = < T : Tower | nfree = (por(R,P) RL):List,
                                   Tower-ATTS >
                     C),
            Airp-ATTS > .
```

```
  rl [takeoff]:
     (takeoff of P at MUC with R)
     (< MUC : Airport |
             state = < P : TakeoffPl | destination = SFA,
                                       Plane-ATTS >
                    (< T : Tower | free = F,
                                   nfree = por(R,P) RL,
                                   Tower-ATTS >
                     < R : Runway | Runway-ATTS >),
             Airp-ATTS >
      < MS : Corridor | (dir = cor(MUC,SFA)),
                        planes = PL,
                        Corr-ATTS >)
  => < MUC : Airport |
             state = < P : TakeoffPl | destination = SFA,
                                       Plane-ATTS >
                    (< T : Tower | free = F R,
                                   nfree = RL,
                                   Tower-ATTS >
                     < R : Runway | Runway-ATTS >),
             Airp-ATTS >
     < MS : Corridor | dir = cor(MUC,SFA),
                       planes = PL < P : FlyingPl | speed = minspeed,
                                                    destination = SFA,
                                                    Plane-ATTS >,
                       Corr-ATTS > .
  }
}
```

A plane takes off at airport `MUC`, it changes its state from `TakeoffPl` to `FlyingPl` and it leaves subconfiguration `MUC` and moves into subconfiguration `(MUC,SFA)` which models the corridor.

Subconfigurations play also an important role in parallel composition. The asynchronous message passing of Maude, the modeling of a state as a subconfiguration and Maude's transition rules with their flexible patterns give rise to a possibly vast number of "new" transitions, i.e., transitions that are not possible in parts of the configuration, when configurations are composed by multiset union. Subconfigurations restrict the possibilities of the encapsulated objects to react and, thus, reduce the number of "new" transitions when composing complex systems.

In Sect. 4.2 we introduce configuration invariants, which are predicates on configurations that provide control over the number and kind of objects inside a subconfiguration. Such invariants specify properties like the presence or absence of messages and objects, the presence or absence of certain patterns. Invariants for subconfigurations are typically quite simple, but they may be quite complicated for whole configurations. Thus, subconfigurations simplify reasoning about properties of Maude specifications.

The subconfiguration is a powerful concept for modeling object-oriented systems. With rules in Maude, we can model that both objects and messages may migrate into and out of configurations. Since one has to specify this migration explicitly, it is natural that messages and objects can change while migrating, as, e.g., in Sect. 2.3.3 message `gget`. Our concept of a subconfiguration is even more flexible: subconfigurations may be created temporarily and the objects contained in a subconfiguration may vary during its lifetime. To model the migration of objects and messages, both equations on configurations, as in Sect. 2.2.3, and transition rules can be used. The structuring process can be iterated: subconfigurations may contain objects which again contain subconfigurations.

Thus the subconfiguration is a language construct for the reuse of classes and a means to structure global states. We summarize the use of subconfigurations in three rules:

- *Use subconfigurations to derive a class that accepts fewer messages than its ancestor.*
- *Identify clusters of communicating objects and shield them in subconfigurations.*
- *When possible, use equations to model the migration of objects and messages into and out of subconfigurations.*

2.4.4 How to Use Message Combinators

In the previous sections, we have dealt with structuring the class hierarchy. For readability, for verification and for refinement of a specification, it is certainly helpful to keep the number of transition rules as small as possible.

The first way to accomplish this is the use of synchronous transition rules. They specify possibly complex synchronization and communication patterns between a number of objects and messages in one rule. A task involving several complex synchronizations and several communications would involve a number of asynchronous transition rules, and figuring out how these rules work together and whether the rest of the specification might interfere is difficult and very complex. It might also be hard to find out whether states that should be reachable are not and vice versa.

Thus, synchronous rules help to keep the number of transition rules small and provide abstraction from the implementation of the synchronization and communication between objects. Synchronous transition rules make specifications more comprehensible as well as “safer”, since they abstract from the error-prone synchronization and communication typically implemented as a number of rendezvous communications.

But, on the other hand, synchronous transition rules are often quite specialized, i.e., a transition rule might cover just one particular case. This might again lead to a high number of transition rules, which might implement certain parts of the communication and synchronization several times. This is certainly not a desirable situation.

Message combinators, as introduced in specification `MSG-ALGEBRA` in Sect. 2.2.4, allow to compose messages and transitions which are triggered by composed messages. This gives rise to a new way of abstraction: instead of implementing all actions in transition rules, we have a basic set of messages (or methods) and build, with message combinators, transition rules that implement complex tasks. Note that we are basically free to specify any message combinator. The message algebra in Sect. 2.2.4 is an example of a customized set of message combinators. Within such a message algebra, the operators of the π-calculus [MPW92, ACS96, Vir96] can be specified as well.

Note that, from our point of view, combinators in the message algebra do not replace the synchronous transition rules, they just complement them. Synchronous transition rules are still the first choice for specifying tasks involving complex synchronizations and lots of exchanges of values to compute the final state of a transition. This level of abstraction cannot be provided by the operators of a message algebra.

Let us have a look at the airport example again. A flight consists of three phases: the take-off, the actual flight and the landing. One is interested in the properties of the individual phases as well as the properties of the whole flight. In the specifications of the phases, one is interested in abstracting from the possibly complex communication and synchronization protocols between an object and the institutions of an airport. Thus, synchronous transition rules are used. Instead of specifying the whole flight in a transition rule, the flight is modeled as a composition of the three phases. Thus, the specification provides the individual phases of the flight as well as abstraction from the phases without specifying issues twice:

```
eq [flight]:
   (flight of P from A1 to A2)
   < MUC : Airport |
           (state = < T : Tower | nfree = (por(R,P) RL):List,
                                  Tower-ATTS >
                    C),
           Airp-ATTS >
   = ((takeoff of P at MUC with R) ;
     ((P flying from A1 to A2) ; (P landing at A2)))
     < MUC : Airport |
             (state = < T : Tower | nfree = (por(R,P) RL):List,
                                    Tower-ATTS >
                      C),
             Airp-ATTS > .
```

This rule is part of specification `FLIGHT`. Let us examine it this rule a little bit more closely. The message `(flight of P from A1 to A2)` is equal to a (sequential) composition of three messages. The first of these three messages needs an additional parameter, `R`. In order to ensure that this parameter is instantiated properly, we have to have "a look at the corresponding value" in the object modeling the airport. Thus, we need the airport as part of the left- and right-hand side of the rule.

An example of the take-off specified as a single synchronous transition rule can be found in Sect. 5.3.

The concept of having a number of basic methods to retrieve information on the state of an object and to change the state of objects and compose them to complex tasks outside objects suits the object-oriented paradigm. Moreover, it suits the conventional sequential notion of an object, in which complex tasks involving several objects are triggered and controlled by one object. The transaction concept of autonomous objects provides transaction objects, which ensure that their task is executed in a consistent way, i.e., inference-free from other transactions [KLMW94, Kri97].

A relatively small number of transition rules is not the only merit of a specification style in which mostly basic operations are implemented in transition rules and complex tasks are implemented by the message algebra. A second advantage is the additional abstraction. The state of the objects becomes less important, because the transition rules observe and modify the state as in the coalgebraic approach. Complex tasks are specified as combinations of such simple accesses.

Finally we would like to summarize how to structure the implementations of communication actions:

- *Identify basic access and update operations—the methods—and implement them by asynchronous transition rules.*
- *Implement state changes involving complex synchronizations or computations of final states by synchronous rules.*
- *Implement complex tasks by message combinators.*

2.4.5 How to Use Modules

In object-oriented languages, the class is the main concept for structuring programs or specifications. Particular to Maude as an object-oriented language is the module as a structuring concept. Meseguer even uses the term “module inheritance” for the reuse of modules [Mes93a, Mes93b].

There is a simple, pragmatic reason why Maude needs a second reuse concept to complement the structuring concept at the object level: synchronous transition rules involve several objects belonging to several classes. Thus, the transition rules cannot be encapsulated in the classes like methods in more traditional languages. Furthermore, in the language we have non-object-oriented sorts and equations, which cannot be assigned to classes, too. Modules group class declarations, sort and function declarations as well as equations and transition rules. A well-designed module structure enables local reasoning, i.e., to compute a task using only few modules of the complete specification.

The use of modules in the specification of data types is well understood [Wir87]. But the object-oriented part of Maude makes it very difficult to divide a specification into modules. The case studies of the specification of the airport [Sal93], graph algorithms [Har95] and a steam boiler [ÖKW96] revealed that it is hardly possible to divide a specification in a sensible way. There are too many criteria which can be used to divide a specification, but none of them works in a general setting, to facilitate local reasoning.

Meseguer uses modules as a reuse construct dual to inheritance. The module is a construct for programming in the large and provides operations like renaming or hiding for the reuse of whole specifications. We are interested in reuse constructs for the reuse of classes, as it is common in object orientation and, thus, refrain from using constructs for programming in the large.

We cannot offer general hints on how to use modules. We suggest that a programming environment should support a programmer on this issue.

2.5 Specification of Single Classes

Maude's transition rules specify synchronization, communication and computational progress of a collection of objects and messages. In a transition rule, the state of the objects is visible. From an object-oriented point of view, this might be considered a bad and not object-oriented style, since it contradicts the principle of encapsulation. But, on the other hand, for us it is a very convenient specification style, since synchronization and communication can be described at a very abstract level. Moreover, Maude abstracts from the methods. In our object model, we have serialized objects, i.e., an object takes part in only one transition and, thus, we have no problems of concurrency inside objects. But, in fact, we do not specify how the state change is implemented, e.g., whether the computation is done sequentially or concurrently and how the objects taking part in a synchronization exchange their values. We distinguish messages which are part of a configuration and methods which are encapsulated inside objects. We complement now specifications in Maude with two alternative specification notations, an algebraic specification similar to OS [Bre91] and a coalgebraic specification similar to [Jac96c, Rei95].

- The algebraic specification is at the intra-object level. An algebraic specification gives the properties of the data modeling the state of an object. At this level nondeterminism and concurrency are not relevant, since we deal with serialized objects.

- The coalgebraic specification is the link between the intra-object level and the inter-object level, which is covered by Maude. The state and the state changes are not described as in the algebraic view, they are observed via the methods.

Algebraically, a Maude specification is interpreted as description of the construction of the objects and their state. Coalgebraically, it is interpreted as a description of the observations of objects. The differences are conceptual and the relation between these two interpretations is described in more detail in Sect. 3.6.

2.5.1 Algebraic Specification

An algebraic specification is a property-oriented description of abstract data types [Wir90]. The properties are given by equations between data types. Typically, algebraic specifications are deterministic and not state-based, i.e., they do use the concepts of state, state changes and time for the description of the state. We can employ the algebraic specification style for the specification of the data and the methods, since we consider only serialized objects, i.e., since we do not have to deal with nondeterminism and concurrency at the intra-object level.

Let us give an algebraic specification of the data modeling the state of an object and the methods operating on these data.

```
module ALG-BD-BUFFER  {
   import {
     protecting (LIST-ENRICHED)
   }

   signature {
     [BdBuffer]

     [NFBdBuffer < BdBuffer]                  --      Non-Full BdBuffer
     [NEBdBuffer < BdBuffer]                  --     Non-Empty BdBuffer

     [FullBdBuffer < NEBdBuffer]              --          Full  BdBuffer
     [EmptyBdBuffer < NFBdBuffer]             --          Empty BdBuffer

     [NENFBdBuffer < NEBdBuffer NFBdBuffer]  -- Neither Empty nor Full

                                              --    State of a BdBuffer
     op empty  : NzNat Nat Nat   -> EmptyBdBuffer
     op state  : Elem NFBdBuffer -> NENFBdBuffer
     op full   : NEBdBuffer      -> FullBdBuffer

                                              --  Methods of a BdBuffer
     op init   : NzNat           -> EmptyBdBuffer
     op max    : BdBuffer        -> NzNat
     op in     : BdBuffer        -> Nat
     op out    : BdBuffer        -> Nat
     op put    : Elem NFBdBuffer -> NEBdBuffer
     op top    : NEBdBuffer      -> Elem
     op pop    : NEBdBuffer      -> BdBuffer
   }

   axioms {
     var  M    : NzNat
     vars I O  : Nat
     vars E E' : Elem
     var  X    : BdBuffer
```

```
    var  NEX  : NEBdBuffer
    var  NFX  : NFBdBuffer

  eq [init]: init(M) = empty(M,0,0) .

  eq [max1]: max(empty(M,I,O)) = M .
  eq [max2]: max(state(E,NFX)) = max(NFX) .
  eq [max3]: max(full(NEX))    = max(NEX) .

  eq [in1]: in(empty(M,I,O))     = I .
  eq [in2]: in(state(E,NFX))     = 1 + in(NFX) .
  eq [in3]: in(full(NEX))        = in(NEX) .

  eq [out1]: out(empty(M,I,O))     = O .
  eq [out2]: out(state(E,NFX))     = out(NFX) .
  eq [out3]: out(full(NEX))        = out(NEX) .

  ceq [put1]: put(E,NFX) = state(E,NFX)
                if sd(in(NFX), out(NFX)) + 1 < max(NFX) .
  ceq [put2]: put(E,NFX) = full(state(E,NFX))
                if sd(in(NFX), out(NFX)) + 1 == max(NFX) .

  eq [top1]: top(state(E,state(E',NFX))) = top(state(E',NFX)) .
  eq [top2]: top(state(E,init(M)))       = E .
  eq [top3]: top(full(NEX))              = top(NEX) .

  eq [pop1]: pop(state(E,state(E',NFX))) = state(E,pop(state(E',NFX))) .
  eq [pop2]: pop(state(E,empty(M,I,O)))  = empty(M,I + 1,O + 1) .
  eq [pop3]: pop(full(NEX))              = pop(NEX) .
  }
}
```

This algebraic specification provides a view, not present in Maude, of object-oriented specification of concurrent systems, the intra-object view. The specification describes the data an object stores the representation of the data and the properties of the methods and how they manipulate the data.

Thus, in this specification of a bounded buffer, we have constructors for constructing the data, namely `empty`, `state` and `full`. The data store the information about the capacity and the history of the bounded buffer in terms of how many elements have been stored and retrieved.

The bounded buffer accepts five methods. `init` creates a new bounded buffer, `in` respectively `out` yield the elements being `put` respectively being retrieved, and `max` yields the capacity. Method `top` yields the oldest element, `pop` and `put` change the state of the bounded buffer.

As in our Maude specification `BD-BUFFER-WITH-STATES`, we implement the partiality of the operations by subsorting. We distinguish three sorts that implement three "states", namely empty, neither empty nor full and full, in which different sets of methods are applicable to a bounded buffer. To facilitate maximal reuse of code, we use the sorts `NotFull` and `NotEmpty` for which the methods are defined and inherited to the respective subclasses modeling the states.

We have made a design decision, namely that we do not consider intra-object concurrency and nondeterminism. Thus, we are allowed employ an algebraic specification style with equational reasoning. The object model of Maude and the concept of this algebraic specification style coincide. The example illustrates this, since we could use the same sorts as classes in our states-as-classes approach. Yet, there is a severe difference in how the specifications deal with partiality. In the Maude specifications, objects do not accept messages, i.e., there is no way to react to a message if it does not suit the state of the object. This is implemented by the asynchronous explicit message passing mechanism. In the algebraic specification we have only the application of functions, which corresponds to synchronous communication. We implement the states-as-classes approach by not allowing that `put` messages, that exceed the capacity of a bounded buffer, are transformed to `state` operations.

Let us make two remarks on the specification style provided by algebraic specification:

1. The restriction of the typing discipline becomes apparent. To describe a bounded buffer purely with the states-as-classes approach, one would need either dependent types or powerful sort constraints.

2. Both specifications describe the states of objects. Yet, for dealing with partiality, the view of Maude is much more convenient than the algebraic specification style.

2.5.2 Coalgebraic Specification

A coalgebraic object-oriented specification describes the observable properties of an object. An object can only be observed by its methods. The observations one makes are whether a method is accepted or not and, in case it yields a result, the result. Like in an algebraic specification, we deal with serialized objects. While the algebraic specification of a bounded buffer–and algebraic specifications in general—are not state-based, coalgebraic specifications are state-based: methods may yield a result and change the state of an object.

We use the specification language introduced in [Jac96a, Jac96b, Jac96c]. A predefined construct, is `self`, which refers to the state of the object. We implement our coalgebraic specifications in CafeOBJ. `self` is implemented as a variable of the sort of the class.

For the application of methods, we use typical object-oriented notation: `o.m(a)` denotes the application of method `m` with parameter `a` to object `o`, `o.m1(a1).m2(a2)` denotes the application of `m2` with parameter `a2` after `m1` with parameter `a1`.

A coalgebraic specification of a bounded buffer observes the buffer via five of its methods: `max`, which yields the capacity of a bounded buffer, `in`, which yields the number of elements being put into the bounded buffer, `out`, which yields the number of elements being retrieved from the bounded buffer, `top`, which yields the oldest element, and `cont`, which yields the elements stored by the bounded buffer as a list of elements.

```
module COALG-BD-BUFFER  {
  import {
   protecting (LIST-ENRICHED)
   using (ACZ-CONFIGURATION)
   protecting (RWL)
  }

  signature {
   [X]                        -- the class sort
   [Nat < Object]
   [List < Object]
   [Elem < Object]

   op init      : NzNat  -> X
   op _.max     : X      -> NzNat { l-assoc }
   op _.in      : X      -> Nat   { l-assoc }
   op _.out     : X      -> Nat   { l-assoc }
   op _.put(_)  : X Elem -> X     { l-assoc }
   op _.pop     : X      -> X     { l-assoc }
   op _.top     : X      -> Elem  { l-assoc }
   op _.cont    : X      -> List  { l-assoc }
  }

   axioms {
     var  M    : NzNat
     var  L    : Nat
     var  E    : Elem
     var  self : X
     var  C    : List

   rl  [max1]: init(M).max        => M .
   rl  [max2]: self .put(E) .max  => self .max .
   crl [max3]: self .pop .max     => M
                      if (self .max ==> M) .

   rl  [in1]: init(M).in      => 0 .
   rl  [in2]: (self .pop) .in => self .in .
```

```
  crl [in3]: (self .put(E)) .in => (self .in) + 1
                   if  (sd(self .in, self .out) < self .max) .

  rl [out1]: init(M).out      => 0 .
  rl [out2]: (self .pop) .out => (self .out) + 1 .
  crl [out3]: (self .put(E)) .out => self .out
                   if  (sd(self .in, self .out) < self .max) .

  crl [top]: self .top => E:Elem
                if (sd(self .in, self .out) < self .max)
                and self .cont ==> (C:List E:Elem):List .
  rl [cont1]: init(M).cont      => eps:List .
  rl [cont2]: self .put(E) .cont => (E:Elem (self .cont)):List .
 crl [cont3]: self .pop .cont    => C
                if self .cont ==> (C:List E:Elem):List .
 }
}
```

A coalgebraic class specification gives the class sort, `X`, the functionality of the methods, and the axioms which specify the causalities between observations made by the methods. We decided implement this specification in CafeOBJ in the easiest way. Thus, we declare all sorts of results of methods to be a subsort of sort `Object`. This makes natural numbers and lists "states" of a transition system.

2.6 Related Work

Maude is a very small but abstract and powerful specification language. Let us discuss a number of other object-oriented languages and how they deal with concurrency.

This section is organized as follows. We briefly recall the basic principles of Maude. Then, we give an overview of object-oriented languages and discuss their origin and the purpose they were designed for. Finally we take the issues of Sect. 2.6.1 we consider to be particular to Maude and discuss in detail how other object-oriented languages realize them.

2.6.1 Summary of Maude

We summarize the most important ideas which we have introduced in this chapter, in which we gave an introduction to Maude and to the concepts and language constructs which we developed for a concurrent object-oriented language.

- *Object model*: Maude's objects are autonomous state-based entities. The synchronization code determines whether or not an object accepts a message. Maude abstracts from methods.

- *Concurrency*: Maude has inter-object concurrency.
- *Communication between objects*: Objects communicate via explicit asynchronous message passing and implicit synchronous communication.
- *Property of objects*: State transitions of objects may happen.
- *Structuring and reuse*: Maude's structuring and reuse constructs are states as classes, inheritance, subconfigurations, message algebra and modules.
- *Philosophy*: Maude is a general-purpose language, it is heterogeneous and reflective.

2.6.2 First Overview

When comparing the object models of sequential object-oriented languages to the object models of concurrent object-oriented languages, one might get the impression that, although the sequential object models differ in fundamental issues, they have much more in common than the concurrent object models.

In our opinion the origin of the language and the intention with which it was designed helps to explain it. Object-oriented concurrent languages that are based on sequential languages are typically good in handling data types, but concurrency plays a less important role. Object-oriented languages that are based on concurrent languages provide much richer concepts for concurrency and fewer concepts for handling the data.

Most of the object-oriented concurrent languages are based on sequential object-oriented languages: one might get the impression that each popular sequential object-oriented language has (at least one) concurrent or distributed descendant: C++ [Str86] with CC++ [CK93] and other concurrent versions [KK93, SG92], Smalltalk [GR83, Gol83] with ConcurrentSmalltalk [YT87], and Distributed Smalltalk [Ben90, SB88] and Eiffel [Mey92] with a concurrent version of Eiffel [Mey93] and distributed Eiffel// [Car93].

A few object-oriented languages are based on concurrent languages. Pict [PT97] is an implementation of the π-calculus which contains, in some versions, an object notation. The π-calculus is also used to implement $\pi o\beta\lambda$ [Jon93b] and is the basis for the object notation in [Wal91, Wal94, PW96]. The actor model [Agh86] resembles many object-oriented concepts and in particular asynchronous message passing. Thus the actor model is closely related to several object-oriented languages, e.g., [Yon90, Frø92, Tal96] and the object-model described in [KLMW94, Kri97]. The languages Java [AG96] and Ada [Bur85] both offer object-oriented concepts and concurrency.

Specification languages offer a greater variety of basic concepts. OS [Bre91] is based on algebraic specifications, OOSpectrum [WNL95] on the specification language SPECTRUM [BFG$^+$93], TROLL on temporal logic and algebraic specification [HSJ$^+$94], Oz on constraint programming [Smo94] and the approach of [Jac96c] on category theory.

In our approach to object-oriented concurrent languages, we have objects as concurrent entities. But there is an entirely different approach to object-oriented concurrent programming, in which object orientation serves only to structure the code of a concurrent programming language. This approach is used in implementations of operating systems like CHARM++ [KK93], or libraries for the design of concurrent systems, like protocol classes in [GFG96], Ada [Bur85] and Java [AG96].

2.6.3 Second, Closer Look

In Sect. 2.6.1 we have stated several issues that are particular to Maude. In this section we compare the languages mentioned in Sect. 2.6.2 with respect to these issues.

Object model: Objects encapsulate a state which is only accessible via methods. Thus, it is not surprising that most object-oriented concurrent languages are state-based, even when the state is modeled as a function like in Cecil [Cha95], or as a process like in Pict [PT97]. An exception is OS [Bre91], which specifies properties of methods in an algebraic way.

Methods are typically an essential component of an object. Maude is the only language which abstracts from methods: we have only messages—for communication between objects—and state changes of objects.

The synchronization code of an object determines whether an object reacts to a message. Only a few object-oriented concurrent languages like $\pi o\beta\lambda$ [Jon93b] and POOL [AdBKR86] do not employ synchronization code. Maude's objects do not have explicit synchronization code, the transition rules describe instead whether an object accepts a message. In Maude, whether a transition is executed depends only on the current state and the transition rules describe whether a transition is possible. Other languages have many ways of expressing their synchronization code: it may be part of the methods or a component of the objects; it may depend on the state or on the history, of the object; it may describe which methods are enabled or disabled. [MY93] gives an excellent survey of different concepts for synchronization code. Particular to Maude and unique among all the languages we mention is that the synchronization code may comprise the state of several objects, and the values of attributes as well as the parameters of messages and not, like all languages in [MY93], only the message or method name.

Concurrency: The majority of object-oriented concurrent languages supports only inter-object concurrency: only one method of an object is active, and there is no parallel composition operator in the methods. Only Pict [PT97] and early versions of $\pi o\beta\lambda$ [Jon92] support intra-object concurrency.

One way to introduce inter-object concurrency is to have objects with threads, which control and trigger the activities of the object. Early versions of POOL [AdBKR86] have objects with threads. This concept never became popular since scheduling the activity triggered by the thread and the activity triggered by method invocations is complicated.

Another way of introducing inter-object concurrency is to let a method return the control to the calling object and continue to execute (the rest of) the method. Hereby only a very small amount of concurrency is introduced. This is a typical approach to introducing concurrency in sequential languages. Languages that use this approach are, e.g., POOL [Ame87] and $\pi o\beta\lambda$ [Jon92].

Communication between objects: Objects communicate via messages or method calls. There is asynchronous and synchronous communication. In asynchronous communication objects communicate via messages; the sender and receiver of a message do not synchronize: a message is part of the state and and is waiting to be processed. When a message is accepted a method is executed. In synchronous communication sender and receiver objects synchronize; objects communicate via methods calls.

The method calls in sequential object-oriented languages are synchronous. Concurrent object-oriented languages based on sequential languages have typically synchronous communication as, e.g., POOL [Ame87], $\pi o\beta\lambda$ [Jon92]. TROLL and its derivatives [HSJ$^+$94] employ synchronous communication as well. Note that in all these languages we have rendezvous communication, i.e., a synchronous communication between two partners.

Asynchronous communication via messages is employed in Maude, in all languages based on the Actor model [Agh86, Frø92, Yon90], in the latest version of Pict [PT97], and in database languages [KMWZ91, KLMW94, Kri97].

Some languages allow communication via channels in addition to communication via messages or method calls, e.g., CC++ [CK93], Pict [PT97] and Hybrid [Nie92, Kon92].

Particular to Maude is the concept of synchronous implicit communication: transition rules specify the number of objects and messages that perform a joint state transition. In such a rule, a fixed but arbitrary number of objects can synchronize and communicate—not, as in all other languages mentioned here, only one or two objects. Particular to synchronous implicit communication is that it allows to exchange all the data stored inside the objects to compute the resulting state without method invocation.

Properties of objects: Maude's transition rules specify that a state transition may happen if the pattern at the left-hand side of the rule matches a part of the state and the precondition of the rule is satisfied. This rather weak "may" is due to the asynchronous communication.

Specification languages like TROLL [HSJ$^+$94] and Oz [Smo94], are more property-oriented. TROLL [HSJ$^+$94] comprises temporal logic to describe the causalities of actions of different objects. Thus, TROLL as a specification language can express more properties directly.

Structuring and reuse: Our structuring and reuse constructs are states as classes, inheritance, subconfigurations, and the message algebra. The seminal article on the inheritance anomaly [MY93] demonstrates that inheritance in an object-oriented concurrent language is hardly possible. Our conclusion is that inheritance alone is not sufficient as a reuse concept. In Sect. 2.3 we have demonstrated that our set of reuse constructs circumvents the inheritance anomaly.

In fact, many of the already cited object-oriented concurrent languages, POOL [Ame90, Ame92], $\pi o\beta\lambda$ [Jon92], Pict [PT97] and TROLL [HSJ$^+$94], do not support inheritance.

TROLL [HSJ$^+$94] and the Actor model [Agh86, AFK$^+$93] have aggregation which is similar to our subconfigurations.

The concept of a message algebra is unique as a reuse construct in object-oriented languages. In [WNL95] the concept of message algebra has been introduced for verification of object-oriented concurrent specifications. The concept of message combinators is well known in process algebras as, e.g., the π-calculus [MPW92].

Subclassing is a form of reuse in strongly typed sequential object-oriented languages. In sequential object-oriented (programming) languages, the inheritance mechanism is much stronger than Maude's and it allows manifold manipulations of the inherited code. Thus, inheritance is considered to be reuse of parts of code, while subclassing is reuse of classes. The characterization of a subclass according to [WZ88] is that an object of a subclass can be used whenever an object of a superclass is expected. In this sense, Maude's inheritance relation is also a subclass relation. There are well-established results on subclass relations in object-oriented sequential languages [CMMS91, HP95, PS94, PS97]. In a concurrent setting, it is necessary to take the behavior of objects into account; for the type of an object it is also relevant, when it accepts which messages. This leads to behavioral subtyping. The approaches towards dynamic subtyping of object-oriented concurrent languages of [Cus91, Nie93] are based on failure semantics. The approaches to behavioral typing and subtyping of concurrent languages like the π-calculus [PS96a, Vas94] are based on simulation relations.

The focus of our work lies on the reuse and structuring concepts at the level of objects and classes. *Design patterns* [PS96b] provide reuse at a coarser granularity. Sect. 5.2 presents a specification of a matching pair of classes matching sender and receiver in a specification of a protocol. This (small) number of classes together with their transition rules can be viewed as a design pattern. How the reuse of such design patterns can be integrated in a complex specification is demonstrated in Sect. 5.2.6. *Frameworks* are complex systems designed for reuse. They are specific for an application field [BE93, EW96]: they are used when similar systems can be derived from a “prototype” with few and quite predictable changes or instantiations. For both design patterns and frameworks, the “prototypes” have to be customized and extended. The reuse concepts at the object level, which we have developed, are a prerequisite for reuse of more complex units, as provided by design patterns or frameworks.

Particular to our approach is that we are interested in the inheritance of properties. In Sect. 4.4 we present results on the inheritance of properties via these reuse constructs.

In general: Maude is a general-purpose language [Mes96]; it is used to implement other concurrent formalisms [Mes96], e.g., Petri Nets [Mes93a], the π-calculus [Vir96], actor languages [Tal96], protocols [Lan96, PMO96], databases [MQ93], graph algorithms [Har95] and even other object-oriented concurrent specification formalisms like TROLL light [DG93]. We present specifications of bounded buffers and of communication protocols (see Sect. 5.2.1), and a specification of an airport (see Sect. 2.4) which is based on [Sal93].

The execution model has been extended to deal with real time [KW95, ÖKW96, ÖM96], reflection and strategies [CM96].

None of the languages mentioned here has this expressiveness. One of the reasons why we can accomplish this is reflection [CM96], which we use, e.g., to specify new reuse and structuring concepts. All other object-oriented concurrent languages lack this concept.

It is somewhat surprising that Smalltalk, *the* object-oriented language, is the only homogeneous object-oriented language among those cited, although we refer to several "research" languages which we would expect to give preference to elegance than to efficiency. Only the language Pict [PT97], which implements objects as π-calculus processes and classes as process declarations, reaches a similar level of purity—although Pict's basic concept is not the object but the process.

2.7 Remarks and Comments

We conclude this chapter with a brief review of our results and with some comments. We relate the material of this chapter also to the next chapters dealing with the semantics of Maude and with the verification and refinement of Maude specifications.

In this chapter we have introduced our specification language Maude. Maude is both very abstract and concrete. The communication and synchronization mechanisms are abstract and the way computational progress is specified is rather concrete. Maude is an operational language and, moreover, the transition rules allow only to describe single steps. Message constructors provide us with the ability to compose messages and to specify some sort of control flow. To overcome this rather operational and not property-oriented view, we use in Chap. 4 a more property-oriented specification language, the μ-calculus. We use it to specify properties of the behavior of both single classes (see Sect. 5.1.3) and collections of objects (see Sect. 4.3).

We have developed a set of reuse constructs in Maude. In this chapter we have proven that they enable various ways of reuse and provide a powerful means for structuring a specification. In Sect. 4.4 we review these reuse constructs and discuss which properties, phrased in μ-calculus, are inherited through them. Throughout this work we demonstrate how to apply these constructs. In particular, in Sect. 5.2.6, we demonstrate how they can be used to integrate local refinements into a complex specification.

In this chapter we have demonstrated with the example of an airport how the features of Maude and, in particular, our sets of reuse constructs can be applied to structure a large specification. We have given guidelines, but we cannot claim to have an analysis and design method.

The different specifications of the bounded buffer as well as fragments of the airport specification demonstrate that Maude supports both an abstract and a concrete specification style. In Chap. 5 we develop a formal refinement relation between Maude specifications at different levels of abstraction.

Maude is a very expressive language. It abstracts from the implementation of methods. Thus, in Sect. 2.5, we have discussed two specification styles: algebraic and coalgebraic specification. The relation between Maude specifications and the algebraic and coalgebraic specification is discussed further in Sect. 3.6. Maude is our main notation, and we use these two alternative specification styles to reason about the object model and the notion of observability and encapsulation in Maude.

We must admit that the object model of Maude and Maude's notation are quite unconventional. Moreover, from the object-oriented point of view, the visibility and accessibility of the internal state is not acceptable. In our point of view, this "object-oriented" drawback is outweighted by the abstractness of Maude w.r.t. communication and synchronization.

Chapter 3

Specifications and Models

In Chap. 2, we have introduced our specification language Maude. In the present chapter we deal with the semantics of Maude specifications and object-oriented specifications in general. We rate our approach as an algebraic approach to the object-oriented specification of distributed systems. This means we take many-sorted algebras and the formal techniques developed for specifications and enrich them with the features of Maude. There are three features not present in traditional many-sorted algebraic specifications [Wir90]: (1) order-sortedness, (2) nondeterminism and (3) partiality. Let us explain how we deal with these concepts:

- In an order-sorted algebra, the sort and class hierarchy of a specification is interpreted by a subset relation on the carrier sets of the sorts. We demand certain properties of our specifications to ensure that reasoning in order-sorted specifications is as intuitive as in many-sorted algebras.
- Nondeterminism is inherent in descriptions of concurrent processes. In algebraic specifications the operations are modeled as functions. We use a relation to model the state transition system with the nondeterministic behavior.
- Partiality is inherent in our object model, since synchronization code is used to prevent methods from being accepted. Thus, in a sensible specification, the synchronization code prevents that methods are applied to a state for which they are not defined. We use order-sorted algebras to model the object such that the operations on them are total.

This section is organized as follows. We begin with basic notions of order-sorted algebraic specifications, i.e., with signatures, algebras specifications and models and mappings. Here we rely on [Bre91, Wir90]. We introduce the properties we would like the order-sorted specifications and algebras to have and use here [HN96]. We extend the many-sorted algebras with a transition relation [Jac96a, Jac96b, Jac96c, Rei95, Rut95]. We introduce behavioral abstraction following [BHW95, Jac96c, MPW92]. As an example we use a specification of a bounded buffer. At the end of this section, we compare our work to other approaches to specification.

3.1 Order-Sorted Algebraic Specifications

An algebraic specification gives the characteristic properties of a data type. These are the signature and a set of axioms, typically first-order formulas. Particular to object orientation is the class hierarchy, which is modeled by a subtype hierarchy on sorts. Thus, we use order-sorted signatures, order-sorted specifications and order-sorted algebras.

The definitions of this section can be found in a similar form in, e.g., [Bre91, GM92, HN96, Wir90].

Definition 3.1 (Order-sorted signature) An *order-sorted signature* is a triple $(S, \leq, OP)$, where $(S, \leq)$ is a partially ordered set of sorts and OP is an indexed family $OP = OP_{w,s}$, where $w \in S^\star$ and $s \in S$.

$\leq \subseteq S \times S$ extends naturally to $S^\star \times S^\star$ such that $s_1 \times \cdots \times s_n \leq s'_1 \times \cdots \times s'_n$ if $s_i \leq s'_i$ (and analogously to $(S^\star, S) \times (S^\star, S)$).

Note that we allow overloading and polymorphism in our specifications.

Let us explain our notation for signatures. In this section we write a signature as either $(S, \leq, OP)$ or in the notation of CafeOBJ depending on the context. Empty components of a signature may be omitted. In object-oriented specification we distinguish two components of sorts in the signatures: (non-object) sorts and classes. In object-oriented specifications we distinguish also two components of operations: operations and messages.

We assume that the carrier set of each sort contains at least one element. A signature is called *sensible* if it admits at least one ground term for each sort. We assume that all signatures are sensible.

Definition 3.2 (Term) Let $\Sigma = (S, \leq, OP)$ be an order-sorted signature and let $X = (X_s)_{s \in S}$ be an S-sorted set of variables. The set $T_s(\Sigma, X)$ is inductively defined by the following rules

1. $x \in T_s(\Sigma, X)$ if $x \in X_s$
2. $op \in T_s(\Sigma, X)$ if $op \in OP_s$
3. $op(t_1, \ldots, t_n) \in T_s(\Sigma, X)$, if $op \in OP_{s_1, \ldots s_n, s}$ and $t_i \in T_{s_i}(\Sigma, X)$ for $1 \leq i \leq n$.
4. $t \in T_s(\Sigma, X)$, if $t \in T_{s'}(\Sigma, X)$ and $s' \leq s$.

We denote the *least sort* of a term t by $LS(t)$.

Signatures and terms over a signature are the syntax of the descriptions. Algebras are the semantics of signatures and terms: a signature Σ defines a class of Σ-algebras. An Σ-algebra contains the elements which are at the syntactic level represented as terms and functions which are represented by function symbols.

Definition 3.3 (Order-sorted algebra) Let $\Sigma = (S, \leq, OP)$ be an order-sorted signature. An order-sorted Σ-algebra A consists of

- an S-sorted family of carrier sets $(A_s)_{s \in S}$
- a total function $f^A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$ for each $f : s_1 \ldots s_n \to s$ in OP and
- a subset relation where $s \leq s'$ implies $A_s \subseteq A'_s$.

The class of order-sorted Σ-algebras is denoted by $Alg(\Sigma)$.

Example. (Natural numbers) Let us give an order-sorted signature of the specification of natural numbers.

```
module STD-NAT {

  signature {
   [NzNat < Nat]

   op zero :         -> Nat
   op succ :  Nat    -> NzNat
   op pred :  NzNat -> Nat
   }
}
```

The signature of natural numbers contains two sorts: `Nat`, the natural numbers and `NzNat`, the natural numbers without `zero`. `NzNat` is a subsort of `Nat`. We have three operations: `zero`, `succ` and `pred`. `zero` yields a natural number, `succ` is a mapping which takes as a parameter a natural number and yields a `NzNat`, number, `pred` accepts as a parameter only a `NzNat` and yields a `Nat`. Fig. 3.1 depicts a $\Sigma_{\text{STD-NAT}}$ algebra, i.e., the signature part of specification `STD-NAT`.

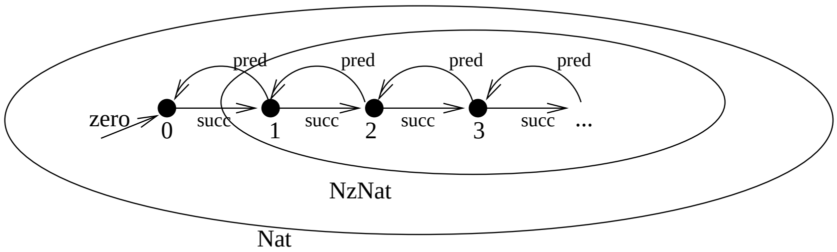

Figure 3.1: An algebra of natural numbers

$\diamondsuit$

An algebra may contain elements that are not representatives of terms of a signature. Since we are interested in elements which are representatives of terms, we introduce the notion of a term-generated algebra.

Definition 3.4 (Valuation, interpretation) If X is an S-sorted set and A a Σ-algebra, then a map $v : X \to A$ is called *valuation* of X in A.

The *interpretation* of a term t in A (w.r.t. v) is a map $v^\star : T(\Sigma, X) \to A$ which is defined as follows:

1. $v^\star(x) =_{\text{def}} v(x)$ for each $x \in X$.
2. $v^\star(f(t_1, \ldots, t_n)) =_{\text{def}} f^A(v^\star(t_1) \ldots v^\star(t_n))$ for each $f : s_1 \ldots s_n \to s$, $t_i \in T_{s_i}(\Sigma, X)$.

The interpretation of a ground term t in a term-generated algebra A is written as t^A.

Definition 3.5 (Term-generated algebra, computation structure) A Σ-algebra A is called *reachable* if, for each sort s of Σ and for each element a of the carrier set A_s of s, there exists a term t with $a = t^A$, i.e., $v^\star$ is surjective. A reachable algebra is also called a *term-generated* algebra or *computation structure*. The class of all term-generated Σ-algebras is denoted by $Gen(\Sigma)$.

$T(\Sigma, \emptyset)$ is called the *ground term algebra*.

We are also interested in "parts" of algebras. Substructures are parts of algebras which observe closure properties, namely, which are closed with respect to application of function symbols.

Definition 3.6 (Subalgebra) For any Σ-algebra A, a Σ-*subalgebra* B of A is a Σ-algebra such that, for any sort $s \in \Sigma$, $B_s = A_s$ and, for any function symbol $f_{s_1, \ldots, s_n, s} \in \Sigma$ and for all $b_1 \in B_{s_1}, \ldots b_n \in B_{s_n}$, $f^B(b_1, \ldots, b_n) = f^A(b_1, \ldots, b_n)$.

3.1.1 Mappings between Signatures and between Algebras

In the previous section we have introduced signatures, terms and algebras. In this section, we introduce mappings between signatures and mappings between algebras and explain the relations between these two mappings.

Definition 3.7 (Order-sorted signature morphism) Let $\Sigma = (S, \leq, OP)$ and $\Sigma' = (S', \leq', OP')$ be order-sorted signatures. An *order-sorted signature morphism* $\sigma : \Sigma \to \Sigma'$ is a pair (σ_S, σ_{OP}) such that

$$\begin{array}{lcl} \sigma_S & : & (S, \leq) \to (S', \leq') \text{ is a monotonic function} \\ \sigma_{OP} & = & (\sigma_{w,s})_{w \in S^\star, s \in S} \text{ is a family of functions with } \sigma_{w,s} : OP_{w,s} \to OP'_{\sigma_{S^\star}(w), \sigma_S(s)} \end{array}$$

The extension of a signature morphism σ to a mapping between terms is denoted by $\sigma^\star$. A signature morphism is a mapping which is compatible with sorts and function symbols.

A Σ-homomorphism is a mapping between two Σ-algebras. Analogously to signature morphisms, which are compatible with the signature, Σ-homomorphisms are compatible with function applications.

Definition 3.8 (Order-sorted Σ-homomorphism) Let $\Sigma = (S, \leq, OP)$ be an order-sorted signature and A and B two Σ-algebras. An order-sorted Σ-*homomorphism* h is a family of maps $(h_s)_{s \in S}$ such that

- $f : s_1 \ldots s_n \to s \in OP$ and $a_1 \in A_{s_1} \ldots a_n \in A_{s_n}$ implies $h_s(f^A(a_1, \ldots, a_n)) = f^B(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$.
- and, for all $s, s' \in S$, $s \leq s'$ and $a \in A_s$ implies $h_s(a) = h_{s'}(a)$.

Note that a Σ-homomorphism preserves equality, i.e., $t_1 = t_2$ implies $h(t_1) = h(t_2)$ but $t_1 \neq t_2$ implies not $h(t_1) \neq h(t_2)$.

Lemma 3.9 Let A,B be Σ-structures and $h : A \to B$ a Σ-homomorphism. Then $h(t^A) = t^B$ for all ground terms t.

A direct consequence is that a Σ-homomorphism between two term-generated Σ-algebras is uniquely determined by $h(t^A) = t^B$.

Example. Let the signature be defined by

$$\Sigma = \{ \quad \begin{array}{l} \texttt{[Nat]} \\ \texttt{op zero:\ \ -> Nat} \\ \texttt{op succ:\ \ Nat -> Nat} \end{array}$$
$$\}$$

Consider two Σ-algebras:

(1) $\mathcal{N}$, the natural numbers and
(2) $\mathcal{N}'$ with
$\mathcal{N}'_{\mathrm{Nat}} = \{0, 1\}$,
$\texttt{zero}^{\mathcal{N}'} = 0$ and
$\texttt{succ}^{\mathcal{N}'}(x) = (1 + x) \bmod 2$.

The Σ-homomorphism $\sigma : \mathcal{N} \to \mathcal{N}'$ is defined by:

$$\begin{array}{lcll} \sigma(x) & = & x \bmod 2 & \text{for } x \in \mathcal{N}_{\mathrm{Nat}} \\ \sigma(\texttt{zero}^{\mathcal{N}}) & = & 0 & \end{array}$$

Let us prove that σ is a homomorphism.

$$\begin{array}{ll} & \sigma(\texttt{succ}^{\mathcal{N}}(x)) \\ = & \quad \{ \text{ valuation } \} \\ & \sigma(x + 1) \\ = & \quad \{ \text{ definition of } \sigma \text{ } \} \\ & (x + 1) \bmod 2 \end{array}$$

$$\begin{array}{ll} & \texttt{succ}^{\mathcal{N}'}(\sigma(x)) \\ = & \quad \{ \text{ definition of } \sigma \text{ } \} \\ & \texttt{succ}^{\mathcal{N}'}(x \bmod 2) \\ = & \quad \{ \text{ valuation } \} \\ & ((x \bmod 2) + 1) \bmod 2 \\ = & \quad \{ \text{ arithmetic } \} \\ & (x + 1) \bmod 2 \end{array}$$

$\diamondsuit$

Signature morphisms are mappings between signatures, homomorphisms are mappings between algebras. A reduct relates the mapping of a signature to a mapping between algebras. A reduct retrieves an Σ-algebra in the image of a Σ-homomorphism.

Definition 3.10 (Reduct) Let $\Sigma = (S, \leq, OP)$ be an order-sorted signature. Given an order-sorted Σ-algebra A' and an order-sorted signature morphism $\sigma : \Sigma \to \Sigma'$, the order-sorted Σ-algebra buried inside A', the *σ-reduct* of A', written $A'\,|_\sigma$, is the order-sorted Σ-algebra with the carrier set $(A'\,|_\sigma)_s =_{\text{def}} A'_{\sigma(s)}$ for each sort $s \in S$, and $f^{A'|_\sigma} =_{\text{def}} \sigma(f)^{A'}$ for each $f \in OP$.

Let $h' : A' \to B'$ where A' and B' are order-sorted Σ'-algebras, the σ-reduct of h' is the Σ-homomorphism $h'\,|_\sigma\colon A'\,|_\sigma \to B'\,|_\sigma$ defined by $(h'\,|_\sigma)_s =_{\text{def}} h\,|_{\sigma(s)}$ for b all $s \in S$.

3.1.2 Specifications and Models

In the previous subsection we have considered signatures and algebras. A specification consists of a signature and a set of axioms, a model is an algebra which satisfies the axioms.

Definition 3.11 (Order-sorted specification) An order-sorted specification $Sp = (\Sigma, E)$ consists of an order-sorted signature Σ and a set of first-order formulas E.

In algebraic specifications, the formulas are conditionally and unconditionally quantified equations.

Definition 3.12 (Equation) Let $\Sigma = (S, \leq, OP)$ be a regular, order-sorted signature. A *Σ-equation* is a triple, (X, t, t'), where X is a variable set and t, t' are in $T(\Sigma, X)$ with $LS(t)$ and $LS(t')$ in the same connected component of $(S, \leq)$. We use the notation $(\forall X)t = t'$. An order-sorted Σ-algebra A satisfies a Σ-equation $(\forall X)t = t'$, written $A \models (\forall X)t = t'$, iff $v^\star(t)_{LS(t)} = v^\star(t')_{LS(t')}$ for every valuation $v : X \to A$.

A satisfies a set of equations if it satisfies all equations in it.

A *conditional Σ-equation* is written as $(\forall X)t = t'$ `if` C, where the condition C is a finite set of unquantified Σ-equations involving only variables in X. An order-sorted Σ-algebra A satisfies a conditional Σ-equation $(\forall X)(t = t'$ `if` $C)$ if, for each valuation $v : X \to A$ such that, if for each equation $t_i = t'_i$ in C, $v^\star(t_i) = v^\star(t'_i)$, then $v^\star(t) = v^\star(t')$.

In the previous section we have introduced signatures and algebras. We consider specifications, which consist of a signature and a set of formulas, and algebras which satisfy these formulas.

With every specification $Sp = (\Sigma, E)$ we associate the class of algebras in which all equations in E hold: $Alg(Sp) = \{A \mid A \in Alg(\Sigma), G \in E \text{ implies } A \models G\}$.

Analogously $Gen(Sp) = \{A \mid A \in Gen(\Sigma), G \in E \text{ implies } A \models G\}$.

We are only interested in term-generated algebras, since term-generated algebras contain only elements which are representatives of syntactical constructs.

Within the class of algebras of a specification there are distinguished elements, the initial and the terminal algebras. In the initial model only those elements are identified, which are provably equal in the specification. Dually, in the terminal algebra all elements are identified which are not distinguished in the specification. The Σ-homomorphisms are mappings between Σ-algebras. They are uniquely determined, provided the algebras are term-generated. For a characterization of initial and terminal algebras see [Wir90].

There are three different approaches to specification: (1) the initial approach, in which the semantics of a specification is the initial model, (2) the terminal approach, in which the semantics of a specification is its terminal model and (3) the loose approach, in which the semantics is the class of all term-generated models. We use in this section the loose approach and later, in refinement, the initial approach.

We observe the following notational conventions: the class of all term-generated models is denoted by $Mod(Sp)$, the initial term-generated model by $I(Sp)$ and the terminal term-generated model by $Z(Sp)$.

The loose semantics, in which $Mod(Sp)$ is the class of all models, is typically used in specification: each term-generated algebra in which the axioms hold is considered to be a model of the specifications. Typically, there are many models of a specification since the specification gives a property-oriented description which can have many different implementations.

Definition 3.13 (Order-sorted congruence relation) Let $\Sigma = (S, \leq, OP)$ be an order-sorted signature and let A be an order-sorted Σ-algebra. An *order-sorted congruence* on A is an S-sorted equivalence relation $\sim$, i.e., $\sim = (\sim_s)_{s \in S}$, if $\sim_s \subseteq A_s \times A_s$ for all $s \in S$ is reflexive, symmetric and transitive and if, for any $f_{s_1 \ldots, s_n, s} \in \Sigma$, $a_1 \sim_{s_1} b_1, \ldots, a_n \sim_{s_n} b_n$ implies $f^A(a_1, \ldots, a_n) \sim_s f^A(b_1, \ldots, b_n)$ for all $a_1, b_1 \in A_{s_1}, \ldots, a_n, b_n \in A_{s_n}$, and if $a_1 \sim_{s'} a_2$ implies $a_1 \sim_s a_2$ for all $s, s' \in S$, $s \leq s'$ and $a_1, a_2 \in A_{s'}$.

Let us now consider the different models and classes of models of a specification.

Lemma 3.14 (Quotient algebra) If $\sim$ is a Σ-congruence in A, then the *quotient* $A/\!\sim$ is a well-defined algebra, where $(A/\!\sim)_s =_{\text{def}} \{[a] \mid a \in A_s\}$, where $[a] = \{b | b \in A_{MS(a)}, a \sim_s b\}$ for each $s \in S$ ($MS(a)$ is the maximal sort of a in s) and for $f : s_1, \ldots, s_n \to F$, $a_1 \in A_{s_1}, \ldots a_n \in A_{s_n}$, $f^{A/\sim}([a_1], \ldots, [a_n]) =_{\text{def}} [f^A(a_1, \ldots, a_n)]$.

Thus every congruence on a Σ-algebra defines a Σ-algebra. This result allows to consider equivalence classes of terms only.

Lemma 3.15 Let A be a Σ-computation structure. For any Σ-algebra B there exists a Σ-homomorphism from A to B iff $\sim^A \subseteq \sim^B$.

Each congruence defines a homomorphism between a computation structure and the quotient algebra by $h(t) = h(t')$ if $t \sim t'$.

Note that we do not require the congruence relation to have special properties. However, arbitrary congruences may lead to algebras which we consider to be unintuitive. We refer here to Sect. 3.2, where we give properties of signature morphisms and, thus, also of homomorphisms which ensure that they observe certain properties.

Thus, we have two ways of establishing relations between Σ-algebras: equivalence relations and Σ-homomorphisms. The initial algebra of a class of algebras identifies only those elements for which the equality is derivable from the equations of the specifications with the equational calculus. The other algebras identify more elements, i.e., their congruence relations are supersets of the congruence relation of the initial algebra. For each Σ-algebra there exists a unique Σ-homomorphism from the initial algebra. In the terminal algebras all elements are identified which are not specified to be different. While the equality in the initial algebra is the smallest one (with respect to the subset ordering), the congruence of the terminal algebra is the largest one – which does not violate the inequations of the specification.

3.1.3 Mappings between Specifications

Signature morphisms are mapping between signatures, Σ-homomorphisms mappings between algebras. A signature morphism induces not only a translation between signatures but also a translation between specifications and a mapping between algebras.

Let us introduce some notation: the abstract data type $\langle \Sigma, Mod(\Sigma, E)\rangle$ described by a specification $Sp = (\Sigma, E)$ is written as $[\Sigma, E]$ or $[Sp]$.

Definition 3.16 (Specification building operators) Let Σ, Σ' be order-sorted signatures and let C, C' be classes of monotone order-sorted Σ-algebras.

1. A signature morphism $\sigma : \Sigma \to \Sigma'$ induces a specification-building operation *translate _ with _* , which translates any specification $\langle \Sigma, C\rangle$ to a Σ'-specification:

   $$\textit{translate}\ \langle \Sigma, C\rangle\ \textit{with}\ \sigma = \langle \Sigma', C'\rangle$$

   where $C' = \{A' \mid A' \in Alg(\Sigma'), A'|_\sigma \in C\}$

2. A signature morphism $\sigma : \Sigma \to \Sigma'$ induces a specification-building operation *derive from _ by* σ, which yields the class of Σ-algebras buried inside an algebra:

   $$\textit{derive from}\ \langle \Sigma', C'\rangle\ \textit{by}\ \sigma = \langle \Sigma, C\rangle$$

   where $C = [\{A' \mid A' \in C', A'|_\sigma \in Alg(\Sigma)\}]$ ($[\ \_\ ]$ denotes the closure of a set of algebras with respect to isomorphism).

3. Let $\Sigma_0 \subseteq \Sigma$ and $in : \Sigma_0 \to \Sigma$ be the canonical injection. We define the specification building operation *export _ from _* :

   $$\textit{export}\ \Sigma_0\ \textit{from}\ \langle \Sigma, C\rangle\ =_{\mathrm{def}}\ \textit{derive from}\ \langle \Sigma, C\rangle\ \textit{by in}$$

With these specification-building operations we can derive specifications from existing specifications. For more operators and their properties see [Wir90].

The specification-building operation provides us with the ability to use the signature morphism to "translate" the specifications such that the translation is compatible with "$\models$".

Lemma 3.17 (Specification morphisms) Let Sp be a coherent order-sorted specification, $\sigma : \Sigma \to \Sigma'$ a signature morphism and $\sigma^\star$ its extension to terms. Then

$$\mathit{translate}\langle\Sigma, C\rangle \ \mathit{with}\ \sigma = [\Sigma', \sigma^\star(E)]$$

Proof. [Wir90]

$\square$

3.2 Properties of Order-Sorted Signatures and Specifications

Compared to many-sorted algebras, order-sorted algebras have assets and drawbacks. Let us give an example, motivate why we use order-sorted algebras and discuss some of the problems that arise from it. This motivates that we are looking for certain properties of signatures and specifications which allow us to reason about order-sorted specifications without taking special care of the order-sortedness.

Let us recall the order-sorted specification `NAT` of natural numbers from Sect. 3.1. The order-sortedness is used to model the partiality of the operations, in particular the partiality of function `pred`, which cannot be applied to `zero`. This example illustrates that order-sortedness is convenient in specification. But order-sortedness leads to a number of problems.

The sort of `pred(X)` depends on the value of `X`. If `X` is greater than `succ(zero)`, then the least sort of `pred(X)` is `NzNat`, the least sort of `pred(succ(zero))` is `Nat`. In this example order-sortedness is used to model a problem typically occurring at run time and, thus, we cannot determine the least type statically any more. This application of order-sortedness is an example where order-sortedness is used to avoid having to deal with applications of functions which are either undefined or yield errors.

In an object-oriented (typed) language a class hierarchy is modeled by a sort hierarchy. In Maude, the inheritance relation induces a subsort relation: the sort of the heir is a subsort of the sort of the ancestor. The concept of order-sortedness is used to deal with partiality. All methods of the supersort are defined for the subsort as well. For the subsort there may be additional operations. Thus, subsorting is an important principle in object orientation. Let us now introduce properties of order-sorted specifications and discuss them subsequently.

Unbridled order-sortedness imposes several problems.

- Regularity (see Def. 3.18) ensures that the term algebras are initial.
- Locally upwards filteredness (see Def. 3.18) ensures that equational satisfaction is closed under isomorphism.

Definition 3.18 (Properties of order-sorted signatures) [HN96] Let $\Sigma = (S, \leq, OP)$ be an order-sorted signature.

- An order-sorted signature Σ is *monotonic* iff for all w_1, w_2, s_1, s_2, op

$$op \in OP_{w_1,s_1} \cap OP_{w_2,s_2} \text{ and } w_1 \leq w_2 \text{ implies } s_1 \leq s_2.$$

- An order-sorted signature Σ is *regular* if it is monotonic and for all $w \in S^\star$ and for all $op \in \{OP_{w,s} \mid w \in S^\star, s \in S\}$ the set

$$R_\Sigma(op, w) =_{\text{def}} \{u \mid u \in S^\star, w \leq u, (\exists s : s \in S : op \in OP_{u,s})\}$$

  is either empty or has a least element.

- An order-sorted signature Σ is *locally upwards filtered* if $(S, \leq)$ is locally upwards filtered, i.e., any two connected sorts s_1 and s_2 have a common supersort.
- An order-sorted signature Σ is *coherent* iff it is regular and locally upwards filtered.

We are interested in specifications of coherent signatures, as we demonstrate later.

Definition 3.19 (Monotonic Σ-algebra) A Σ-algebra A is *monotonic* if $w_1 \leq w_2$, $s_1 \leq s_2$ and $op \in OP_{w_1,s_1} \cap OP_{w_2,s_2}$ implies that $op^A_{w_1,s_1} : A_{w_1} \to A_{s_1}$ equals $op^A_{w_2,s_2} : A_{w_2} \to A_{s_2}$ on A_{w_1}.

Proposition 3.20 (Properties of specifications) For regular signatures, Σ, any term has a least sort and the initial Σ-algebra can be defined as a term algebra.

Given a coherent signature Σ and isomorphic, monotonic Σ-algebras A and B, A satisfies an equation $(\forall X)t = t'$ if and only if B does.

Proof. [HN96].

□

Let us give an example of two specifications of lists. The signature of the first specification is not coherent, while the second is. We discuss the effects of the lack of coherence in the specification.

Example. (Specifications of lists) In the first specification an element of a list is considered to be a one-element list. To model this, `Elem`, the sort of elements of lists, is a subsort of `List`.

```
module LIST-WITH-SUBSORTING {
  signature {
  [Elem < List]

  op eps : -> List
  op _ _ : Elem List -> List
  op _ _ : List Elem -> List
  }

  axioms {
    vars E E1 E2 : Elem
    var  L       : List

  eq [Eq1]:  E eps = E .
  eq [Eq1]:  eps E = E .
  eq [Eq2]:  E1 (L E2)  = (E1 L) E2 .
  }
}
```

Let us have a look at what happens if we extend `LIST-WITH-SUBSORTING`. Let `f` be an overloaded function symbol:

```
  op f : Elem -> R1
  op f : List -> R2
```

Then, for any element `E`, `f(E)` yields two different result types, depending on whether `E` is treated as a list, i.e., `f : list` $\rightarrow$ `R2` is chosen, or `E` is treated as an element, i.e., `f : Elem` $\rightarrow$ `R1` is chosen. Thus, the least type of `f(E)` cannot be determined uniquely.

This example illustrates that, in the presence of subsorting, the union operation, in general, does not preserve the coherence of signatures.

The next specification, `LIST`, uses overloading, not subsorting to implement lists. Note that this specification has been used throughout Chap. 2 and has been given before in Sect. 2.1.2.

```
module LIST {

  signature {
    [Elem]
    [List]

    op eps : -> List
    op _ _ : Elem List -> List
    op _ _ : List Elem -> List
  }
```

```
  axioms {
    vars E E1 E2 : Elem
    var  L : List

    eq [Eq1]:  E2 eps = eps E2 .
    eq [Eq2]:  E1 (L E2)  = (E1 L) E2 .
  }
}
```

This example uses overloading to implement lists, but not subsorting, and is much more benign in inheritance of properties.

When we analyze the first specification of lists, `LIST-WITH-SUBSORTING`, we note that its signature is not coherent, and this induces several problems, as, e.g., in determining the least type. But let us give another example of specifications, and extensions of specifications and investigate what happens. Assume we have a specification `NAT` of natural numbers, which is extended twice: (1) to a specification containing lists of natural numbers, specified according to `LIST-WITH-SUBSORTING`, and (2) to a specification containing integers:

$$
\begin{array}{rcl}
\Sigma_1 & = & \{\ \texttt{[R1 Nat NatList]} \\
 & & \ \ \texttt{[Nat < NatList]} \\
 & & \ \ \texttt{op f : NatList -> R1}\ \} \\
\Sigma_2 & = & \{\ \texttt{[R2 Nat Int]} \\
 & & \ \ \texttt{[Nat < Int]} \\
 & & \ \ \texttt{op f : Int -> R2}\ \}
\end{array}
$$

sharing the common subsignature

$$\Sigma_0 \ = \ \{\ \texttt{[Nat]}\ \}$$

Then the union of Σ_1 and Σ_2, more precisely, the pushout of the inclusions

$$
\begin{array}{rcl}
\sigma_1 : \Sigma_0 & \hookrightarrow & \Sigma_1 \\
\sigma_2 : \Sigma_0 & \hookrightarrow & \Sigma_2
\end{array}
$$

is

$$
\begin{array}{rcl}
\Sigma & = & \{\ \texttt{[R1 R2 Nat Int NatList]} \\
 & & \ \ \texttt{[Nat < Int]} \\
 & & \ \ \texttt{[Nat < NatList]} \\
 & & \ \ \texttt{op f : Int -> R1} \\
 & & \ \ \texttt{op f : NatList -> R2}\ \}
\end{array}
$$

Σ is not locally upwards filtered and the overloaded function `f`, with `f : Int -> R1` and `f : NatList -> R2` yields two different results.

$\diamondsuit$

`NAT` is a coherent specification and, although the two extensions are coherent as well, the combination of the two signatures lacks coherence. Thus, we have to impose properties not only on the signatures of specifications, but also on the mappings between specifications.

Let us first explain *pushouts*, which we use to model the specification-building operations In particular, pushouts are used to model the enrichment of specifications with new sorts, subsort relations and operation symbols. We explain the construction with the examples of signature morphisms, which we use in this section. We consider here the category OSCAT, the category of order-sorted signatures and signature morphisms preserving overloading (see Def. 3.21). Let Σ_0 be a signature and $\sigma_1 : \Sigma \to \Sigma_1$ and $\sigma_2 : \Sigma_0 \to \Sigma_2$ two signature morphisms. The "union" of the two signatures Σ_1 and Σ_2 is defined by a pushout. The pushout identifies the part of the signature which is common to Σ_1 and Σ_2, namely Σ_0. The pushout of Σ_0, σ_1 and σ_2 is the signature Σ together with two signature morphisms $\sigma_1' : \Sigma_1 \to \Sigma$ and $\sigma_2' : \Sigma_2 \to \Sigma$ such that for all x_1, x_2 such that $\sigma_1'(x_1) = \sigma_2'(x_2)$ there exists a z such that $x_1 = \sigma_1(z)$ and $x_2 = \sigma_2(z)$.

The example specification `LIST-WITH-SUBSORTING` and the analysis of the pushout can also be found under the specification name `LIST` in [HN96].

First we would like to sum up the observations we have already made in the examples [HN96]:

- Monotonicity, regularity, locally upwards filteredness and coherence of order-sorted signatures are not always preserved by pushouts in the category OSCAT.

More examples for signatures and mappings between signatures, which do not preserve these properties, can be found in [HN96].

To ensure that the pushout of monotonic, regular, locally upwards filtered or coherent signatures has the respective properties as well, the signature morphisms have to have certain properties, which we define here. The results on preservation of properties can be found in Thm. 3.22.

Definition 3.21 (Properties of mappings) [HN96] Let Σ, and Σ' be order-sorted signatures and $\sigma : \Sigma \to \Sigma'$ an order-sorted signature morphism.

1. A signature morphism $\sigma : \Sigma \to \Sigma'$ *preserves overloading* if $\sigma_{w,s}(op) = \sigma_{w',s'}(op)$ for all $w, w' \in S^\star$, $s, s' \in S$ and all $op \in OP_{w,s} \cap OP_{w',s'}$.

   A specification morphism *translate* _ *with* $\sigma^\star$ preserves overloading if σ preserves overloading.

2. Let Σ, and Σ' be regular and σ preserve overloading. σ is called *regular* if, provided $R_{\Sigma'}(\sigma(op), \sigma^\star(w))$ is not empty, then also $R_\Sigma(op, w)$ is not empty and the least element of $R_{\Sigma'}(\sigma(op), \sigma^\star(w))$ is determined by the equation $\min(R_{\Sigma'}(\sigma(op), \Sigma^\star(w))) = \sigma^\star(\min(R_\Sigma(op, w))$, for all $w \in S^\star$ and $op \in \cup OP$.

3. An embedding $\sigma : \Sigma \to \Sigma'$ is called a *single inheritance extension* if and only if, for $s \in S$, the set $\{t \mid t \in S, s \leq \sigma(t)\}$ is either empty or has a least element in S.

   Let $(S, \leq)$ be a partially ordered set and $A \subseteq S$. The *downward closure* $DOC_S(A)$ is given by $DOC_S(A) = \{s \mid s \in S, (\exists a : a \in A : s \leq a)\}$

   Let Σ and Σ' be regular signatures. Then a regular signature morphism $\sigma' : \Sigma \to \Sigma$ is called *strongly regular* if $\min(R_{\Sigma'}(\sigma(op), w)) \in \sigma^\star(S^\star)$, for all $w \in DOC_{S'^\star}(\sigma^\star(S^\star))$, $op \in \cup OP$, $\{w_k \mid 1 \leq\mid w \mid, w_k \in \sigma(S)\}$ and $R_{\Sigma'}(\sigma(op), w) \neq \emptyset$. ($w_k$ is the k-th component of the word w, and $|\, w\,|$ is the length of word w.)

4. Let $(S, \leq)$ and $(S', \leq)$ be partially ordered sets. A monotonic function $f : (S, \leq) \to (S', \leq')$ is called *locally upwards dense* if, for all $x \in S$ and $y \in S'$ such that $f(x)$ and y are connected, there is an upper bound of $f(x)$ and y in $f(S)$.

   σ is locally upwards dense if $\sigma_s : (S, \leq) \to (S', \leq')$ is locally upwards dense.

Theorem 3.22 (Property preservation by pushouts) [HN96] Let Σ_0, Σ_1 and Σ_2 be order-sorted signatures, and $\sigma_i : \Sigma_0 \to \Sigma_i$ be embeddings. Let Σ together with $\tau_i : \Sigma_i \to \Sigma$ be their pushout in OSCAT.

1. Let Σ_0, Σ_1 and Σ_2 be regular, and σ_i be regular embeddings. Then Σ is monotonic.

2. Let Σ_0, Σ_1 and Σ_2 be regular, and let σ_i be regular embeddings and single inheritance extensions. If σ_1 or σ_2 is strongly regular, then Σ is regular.

3. Let Σ_0, Σ_1 and Σ_2 be locally upwards filtered. If σ_1 or σ_2 is locally upwards dense, then Σ is locally upwards filtered.

Proof. [HN96]

□

3.3 Nondeterministic Order-Sorted Specifications

Algebraic specification and algebras are about elements and functions as mappings between these elements. Nondeterminism is an essential property of concurrent systems, but nondeterminism cannot be expressed within an algebra since this would lead to undesired effects in conjunction with equality as a congruence relation. Let us explain this with an example of a specification of sets and a nondeterministic function.

Example.

```
module CHOICE  {
  import {
    protecting (NAT)
  }
```

```
  signature {
  [Term]

  op choice : Nat Nat -> Term
  op some   : Term    -> Nat
  }

  axioms {
    vars A B : Nat

  eq [Eq1]: some(choice(A,B)) = A .
  eq [Eq2]: some(choice(A,B)) = B .
  }
}
```

`some` chooses one of the two components of a `Term`. Equality is symmetric and transitive and, thus, we can infer

$$
\begin{array}{lll}
(\forall \texttt{A},\texttt{B} : \texttt{A,B:Nat} : & & \texttt{A = some(choice(A,B))} \\
& \text{and} & \texttt{B = some(choice(A,B))} \\
& \text{implies} & \texttt{A = B})
\end{array}
$$

This is certainly not a property of the natural numbers and, thus, we are in need of new formal concepts.

$\diamondsuit$

We introduce a relation, the state transition relation, as a new component in an algebra. State transition relations are common in modeling concurrent systems. In our work, we use a labeled transition relation.

Definition 3.23 (Order-sorted relation) Let $\Sigma = (S, \leq, OP)$ be an order-sorted signature and A and order-sorted Σ-algebra. An order-sorted relation $\rightarrow$ is an S-sorted family of relations $(\rightarrow)_{w \in S^\star, w' \in S^\star}$

We use alternative notations for relations. $(c, d) \in R$ and $c \rightarrow_R d$. Here, we omit the index when the relation is clear from the context. In our work, we use a labeled transition relation. R_l is a indexed with the label l. We use also the notation $c \xrightarrow{l}_R d$ and $(c, l, d) \in R$.

The transition relation is specified in transition rules.

Definition 3.24 (Transition rule) Let $\Sigma = (S, \leq, OP)$ be a regular, order-sorted signature. A Σ-*transition rule* is a triple (X, t, t'), where X is a variable set and t, t' are in $T(\Sigma, X)$. We will use the notation $(\forall X)t$`=>`t'.

An order-sorted Σ-algebra (A, R) satisfies a Σ-transition rule $(\forall X)t\texttt{=>}t'$, written $(A, R) \models (\forall X)t\texttt{=>}t'$, iff $v^\star(t)_{LS(t)}, v^\star(t)_{LS(t')} \in R$, for every valuation $v : X \to A$.

(A, R) satisfies a set of transition rules if it satisfies all transition rules in it.

A *conditional* Σ-*transition rule* is a rule $(\forall X)t\texttt{=>}t'$ `if` C, where condition C is a finite set of unquantified Σ-equations and Σ-transition rules involving only variables in X. An order-sorted Σ-algebra (A, R) satisfies a conditional Σ-transition rule $(\forall X)t\texttt{=>}t'$ `if` C if,

- $v^\star(t_i)_{LS(t_i)} = v^\star(t_i')_{LS(t_i')}$ for each equation $(\forall X)t_i = t_i'$ in C and
- $(v^\star(t_j)_{LS(t_j)}, v^\star(t_j')_{LS(t_i')}) \in R$ for each transition rule $(\forall X)t_j\texttt{=>}t_j'$ in C

implies $(v^\star(t), v^\star(t')) \in R$ for any valuation $v : X \to A$.

The transition rules describe an order-sorted state transition relation between elements of the algebra.

We extend the concepts of order-sorted specifications and allow them to contain, additionally, transition rules.

Definition 3.25 (Order-sorted specification) An order-sorted specification (Σ, E, T) consists of an order-sorted signature Σ, and set of first-order equations E and a set of first-order transition rules T.

Analogously, the notion of an algebra has to be extended. The structures we consider consist of an algebra and a relation for modeling the nondeterministic, state-based notions specified by the transition rules.

Definition 3.26 (Transition system) Let Σ be a coherent order-sorted signature. Let A be a term-generated Σ-algebra and $R \subseteq A \times A$ an order-sorted relation. Then (A, R) is called a *transition system*.

We define classes of transition systems

$$\begin{array}{lcl}
Alg(Sp) & = & \{(A,R) | A \in Alg(\Sigma), R \subseteq A \times A, (A,R) \models E \text{ and } (A,R) \models T\} \\
Gen(Sp) & = & \{(A,R) | A \in Gen(\Sigma), R \subseteq A \times A, (A,R) \models E \text{ and } (A,R) \models T\} \\
Mod(Sp) & = & Gen(Sp) \\
I(Sp) & = & \{(A,R) | (A,R) \text{ initial in } Gen(Sp)\} \\
Z(Sp) & = & \{(A,R) | (A,R) \text{ terminal in } Gen(Sp)\}
\end{array}$$

Note that our definitions of homomorphisms are defined for all operation symbols and need not be generalized in this context. Let us explain this on an example. Let R be a relation and $(a, l, b) \in R$. Let h be an order-sorted Σ-homomorphism, $h : (A, R) \to (A', R')$. Then $(h(a), h(l), h(b)) \in R'$.

Thus, the properties of signatures, i.e., monotonicity, regularity and upwards filteredness apply.

Let us review a second notion familiar from algebras: subalgebras. Subalgebras are closed with respect to function application. When we apply this notion to transition systems, a subsystem is closed with respect to outgoing transitions.

Let us have a closer look at the properties of mappings between transition systems (A, R_A) and (B, R_B). For a mapping between transition systems, being a homomorphism is a strong property, since a homomorphism is compatible with the application of functions to create a state and with the transition relation to the state changes. Let *post* denote the successors of a state in a relation, as defined in Sect. 4.3. Let c be a state, $c \in A_{\mathrm{Cf}}$, $c = c_1 c_2$ and $post(c, l) = d$. Let h be a homomorphism. Then

$$
\begin{array}{ll}
& h(post^A(c,l)) \\
= & \quad \{ \text{ homomorphism property of transition systems } \} \\
& post^B(h(c), h(l)) \\
= & \quad \{ \; c = c_1 \; c_2, \text{ homomorphism property } \} \\
& post^B(h(c_1 c_2), h(l)) \\
= & \quad \{ \text{ homomorphism property } \} \\
& post^B(h(c_1) \; h(c_2), h(l))
\end{array}
$$

Thus, a homomorphism as a relation between transition systems forces us to establish a one-to-one mapping at the level of the states. Moreover, ideally one would like *post* to commute with composition. In our specification we have a normal form which employs multiset union, which is comparable to parallel composition in languages like CCS or π-calculus. In CCS the *post* operation and a homomorphism distribute over the choice operation, while in Maude the post operations does not distribute over parallel composition. This is due to the multiset union and the transition rules, in particular synchronous transition rules.

Definition 3.27 (Simulation, bisimulation) Let (A, R) and(A', R') be two transition systems.

A binary relation $\mathcal{S}$ is a *simulation* if $P \mathcal{S} Q$ implies that, if $P \xrightarrow{a} P'$ and a is any action, then, for some Q',$Q \xrightarrow{a} Q'$ and $P' \mathcal{S} Q'$. Relation $\mathcal{S}$ is a *bisimulation* if both $\mathcal{S}$ and $\mathcal{S}^{-1}$ are simulations.

A state Q simulates a state P, or P is simulated by Q, if there is a simulation relation $\mathcal{S}$ such that $P \mathcal{S} Q$.

A transition system (A, R) simulates a transition system (A', R'), or (A', R') is simulated by (A, R), written $(A, R) \precsim_{\mathcal{S}} (A', R')$, or, if $\mathcal{S}$ is not of interest, $(A, R) \precsim (A', R')$, if there is a simulation relation $\mathcal{S}$ for all states in A. (A, R) and (A', R') are bisimilar, written $(A, R) \approx_{\mathcal{S}} (A', R')$ or, if $\mathcal{S}$ is not of interest, $(A, R) \approx (A', R')$, if $(A, R) \precsim_{\mathcal{S}} (A', R')$ and $(A', R') \precsim_{\mathcal{S}^{-1}} (A, R)$. Analogously we use the symbols $\precsim_{\mathcal{S}}$ and $\approx_{\mathcal{S}}$ also to denote simulation and bisimulation relations between states.

Note that these relations are, in terms of [MPW93], early simulation and bisimulation relations. All values are instantiated before the relation is established. An early relation is coarser than a late simulation relation since it does not—like the late simulation relation—take the structure of the program into account. We use the early version for two reasons: (1) it treats instantiation just in the same manner as the rewriting calculus, and (2) at the level of a specification language we are only interested in the properties described by a specification and not in the structure of the specification.

Note that a homomorphism on a transition system is a simulation relation [Jac96c].

3.4 Partiality in Maude

Partiality is inherent in object-oriented languages. The ancestor has typically more methods than the heir. Provided the class hierarchy is modeled by a subtype hierarchy, the methods which are particular to the ancestor are only defined for the subsort which models the ancestor. We encounter three different issues in partiality in Maude:

- In the specification of data types, order-sortedness is introduced to circumvent partiality. Recall specification `NAT` from Sect. 3.1, where sort `NzNat` is introduced to be the sort for the parameter of `pred`.

- The inheritance relation is modeled by a subsort relation. The transitions of the heir classes are only defined for the states of the subsort, not for the supersort which models the ancestor.

- The synchronization code determines whether an object accepts a message or not. A sensible specification prevents methods to be accepted which are implemented by operations which are not defined for the data describing the state. Moreover, in a sensible specification, the synchronization code is total, i.e., the operations necessary to check whether a message is accepted are total.

We have identified three issues relevant for partiality in Maude. Let us discuss the influence of the specification language and the specification style on partiality in specification. We compare two different implementations of an operation `get` of a bounded buffer. Recall rule `G` from specification `BD-BUFFER` of Sect. 2.1.1.

```
rl [G]: (get B replyto R)
        < B : BdBuffer | cont = C E, in = I, out = O, max = M,
                         ATTS >
        => < B : BdBuffer | cont = C, out = O + 1, in = I, max = M,
                            ATTS >
           (to R answer to get is E) .
```

This rule uses a pattern `C E`, which matches only a non-empty list, to determine whether a `get` is accepted. It abstracts (1) from the operations necessary to implement the access and manipulation of the data describing the state and (2) from the partiality of such operations.

Rule `G` demonstrates the abstractness of the rewriting logic and the advantages we gain in specification by using the features of pattern matching. Let is illustrate the role of the specification style with a second, less abstract implementation which uses a specification style that could be implemented easily in a more conventional language. Here, the access and manipulation of the data, in particular the value of attribute `cont`, is implemented by operations:

```
crl [G']: (get' B replyto R)
           < B : BdBuffer | cont = C, in = I, out = O, max = M,
                            ATTS >
           => < B : BdBuffer | cont = C', in = I, out = O + 1, max = M,
                               ATTS >
              (to R answer to get is E)
          if     sd(I,O) > 0
             and removefirst(C)  == C'
             and first(C)        == E .
```

where the functions `removefirst` and `first` are only defined for non-empty lists:

```
eq removefirst(L E) = L .
eq first(L E)       = E .
```

A message `get` is accepted when the value of `I-O` is greater than `zero`. The values of the resulting states are computed by using the two operations `removefirst` and `first`. A specification has to take care that these two operations are defined for the data, i.e., the value of attribute `cont`. However, this specification is not sensible. Assume the state of the bounded buffer is inconsistent. Then it might happen that `removefirst` and `first` are applied to empty lists.

When comparing rule `G'` to rule `G`, we notice that rule `G'` yields much more undesired partiality. To deal with this partiality in a specification, we could use subsorting to distinguish empty and non-empty lists. This subsorting of the data types describing the state of an object can be lifted to the class an object belongs to and leads then to the states-as-classes approach.

The two implementations of the state change induced by a message `get` demonstrate that the specification style is important in how far partiality of operations in a Maude specification is relevant. Let us compare the partiality of a Maude specification to the partiality in the specification of a data type. In the discussion of the two transition rules, we have explained how the two rules circumvent the occurrence of partiality. In specification `NAT` in Sect. 3.1, we discuss why we have to introduce subsorting to deal with partiality. Thus, at the level of specifications of data types and specifications of transitions, partiality is very different. Crucial for this is the difference of applications of functions and applications of transition rules: in the application of a function it has to be ensured that the function is defined for its parameters. This is done by subsorting. A transition rule is only applied if the pattern matches. A transition rule may be applied and operations may be executed on the data types. Thus, while a function *must* be defined for its parameters, a transition *may* be possible for a state. Since the transition rules are for us the top level of a specification, a sensible specification in which only transitions are possible whose operations on data types are defined allows us to neglect partiality at the level of data types. Moreover, if a specification takes advantage of pattern matching instead of using operations on data types, dealing with partiality can be circumvented at least to some extent.

Let us close this section with a review of, how the different specifications of bounded buffers deal with the partiality of `put` and `get`. Up to now, we have used two different models of a bounded buffer: one in which all states have the same sort, e.g., `BD-BUFFER` in Sect. 2.1.1, and one which uses states as classes, `BUFFER-WITH-STATES` in Sect. 2.2.1. Later we present an algebraic specification of a bounded buffer.

Specifications `BD-BUFFER` and `BUFFER-WITH-STATES` specify the respective transition systems. In both specifications, synchronization code prevents that messages are accepted when the operations on the states cannot be executed. As described above, partiality of the operations on the data is not an issue in the specification.

Let us give an algebraic intra-object specification using the same class hierarchies of a bounded buffer as in the Maude specifications. An algebraic class specification which uses a flat class and sort hierarchy like, e.g., `BD-BUFFER`, would not be acceptable since the operations `put` and `get` are partial. Let us give the signature of an intra-object specification `BUFFER-WITH-STATES` translated to an algebraic specification:

```
module BUFFER-WITH-STATES-ALGEBRAICALLY {
  import {
    protecting (NAT)
    protecting (LIST)
  }

  signature {
    [Elem]
    [BdBuffer]
    [EmptyBdBuffer < BdBuffer]
    [NormalBdBuffer < BdBuffer]
    [FullBdBuffer < BdBuffer]


    op  put : Elem EmptyBdBuffer  -> NormalBdBuffer
    op  put : Elem NormalBdBuffer -> BdBuffer
    op  top : NormalBdBuffer      -> Elem
    op  top : FullBdBuffer        -> Elem
    op  pop : Elem NormalBdBuffer -> BdBuffer
    op  pop : Elem FullBdBuffer   -> NormalBdBuffer
    op  max : BdBuffer            -> NzNat
  }
}
```

Thus, we have here a specification containing three overloaded function symbols: `put`, `pop` and `top`. Note that this specification is coherent.

In Sect. 3.6.1 we introduce a specification `ALG-BD-BUFFER` of a bounded buffer which uses polymorphism, not overloading, to implement the bounded buffer.

Our three different implementations of bounded buffers demonstrate that our specification language provides different specification styles to cope with partiality. The specification `BUFFER-WITH-STATES` does not take advantage of inheritance but has a reasonably simple class hierarchy. The class hierarchy of specification `ALG-BD-BUFFER`, which uses inheritance, is much more complicated. A clear disadvantage of specification `ALG-BD-BUFFER` is that static typing is not possible. Although all three specifications do not have a strong type system, this problem becomes more serious with a higher number of classes. However, this is inherent in all specifications or programs using order-sortedness to deal with run-time issues in partiality.

3.5 Behavioral Specifications

We deal with structures (A, R), where A is an algebra and R a relation between elements of the algebra. An algebra is a model of an algebraic specification, a relation a model of a nondeterministic specification. In [Jac96a, Jac96b, Jac96c] relations are presented as models of coalgebraic specifications of objects. Coalgebraic specifications resemble some of the ideas and concepts of behavioral specifications [BHW95, HS95].

The characteristic properties an algebraic specification describes are how the elements of an algebra are constructed and which of the elements are equal. A coalgebraic specification describes observable properties and does not deal with the construction of the elements. In this setting, equality of elements is "observable equality", a notion which is coarser than the "equality". This notion of equality is similar to behavioral equality in [BHW95, HS95].

We are interested in the construction of (collections of) objects. What we gain from the coalgebraic style is the concept, how the encapsulation, the abstraction from the implementation details of the state of objects should be modeled, namely by a bisimulation relation.

In this section we use material from [BHW95] to relate equality and observable equality.

The observable equality of elements of an algebra is modeled by a partial congruence relation. Elements of the algebra that are in the partial congruence relation are observable and elements at are actually related by the partial congruence relation are observably equal.

Definition 3.28 (Partial congruence) A partial Σ-congruence of a Σ-algebra A is a family $(R_s)_{s\in S}$ of binary relations $R_s \subseteq A_s \times A_s$ such that each R_s is symmetric and transitive and such that R is compatible with the signature, i.e., $a_1 R_{s_1} b_1 \ldots a_b R_{s_n} b_n$ implies $f(a_1, \ldots a_n) R_s f(b_1, \ldots b_n)$ for all $f : s_1 \ldots s_n \to s \in OP$

The properties that one would like to observe determine the partial congruence. In the specification of concurrent systems the bisimulation relation relates states that are indistinguishable via observations.

Lemma 3.29 Let A be a Σ algebra and $R = (R_s)_{s\in S}$ a family of binary relations $R_s \subseteq A_s \times A_s$. R is a partial Σ-congruence on A if an only if there exists a unique subalgebra B of A such that $R_s \subseteq B_s \times B_s$ for all $s \in S$ and R is a Σ-congruence on B.

Proof. [BHW95].

□

The observations modeled by a partial Σ-congruence define for us an algebra again. Thus we do not have to care about the unobserved elements of the algebra anymore.

Example. (Observational equivalence of bounded buffers) Assume that we have a specification which implements a bounded buffer with a state having attributes `in`, `out`, `max` and `cont`. The values of `in` and `out` are used to implement whether a bounded buffer accepts a `put` or a `get` message, but there are no messages that trigger a state change in which the message is consumed. Thus, the observations of a bounded buffer are whether it accepts a `put` or `get` message and the result a `put` message `yields`.

We define the partial congruence R by:

```
< B : BdBuffer | in = I, out = O, max = M, cont = L >
```
R_{BdBuffer}
```
< B' : BdBuffer | in = I', out = O', max = M', cont = L' >
```

$$\begin{array}{lll}
 & \text{iff} & \mathtt{I} - \mathtt{O} = \mathtt{I'} - \mathtt{O'}, \mathtt{B} = \mathtt{B'}, \mathtt{M} = \mathtt{M'}, \mathtt{L} = \mathtt{L'} \\
a \; R_s \; b & \text{iff} & a =_s b \text{ for all } a, b \in A_s, s \notin \text{BdBuffer}_{\text{Cf}} \\
c_1 \; c_2 \; R_{\text{Cf}} \, d & \text{iff} & \text{if } d = d_1 d_2 \text{ such that } c_1 R_{\text{Cf}} \, d_1 \text{ and } c_2 R_{\text{Cf}} \, d_2
\end{array}$$

Two bounded buffers are behaviorally equivalent—more precisely, two states are behaviorally equivalent—if the difference between the attributes `in` and `out` and the values of attributes `max` and of attributes `cont` are equal.

Two values are observably equal if they are equal. Two messages are observably equal when they are equal and two configurations are observably equal if the objects and the messages they consist of are observably equal.

◇

Let us discuss the implications of behavioral equivalence for our specifications. The way objects are phrased and treated in Maude is quite unusual from an object-oriented point of view. In particular, one might criticize that the state is visible and accessible to an observer and to other objects in a synchronous transition rule. Thus, instead of considering the notation `< B : BdBuffer | in = I, out = O, max = M, cont = L >` as a term, which is isomorphic to a record with corresponding values, one can interpret it as a state which is observed by methods yielding the object identifier, the class identifier, and methods `in`, `out`, `max`, `cont`, which yield the respective values. The methods which are used to observe a bounded buffer, but which do not change its state, are always accepted by an object. In this view, the objects and their states are observable, but the state of the global system is accessible. Thus, two configurations are observably equal if their state is constructed from observably equal objects and messages.

Note that the behavioral equivalence and the partial congruence that we introduce here is modeling encapsulation. "Inside" an object the state, the way it is constructed, is visible.

This is the algebraic view. From the outside only behavioral equivalence is observable, due to the principle of encapsulation. The behavioral equivalence is modeling precisely the abstraction induced by encapsulation.

3.6 Specification Styles: Algebraic → Coalgebraic → Maude

The question " What is object-oriented specification of distributed systems?" has different answers—depending on the point of view from which it is answered. We have introduced three specification styles: algebraic and coalgebraic specification and Maude. Each one is a different answer to this question.

- The algebraic style addresses deals with the intra-object level, namely, the specification of the state, the operations on the state and the properties of the methods.
- The coalgebraic style describes how an object can be observed.
- Maude covers the inter-object level. Here, one deals with the synchronization and communication of objects.

We demonstrate the issues of object-oriented specification at these three different levels of specification and comment on the relations between the different levels. We show how encapsulation can be modeled. We discuss Maude's object-model and, in particular, the criticism of Maude that is does not support encapsulation.

What we achieve the ability to specify a distributed system beginning from the data and operations stored "deep" inside the objects, on to the observations of objects and further on to the global state. This complements the view and expressiveness provided by Maude, and it adds a new dimension to the specification style and reuse principles developed in Sect. 2, where we deal with design and reuse constructs and specification methods at the level of Maude.

In this section, we illustrate the formal framework with the example of a bounded buffer, which "understands" five methods: `put`, `length`, `max`, `top` and `pop`. A bounded buffer accepts a `put` only if it is not full, a `top` and a `pop` only if it is not empty. A bounded buffer accepts always `max` and `length`. `max` yields the capacity of the bounded buffer and `length` yields the number of elements stored. In the figures and examples, our bounded buffer has a capacity of 2.

We have decided to use two methods, `pop` and `top`, instead of `get` simply for technical reasons. `get` yields a result and changes the state. This would make it necessary to deal with tuples in the algebraic and coalgebraic specification. We avoid this by using `pop`, which changes the state, and `top`, which yields a result.

Let us recall what we would like to achieve with our specifications and the different specification styles. Our goal is to find a language appropriate for object-oriented specifications of distributed systems.

All three specification styles could be employed together to specify a system. In this case, the specifications have to comply in some sense in order to ensure consistency. However, this aspect is secondary for us and, thus, we do not study the relations between the specifications in detail. We use the specification styles to reason about the object model and how the typical object-oriented concepts like, e.g., inheritance can be implemented.

In this section, we discuss on the example of the bounded buffer for each of the three specification styles, assets, drawbacks, properties that are described in the specification and properties that can be deduced. Fig. 3.2 depicts the three specification styles with the part of the algebras and transition systems considered later in Demonstranda 3.30 and 3.34.

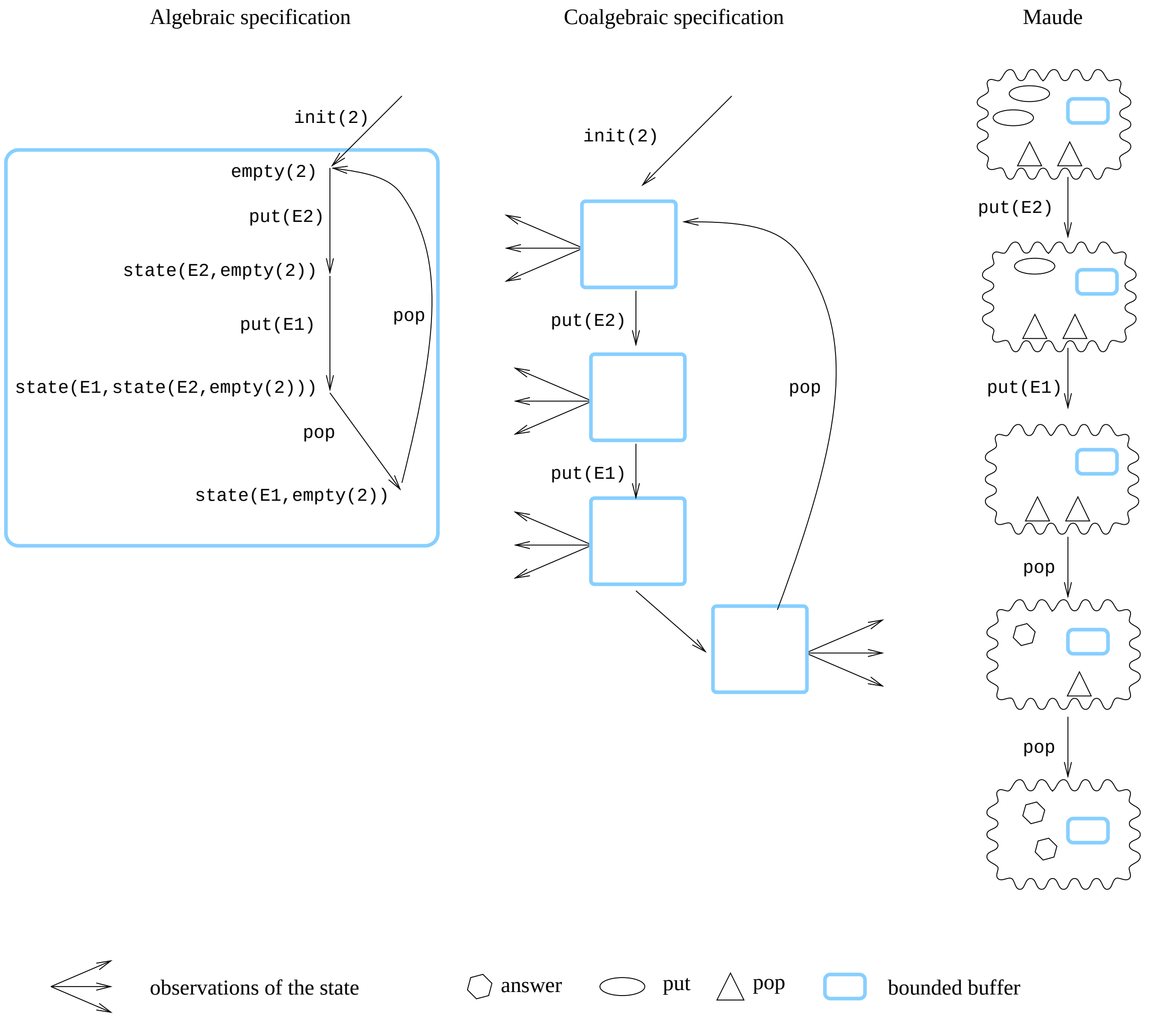

Figure 3.2: Specification of the bounded buffer

3.6.1 Algebraic Specification

In the algebraic specification, we describe the data a bounded buffer stores, the state of the bounded buffer and the operations on the state. The state of a bounded buffer is constructed by three operations, `init`, `state` and `full`. Let us explain these operations. `init` creates a bounded buffer. Its argument is the buffer's capacity. Note that a bounded buffer which is created by `init` is of sort `EmptyBdBuffer`. Operation `state` stores one element in the buffer. The operator `full` is applied to mark a full bounded buffer, in which case it cannot accept any more method invocations. Note that a full buffer is of sort `FullBdBuffer`.

A bounded buffer accepts five methods. `pop` and `top` are only accepted if the buffer is not empty, i.e., in state `NEBdBuffer`; `put` is only accepted if the buffer is not full, i.e., in state `NFBdBuffer`; `max` and `length` are accepted at any time. Note that the class hierarchy provides us with reuse, in contrast to the class hierarchy used in specification `BD-BUFFER-WITH-STATES` in Sect. 2.2.1. The class hierarchy used to implement the bounded buffer is depicted in Fig. 3.3.

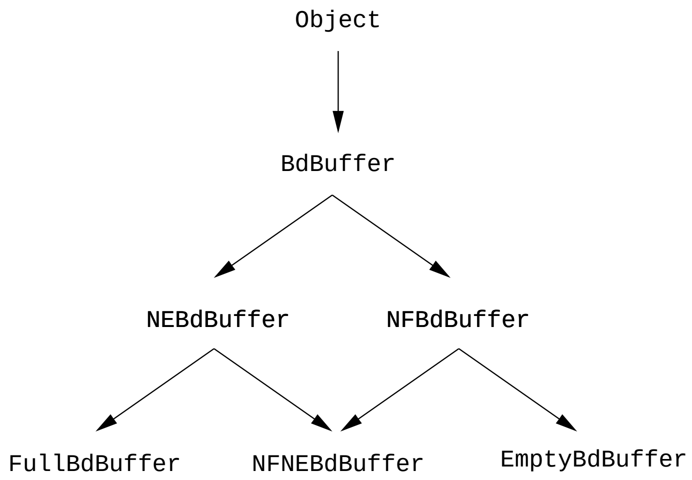

Figure 3.3: Class hierarchy to model a bounded buffer

```
module ALG-BD-BUFFER-SIMPLE  {
   import {
     protecting (LIST-ENRICHED)
   }

   signature {
     [BdBuffer]

     [NFBdBuffer < BdBuffer]                --      Non-Full BdBuffer
     [NEBdBuffer < BdBuffer]                --     Non-Empty BdBuffer
```

```
  [FullBdBuffer < NEBdBuffer]           --          Full BdBuffer
  [EmptyBdBuffer < NFBdBuffer]          --         Empty BdBuffer

  [NENFBdBuffer < NEBdBuffer NFBdBuffer] -- Neither Empty nor Full

                                        --    State of a BdBuffer
  op init   : NzNat           -> EmptyBdBuffer
  op state  : Elem NFBdBuffer -> NENFBdBuffer
  op full   : NEBdBuffer      -> FullBdBuffer

                                        --  Methods of a BdBuffer
  op max    : BdBuffer        -> NzNat
  op length : BdBuffer        -> Nat
  op put    : Elem NFBdBuffer -> NEBdBuffer
  op top    : NEBdBuffer      -> Elem
  op pop    : NEBdBuffer      -> BdBuffer
}

axioms {
  var  M    : NzNat
  var  N    : Nat
  vars E E' : Elem
  vars X    : BdBuffer
  vars NEX  : NEBdBuffer
  vars NFX  : NFBdBuffer

eq [max1]: max(init(M))      = M .
eq [max2]: max(state(E,NFX)) = max(NFX) .
eq [max3]: max(full(NEX))    = max(NEX) .

eq [length1]: length(init(M))      = 0 .
eq [length2]: length(state(E,NFX)) = 1 + length(NFX) .
eq [length3]: length(full(NEX))    = length(NEX) .

ceq [put1]: put(E,NFX) = state(E,NFX)
                      if length(NFX) + 1 <  max(NFX) .
ceq [put2]: put(E,NFX) = full(state(E,NFX))
                      if length(NFX) + 1 == max(NFX) .

eq [top1]: top(state(E,state(E',NFX))) = top(state(E',NFX)) .
eq [top2]: top(state(E,init(M)))       = E .
eq [top3]: top(full(NEX))              = top(NEX) .
```

```
   eq [pop1]: pop(state(E,state(E',NFX))) = state(E,pop(state(E',NFX))) .
   eq [pop2]: pop(state(E,init(M)))       = init(M) .
   eq [pop3]: pop(full(NEX))              = pop(NEX) .
   }
}
```

The carrier sets, with the mappings between them, are depicted in Fig. 3.4. For simplicity we have omitted the carrier sets of the parameter sorts of the mappings.

Let us discuss several issues of specification ALG-BD-BUFFER-SIMPLE. Note that its signature ALG-BD-BUFFER-SIMPLE is not coherent.

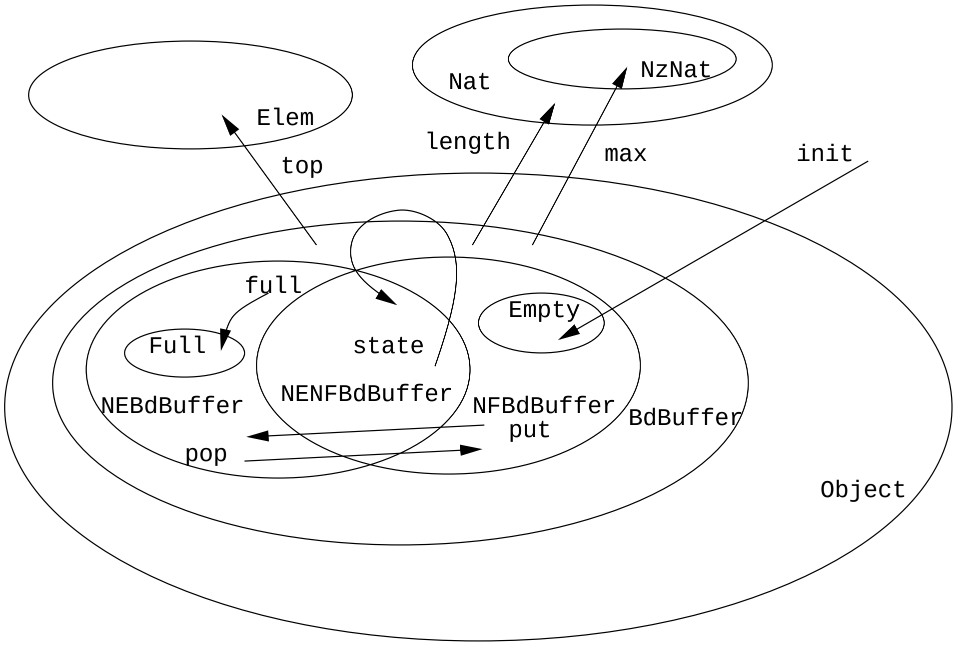

Figure 3.4: Class hierarchy and carrier sets with mappings

The properties that are specified in specification ALG-BD-BUFFER-SIMPLE are equations, in particular, equations between states of the bounded buffer. The two demonstranda 3.30 and 3.31 are examples for such properties.

Demonstrandum 3.30
ALG-BD-BUFFER-SIMPLE $\models$ pop(pop(put(E1,put(E2,init(2))))) = init(2) for all E1, E2

Proof.

```
  pop(pop(put(E1,put(E2,init(2)))))
=    { put1 }
```

```
  pop(pop(put(E1,state(E2,init(2)))))
=    { put1 }
  pop(pop(full(state(E1,state(E2,init(2))))))
=    { pop3 }
  pop(pop(state(E1,state(E2,init(2)))))
=    { pop1 }
  pop(state(E1,pop(state(E2,init(2)))))
=    { pop2 }
  pop(state(E1,init(2)))
=    { pop2 }
  init(2)
```

□

Demonstrandum 3.31
`ALG-BD-BUFFER-SIMPLE` $\models$ `pop(put(E,put(E,init(2))))` = `put(E,init(2))` for all `E`

Proof.

```
  pop(put(E,put(E,init(2))))
=    { put1 }
  pop(put(E,state(E,2)))
=    { put2 }
  pop(full(state(E,state(E,init(2)))))
=    { pop3 }
  pop(state(E,state(E,init(2))))
=    { pop1 }
  state(E,pop(state(E,init(2))))
=    { pop2 }
  state(E,init(2))
=    { put1 }
  put(E,init(2))
```

□

Equality is a congruence relation in algebras. Thus, application of methods to equal terms, modeling states, yield equal results or equal successor terms. That is the application of method `length`, respectively method `max`, to the two states proven equal in Demonstrandum 3.31 yield the same results.

Demonstrandum 3.32
`ALG-BD-BUFFER-SIMPLE` $\models$
    `length(pop(put(E,put(E,init(2)))))` = 1 = `length(put(E,init(2)))` for all `E`

Demonstrandum 3.33
`ALG-BD-BUFFER-SIMPLE` $\models$
`max(pop(put(E,put(E,init(2)))))` $= 2 =$ `max(put(E,init(2)))` for all `E`

Let us have a look at the initial computation structure of specification `ALG-BD-BUFFER-SIMPLE`. The elements of the carrier sets are terms as, e.g., for a bounded buffer with capacity 2

```
init(2),
put(E1,init(2)),
state(E1,init(2)) ... .
```

Part of the algebra is depicted on the left side of Fig. 3.2.

3.6.2 Coalgebraic Specification

In the coalgebraic specification, we describe the observations of a class modeling a bounded buffer. We describe the observations by (conditional) transition rules. The observations are the results which the methods `max`, `length`, `top` and `cont` yield. The specification describes the changes in the `observations`, triggered by applications of methods.

In this specification style, we have only a class sort, but no operations to construct elements of the class sort. In Fig. 3.2 we represent this notion by a transition system. The states are elements of an algebra, they have structure but the structure itself is not visible.

```
module COALG-BD-BUFFER-SIMPLE  {
  import {
   protecting (LIST-ENRICHED)
   using (ACZ-CONFIGURATION)
   protecting (RWL)
  }

  signature {
   [X]                         -- the class sort
   [Nat < Object]
   [List < Object]
   [Elem < Object]

   op init      : NzNat  -> X
   op _.max     : X      -> NzNat { l-assoc }
   op _.length  : X      -> Nat   { l-assoc }
   op _.put(_)  : X Elem -> X     { l-assoc }
   op _.pop     : X      -> X     { l-assoc }
   op _.top     : X      -> Elem  { l-assoc }
   op _.cont    : X      -> List  { l-assoc }
```

```
  }

   axioms {
     var  M    : NzNat
     var  L    : Nat
     var  E    : Elem
     var  self : X
     var  C    : List

   rl  [max1]: init(M).max        => M .

   crl [max2]: self .put(E) .max  => M
                    if self .max ==> M .

   crl [max3]: self .pop .max     => M
                    if self .max ==> M .

   rl [length1]: init(M).length => 0 .

   rl [length2]: (self .pop) .length => sd(self .length, 1) .

   crl [length3]: (self .put(E)) .length => (self .length) + 1
                      if  (self .length < self .max) .

   crl [top]: self .top => E:Elem
                   if (self .length < self .max)
                  and self .cont ==> (C:List E:Elem):List .

   rl [cont1] : init(M).cont => eps .

   rl [cont2] : self .put(E) .cont => (E:Elem (self .cont)):List .

   crl [cont3]: self .pop .cont   => C
                   if self .cont ==> (C:List E:Elem):List .
  }
}
```

Note that we observe the behavior but not how the state is constructed. The two functions `full` and `state`, which we use to model the state in the algebraic specification of the bounded buffer, do not appear in this specification.

We observe the bounded buffer with the methods `max`, `length`, `top` and `cont`. The methods `init`, `pop` and `put` change the state. We specify how the observations made by `max`, `length`, `top` and `cont` change when the methods changing the state are applied.

Let us have a look at the properties that are derivable from the specification. Specification `COALG-BD-BUFFER-SIMPLE` is a description of the behavior of a bounded buffer as it is visible from the outside. The state itself is not visible, only observable. Thus, the property of states that we can prove from this specification is behavioral equivalence. We prove the behavioral analogies of Demonstranda 3.30 and 3.31.

Demonstrandum 3.34
`COALG-BD-BUFFER-SIMPLE` $\models$ `init(2).put(E2).put(E1).pop.pop` $\approx$ `init(2)` for all `E1`, `E2`.

Proof. Let us validate this by proving that all the observation functions yield the same result.

`max`:

```
    init(2).put(E2).put(E1).pop.pop.max => M
⇐   { max3 }
    init(2).put(E2).put(E1).pop.max => M
⇐   { max3 }
    init(2).put(E2).put(E1).max => M
⇐   { max2 }
    init(2).put(E2).max => M
⇐   { max.2 }
    init(2).max => M
```

`length`:

```
    init(2).put(E2).put(E1).pop.pop.length => L
⇐   { max3 }
    init(2).put(E2).put(E1).pop.length => L-1
⇐   { max3 }
    init(2).put(E2).put(E1).length => L-2
⇐   { max2 }
    init(2).put(E2).length => L-2+1
⇐   { max2 }
    init(2).length => L-2+2
```

```
cont: init(2).put(E2).put(E1).pop.pop.cont => eps
      init(2).cont => eps
```

`top`: Since `self.top => E if self.length => L and L > 0` is the only rule which allows `top`, a transition triggered by `top` is not possible in both states.

□

Demonstrandum 3.35
`COALG-BD-BUFFER-SIMPLE` $\models$ `pop.put(E).put(E).init(2)` $\approx$ `put(E).init(2)` for all `E`.

Specification `COALG-BD-BUFFER-SIMPLE` describes a transition system. While, in the algebra which is the model of the algebraic specification, `ALG-BD-BUFFER-SIMPLE`, the mappings are undirected, a transition system modeling the changes and the mappings between the states are directed. Both in the algebraic and the coalgebraic model, we obtain "backward" edges like in Demonstranda 3.30 and 3.34 and Fig. 3.2.

The algebraic and the coalgebraic specification describe our running example, the bounded buffer. Thus, the algebra and the transition system they describe are very similar. The difference lies in the properties that they specify. In the algebraic specification, we have equality of bounded buffers, in the coalgebraic specification, behavioral equality of bounded buffers. The coalgebraic specification allows to abstract

- from the implementation of the state: `state` and `full` are not observed, and
- from whether information is stored in or derived from the bounded buffer: our algebraic specification computes the value of `length`; alternative specifications are, e.g., bounded buffers which implement `length` as in specification `BD-BUFFER`.

Sect. 2.5, contains another couple of algebraic and coalgebraic specification of a bounded buffer: specifications `ALG-BD-BUFFER` and `COALG-BD-BUFFER`. For simplicity, we assume, that both specifications contain also a specification of function `length`. The bounded buffer described in `ALG-BD-BUFFER` and `COALG-BD-BUFFER` stores the history. Let us relate these two specifications to the specifications given in this section.

Note that the two demonstranda are not satisfied for specification `ALG-BD-BUFFER`.

Demonstrandum 3.36
`ALG-BD-BUFFER` $\not\models$ `pop(pop(put(E1,put(E2,init(2)))))` $=$ `init(2)` for all `E1`, `E2`

Demonstrandum 3.37
`ALG-BD-BUFFER` $\not\models$ `pop(put(E,put(E,init(2))))` $=$ `put(E,init(2))` for all `E`

However, if we apply the techniques of behavioral specification and consider the states to be equivalent that are behaviorally equivalent by the observations made by `max`, `length`, `cont` then we obtain an equality that allows to identify states in which the difference between the `I` and `O` is equal. We define this by enriching `ALG-BD-BUFFER` to `ALG-BD-BUFFER+` with the equation modeling the behavioral equality of states:

```
ceq empty(M,I,O) = empty(M',I',O')
   if sd(I',O') == sd(I,O) .
```

We obtain then

Demonstrandum 3.38
`ALG-BD-BUFFER+` $\models$ `pop(pop(put(E1,put(E2,init(2)))))` $=$ `init(2)` for all `E1`, `E2`

Demonstrandum 3.39
`ALG-BD-BUFFER+` $\models$ `pop(put(E,put(E,init(2))))` = `put(E,init(2))` for all `E`

Thus, the specification `ALG-BD-BUFFER` can be used to implement an object with a behavior described in specification `COALG-BD-BUFFER-SIMPLE`; in the loose approach, the class of `ALG-BD-BUFFER+` algebras is a subset of the class of `ALG-BD-BUFFER` algebras.

Naturally, specification `ALG-BD-BUFFER-SIMPLE` cannot be used to implement a bounded buffer as described by specification `COALG-BD-BUFFER`—provided we use the standard interpretation of natural numbers.

3.6.3 Maude Specification

The Maude specification deals with the global state of a bounded buffer.

```
module BD-BUFFER-SIMPLE {
  import {
    protecting (NAT)
    protecting (LIST-ENRICHED)
    protecting (ACZ-CONFIGURATION)
  }

  signature {
    class BdBuffer {
      max  : NzNat
      cont : List
    }

    op to _ top              : ObjectId -> Message
    op to _ pop              : ObjectId -> Message
    op to _ put _            : ObjectId Elem -> Message
    op to _ max              : ObjectId  -> Message
    op to _ length           : ObjectId  -> Message
    op answer to top is _    : Elem -> Message
    op answer to max is _    : Nat -> Message
    op answer to length is _ : Nat -> Message
  }

  axioms {
    var B    : ObjectId
    var E    : Elem
    var C    : List
    var M    : NzNat
    var ATTS : Attributes
```

```
    rl [top]: (to B top)
              < B : BdBuffer | cont = C E, ATTS >
              => < B : BdBuffer | cont = C E, ATTS >
                 (answer to top is E) .

    rl [pop]: (to B pop)
              < B : BdBuffer | cont = C E, ATTS >
              => < B : BdBuffer | cont = C, ATTS > .

    crl [put]: (to B put E)
               < B : BdBuffer | cont = C, max = M, ATTS >
               => < B : BdBuffer | cont = E C, max = M, ATTS >
               if M > length(C) .

    rl [length]: (to B length)
                 < B : BdBuffer | cont = C, max = M, ATTS >
                 => < B : BdBuffer | cont = C, max = M, ATTS >
                    (answer to length is (length(C))) .
  }
}
```

A Maude specification describes the computational progress of a collection of objects. In Chap. 2, we have discussed the particular view provided by Maude. Maude's focus is on the inter-object level and the rules access and manipulate the states of the objects. Let us discuss the object model of Maude specifications more closely. The algebraic and the coalgebraic specification describe behaviorally equivalent classes of algebras. Thus, at the inter-object level, it is irrelevant how we interpret a Maude object `< B : BdBuffer | cont = C, max = M >` and Maude rules.

- In an algebraic interpretation, this Maude object is interpreted as an object with a state modeled as a record with two entries `cont` and `max`. The transition rules describe state changes.

- In the coalgebraic interpretation, this object can be observed by two methods: `cont` and `max`. A transition rule describes the changes in the observations which can be made. Thus, the changes in the observations described by transition rule `pop` can be rephrased in a coalgebraic style as follows:

  ```
  self.pop.cont => C if self.cont ==> C E
  self.pop.max  => M if self.max ==> M
  ```

Maude specifications with their particular view can be considered to be "only" an abstract notation for a number of observations made from an object of an more conventional object model. Naturally, the coalgebraic specification style focuses again on the intra-object level and does not specify the relation between messages and methods as aptly as between results of methods and answer messages.

Note that, for equality of Maude specifications, the states of the objects and the messages are relevant. Thus, since the states of the Maude specifications model more information, state transition systems are finer and have more states than transition systems modeling the behavior of single objects (see also Fig. 3.2).

The link between the algebraic and the coalgebraic specification is the bisimulation relation which allows to identify observationally equivalent states. We have studied the relation between these two specification styles at the intra-object level.

We conclude the discussion of specification style and method with the relation of the specification of a bounded buffer used in this section to specification `BD-BUFFER`. A bounded buffer, as specified in `BD-BUFFER`, accepts two messages, `put` and `get`, while the bounded buffer in this section accepts `put`, `pop` and `top` (as well as `max` and `length`, which we neglect for simplicity for a moment). In Sect. 2.4, we have given guidelines for the modeling of objects and, in particular, the messages that they accept. The processing message `get` can be implemented by a sequential composition of a `top` and a `pop` method. The answer message to `get` is just the answer to `top`.

```
[put]    (to B get) = (to B top);(to B pop) .

[answer] (answer to top is E) = (answer to get is E) .
```

In order to prevent a bounded buffer from accepting `pop` and `top` messages, subconfigurations can be used.

3.7 Related Work

We use as models order-sorted algebras with an additional relation to model the state transition relation. We model partiality by order-sortedness. When comparing our approach to OS, TROLL, $\pi o\beta\lambda$ and the language of Jacobs [Jac96c], which are all object-oriented concurrent languages with a formal basis, we note that all approaches differ significantly from each other.

- The language OS [Bre91] employs partial order-sorted specifications and, accordingly, uses partial order-sorted algebras. Thus, OS specifications tend to have a less complicated sort hierarchy than our specifications.

- The semantics of TROLL light (a subset of TROLL) is given by a translation to Maude [DG93]. TROLL specifications are many-sorted specifications and partiality is not considered [Den96b].

- The object-oriented language constructs of OOSpectrum [WNL95] are implemented in SPECTRUM [BFG+93]. SPECTRUM supports partial specifications, not order-sortedness. Thus, this approach is dual to our approach: order-sortedness is modeled as partiality.

- The coalgebraic specifications of [Jac96a, Jac96b, Jac96c, Rei95] are dual to our algebraic approach. Their transition systems are unsorted and partiality is not an issue. They deal only with sequential specifications of single classes and not with communication between objects.

- The approach of $\pi o\beta\lambda$ [Jon92], Pict [PT97] and the object-oriented concurrent language in [Wal91, PW96] is different to our approach. They use the π-calculus to describe the semantics of their languages [Jon93b] by transition systems. Thus, bisimulation and other abstract semantics can be applied to those languages as well. A structural operational semantics for $\pi o\beta\lambda$ is given in [HJ96].

This variety of semantics of object-oriented concurrent languages is somewhat surprising. Each of the few formal, object-oriented languages is based on a different concept. Naturally, the different semantic background of the approaches induces different formal techniques, to which we refer in Chap. 4 and 5. Moreover, since we use a much richer class hierarchy, as we illustrated in Chap. 2, the problem of treating the class hierarchy are much more urgent than in sequential object-oriented languages. The property of coherence of signatures has already been defined in [GM92] for order-sorted algebraic specifications and is required for OS specifications as well [Bre91]. The property of coherence ensures that there is a least sort for each term, and there is a least sort for each application of a function. We do not deal with the typing of object-oriented concurrent languages. However this is a nontrivial problem, as the results of typing of object-oriented sequential languages [PT94, HP95, PS94, PS97] and of concurrent languages like the π-calculus [PS96a, Vas94, VT93] suggest.

Particular to our work is the combination of algebraic techniques and concepts with concepts from concurrency on an algebraic basis. Similarly, the SMoLCS approach, a framework for defining the semantics of concurrent languages in a structured way, uses algebraic concepts for specification of concurrent systems [AW89, Lec92]. We use in our work results on behavioral specifications of [BHW95, HS95] to model bisimulation as behavioral congruence for algebraic data types. Later we will develop other, coarser notions of abstraction in the process of behavioral refinement (see Sect. 5.3).

We conclude this section by relating our approach to Meseguer's approach to defining semantics of Maude. He uses category theory: the elements of the category are the states and the values, the morphisms between the elements are derived from the applications of the rules of the calculus [Mes92a]. These morphisms are so fine, that they are deterministic. Thus, Meseguer deals with an order-sorted, deterministic framework and neither partiality nor properties of signatures and algebras are issues in his approach.

3.8 Remarks and Discussion

In this chapter, we have developed the algebraic part of our formal framework. We have discussed briefly the problems that come along with using order-sortedness. Order-sortedness in object orientation is induced by the class hierarchy and we have decided to use it also to deal with partiality. In concurrent object-oriented languages, partiality is an important concept—much more than in sequential object-oriented languages. Thus, in practice, it is important to know the criteria for which order-sorted signatures and specifications are as unproblematic to handle as many-sorted signatures and specifications.

We have also discussed the specification style and the view provided by Maude. An important advantage of Maude is that it allows to deal with partiality in a very convenient way (if necessary), and to abstract to a very high degree from dealing explicitly with partiality in the specifications, in particular, in the specification of synchronization and communication.

The answer to the question "What is object-oriented specification of distributed systems?" depends on the level at which one view. We distinguish here three different levels: (1) the data and the operations on the data, (2) the observations of a class and (3) the global state with synchronization and communication. With the three according specification styles, we provide three different views to object-oriented specification of distributed systems: in the algebraic specification we deal with the data and the operations on the data, in the coalgebraic specification with the observations of a class and in Maude with the behavior of collections of objects.

The focus of Maude specifications is at the inter-object level. We offer two interpretations of Maude objects (1) an algebraic interpretation, where a Maude object describes an object with its state being visible and accessible (2) a coalgebraic, where a Maude object describes the observations of an object. This illustrates the particular approach of Maude; Maude abstracts from all the "object-oriented stuff" of the objects and classes. The visibility of the state of objects gives rise to criticism since it seems that encapsulation, the most important principle in object-orientation, is not supported. We argue that Maude is truly object-oriented, since it respects the principle of encapsulation—provided one chooses a coalgebraic interpretation of Maude objects.

Important for us is the way encapsulation is modeled, namely, as the largest bisimulation relation according to [Jac96c]. This is somewhat surprising, considering the complexity of other approaches in modeling the link between the intra- and inter-object level. Moreover, it suits nicely the formal techniques of verification and refinement, where bisimulation also plays an important role.

Let us discuss what we have achieved up to now with respect to modeling distributed systems in an object-oriented language. We are able to specify a distributed system from the inside of the objects up to the global state and behavior, to structure and to reuse such a specification in an object-oriented way. Yet this is not sufficient in order to demonstrate that we have a well-designed language and powerful concepts. In the next two sections, we deal with verification and refinement and we will establish the object model and the constructs of reuse at a more property-oriented, abstract level.

Chapter 4

Properties and Verification

Maude is a an expressive and abstract specification language, as we have demonstrated in Chap. 2. However, the way the behavior of objects is described is operational and the transition rules describe only single steps of the objects involved. While its operational and modular specification style makes Maude a very simple language, one would like to express the properties of single objects and collections of objects in a direct way. Thus, we use a more property-oriented notation, the μ-calculus [Bra92, Koz83, Sti92], and describe the verification of μ-properties for Maude specifications.

Let us motivate our choice of the μ-calculus for reasoning about Maude specifications. μ-calculus formulas characterize bisimilar processes [SI94]. I.e., two (finitely branching) processes satisfy the same set of μ-formulas if and only if they are bisimilar. We have already used bisimulation and simulation relations are to model encapsulation (see Sect. 3.6). In this section we use this duality twice:

1. We use the duality of bisimulation relations and properties in the μ-calculus to provide abstraction mechanisms in the verification of object-oriented systems. Such abstraction mechanisms are paramount for the verification of Maude specifications and object-oriented concurrent systems in general. Let us explain this in more detail. According to the object-oriented design philosophy, a program consists of a collection of objects and messages. The objects are often small entities to which techniques of verification can be applied that deal with the properties of data and the operations on the data. Verifying the properties of a possibly large collection of objects demands the use of abstraction techniques. This abstraction is provided by a special kind of simulation and bisimulation relations. At the inter-object level one is interested in the way objects communicate and synchronize and in the messages that are consumed and produced. It is rather irrelevant that the units which communicate are objects. Accordingly, the approach of abstract interpretation [LGS+95], which we employ here, has been developed for the verification of concurrent processes.

2. We use these mappings, which are induced by simulation and bisimulation relation to capture as concepts the relations between classes of algebras which correspond to the reuse via the constructs inheritance, subconfiguration and message algebra. We

characterize which properties, phrased in the μ-calculus are inherited via these reuse concepts and constructs.

Our motivation for dealing extensively with the inter-object level of Maude stems from the abstract object model of Maude and, in particular, from its flexible synchronization and communication patterns. Thus, the properties of single objects as well as of collections of objects is of interest and is subject to verification.

The verification of a complex system is cumbersome. The abstraction techniques help to reduce the complexity of this task. We use the techniques of abstraction and verification to reason about the properties that are inherited via the reuse constructs that we have developed in Chap. 2. With these results on inheritance of properties, proofs of properties of systems can be modularized and inherited. The inheritability of properties is particularly important at the abstract, property-oriented level, since one would like to inherit the properties, not the code of a specification.

This chapter is organized as follows. We start with a special class of safety properties of states, the so-called configuration and robustness invariants. We use these invariants to exclude states from the state space of the algebras that we do not consider to be sensible. In a second step, we develop a framework for abstraction and verification. We apply it to several small examples and prove a mutual exclusion property for airports. In a third step, we investigate another approach coping with the complexity, namely structured verification. Hereby, we identify classes of properties that are inheritable via the reuse concepts inheritance, subconfigurations and message algebras.

4.1 The μ-Calculus

The μ-calculus is used to reason about state transition systems at a property-oriented level [Koz83, Sti92, Bra92]. The language of μ-formulas consists of propositions, for reasoning about states, the modal connectives, quantifiers and fixpoint operators.

Our language of propositions for a specification is given by the grammar:

$$p ::= \texttt{tt} \mid \texttt{ff} \mid \neg p \mid \text{“}o\text{”} \mid \text{“}m\text{”}$$

where o, respectively m, is a term over a signature Σ representing an object respectively a message. We denote with $P(\Sigma)$ the basic propositions with “o” and “m” using terms of signature Σ.

According to the two interpretations of specifications, the algebraic and the coalgebraic there are two interpretations of the propositions “o” and “m”.

- In the algebraic interpretation, the double quotes around an object or message represent the proposition “this object exists” or “this message exists”. E.g., state C satisfies “`< B1 : BdBuffer | in = 1 >`” if one of its elements is an object with object identifier `B1` belonging to class `BdBuffer` (which includes all subclasses of `BdBuffer`) whose value of attribute `in` is equal to `1`.

- In the coalgebraic interpretation, the double quotes around an object or message indicate observation. E.g., state C satisfies "`< B1 : BdBuffer | in = 1 >`" if a bounded buffer with object identifier `B1` is observed via the method, `in` which yields as a result the value `1`.

Note that the use of negation is restricted to basic propositions.

Let p be a proposition and the set R to be non-empty. We define the formulas of the *modal μ-calculus* over a set of basic propositions of signature Σ as follows:

$$\begin{array}{lll} \phi & ::= & p \mid (\wedge i : i \in T : \phi_i) \mid (\vee i : i \in T : \phi_i) \\ & & \mid \langle L \rangle \phi \mid [L]\phi \\ & & \mid (\nu X.\phi) \mid (\mu X.\phi) \\ & & \mid (\exists x \in T : \phi) \mid (\forall x \in T : \phi) \end{array}$$

A μ-formula is constructed from atomic propositions, conjunction and disjunction, abstraction, modal connectives and fixpoint operators. For modal connectives we use the quantifier notation introduced in Sect. 1.6. We allow to abbreviate the conjunction, respectively disjunction, of two formulas ϕ_1 and ϕ_2 also by $\phi_1 \wedge \phi_2$, respectively $\phi_1 \vee \phi_2$.

Note that we allow the range R in conjunction, disjunction and quantification to be infinite (but not empty).

L is a set of labels. $[L]\phi$ and $\langle L \rangle \phi$ are the labeled modal connectives. $[L]$ is called the *box operator*, $\langle L \rangle$ is called the *diamond operator*. Intuitively, $[L]\phi$ holds if ϕ holds immediately after all transitions with labels in L. Dually, $\langle L \rangle \phi$ holds if there is a transition with a label in L such that ϕ holds immediately afterwards. We use $\langle - \rangle$ and $[-]$ as abbreviations for modal connectives with the label set of all possible labels.

ν is the *greatest fixpoint* operator used, typically, for invariant (safety, "always") properties. μ is the *least fixpoint* operator used, typically, for variant (liveness, "sometime") properties.

We give special names to the following fragments of the modal μ-calculus:

- $\langle\,\rangle\mathcal{L}_\mu(P(\Sigma))$ denotes $[\,]$-free formulas with atomic propositions $P(\Sigma)$ and, dually,
- $[\,]\mathcal{L}_\mu(P(\Sigma))$ denotes $\langle\,\rangle$-free formulas.

With "+", we superscribe fragments that do not contain negation:

- $\langle\,\rangle\mathcal{L}^+_\mu(P(\Sigma))$ and
- $[\,]\mathcal{L}^+_\mu(P(\Sigma))$

are the respective fragments containing only positive atomic propositions.

We are interested in the truth of formulas in a structure (A, R) which is a model of a Maude specification $Sp = (\Sigma, E, T)$. Let us introduce some notation. Let VAR denote the set of variables. and $v : VAR \to A$ be a valuation and let $v^\star$ denote the canonical extension of v to an interpretation function of terms. Let $X := C$ be a valuation, where

$$
\begin{array}{llll}
(A,R),C,v \models \text{“}\texttt{< o : P | \{a = v\} >}\text{”} & \text{iff} & v^\star(\texttt{< o : P | \{a = v\}, \{b = w\} >}) \in C \\
 & & \text{for some } \{b\ =\ w\} \\
 & & \text{and } \texttt{a} \cup \texttt{b} \text{ are the attributes of class } \texttt{P} \\
(A,R),C,v \models \text{“}m\text{”} & \text{iff} & v^\star(m) \in C
\end{array}
$$

$$
\begin{array}{lcll}
(A,R),C,v & \models & \neg\phi & \text{iff } (A,R),C,v \not\models \phi \\
(A,R),C,v & \models & \phi_1 \wedge \phi_2 & \text{iff } (A,R),C,v \models \phi_1 \text{ and } (A,R),C,v \models \phi_2 \\
(A,R),C,v & \models & \phi_1 \vee \phi_2 & \text{iff } (A,R),C,v \models \phi_1 \text{ or } (A,R),C,v \models \phi_2 \\
(A,R),C,v & \models & \langle L\rangle\phi & \text{iff for some } l \in L, (C,C') \in R_{v^\star(l)} \text{ and } (A,R),C',v \models \phi \\
(A,R),C,v & \models & [L]\phi & \text{iff } l \in L \text{ and } (C,C') \in R_{v^\star(l)} \text{ implies } (A,R),C',v \models \phi
\end{array}
$$

$$
\begin{array}{lcll}
(A,R),C,v & \models & (\wedge i : i \in T : \phi_i) & \text{iff } (A,R),C,v \models \phi_i \text{ for all } i \in T \\
(A,R),C,v & \models & (\vee i : i \in T : \phi_i) & \text{iff } (A,R),C,v \models \phi_i \text{ for some } i \in T \\
(A,R),C,v & \models & (\forall x \in T : \phi) & \text{iff } (A,R),C,x := t+v \models \phi \text{ for all } t \in T \\
(A,R),C,v & \models & (\exists x \in T : \phi) & \text{iff } (A,R),C,x := t+v \models \phi \text{ for some } t \in T
\end{array}
$$

$$
\begin{array}{ll}
(A,R),C,v \models (\mu X.\phi) & \text{iff } C \in (\inf S \subseteq A_{\mathrm{Cf}} : |\phi|_{(A,R),X:=S+v} \subseteq S) \\
(A,R),C,v \models (\nu X.\phi) & \text{iff } C \in (\sup S \subseteq A_{\mathrm{Cf}} : |\phi|_{(A,R),X:=S+v} \supseteq S)
\end{array}
$$

As abbreviations we define:

$$
\begin{array}{lclll}
(A,R),C & \models & \phi & \text{iff} & \text{for all } v : FV(\phi) \to A_s \text{ holds } (A,R),C,v \models \phi \\
(A,R) & \models & \phi & \text{iff} & \text{for all } C \in A_{\mathrm{Cf}} \text{ holds } (A,R),C \models \phi \\
Sp & \models & \phi & \text{iff} & \text{for all } (A,R) \in Mod(Sp) \text{ holds } (A,R) \models \phi
\end{array}
$$

Figure 4.1: Truth of μ-formulas

C is assigned to X, and $w + v$ a valuation such that $w + v(X) = w(X)$ if $X \in \mathrm{dom}(w)$ and $v(X)$ if $X \notin \mathrm{dom}(w)$. Let $FV(\phi)$ denote the free variables of formula ϕ.

We define the truth of formulas of the μ-calculus with respect to a state (or configuration) $C \in A_{\mathrm{Cf}}$ in Fig. 4.1.

Let I be the interpretation of μ-formulas in (A, R) under valuation v. We define $\mid \phi \mid_{(A,R),I} (v)$ to be the set of all states of A for which ϕ under the valuation v holds

$$\mid \phi \mid_{(A,R),I} (v) = \{C \mid C \in A \text{ and } (A,R), C, v \models \phi\}$$

Thus, I determines the algebra in which a μ-formula is interpreted. We allow ourselves to abbreviate $\mid \phi \mid_{(A,R),I} (v)$ by $I(\phi)$, provided valuation and transition system are clear in the context.

For a μ-specification $\Phi = \{\phi_1, \ldots, \phi_n\}$ with μ-formulas over an algebraic specification (Σ, E), we define the class $\mathit{Mod}(\Phi)$ of models of Φ, to be the set of structures (A, R), where A is a model of (Σ, E) and $(A, R) \models \phi_1 \wedge \ldots \wedge \phi_n$.

To prove results later we need a way to compute the fixpoints by approximations over induction on the number of iterations.

Proposition 4.1 [Koz83, Bra92] Let (A, R) be a transition system, ϕ a μ-formula and v a valuation, I the standard interpretation of μ-formulas in (A, R) and Ord the class of all ordinals. Then:

$$\begin{array}{lcl} |\mu X.\phi|_{(A,R),I}\,(v) & = & (\sup \alpha \in \mathit{Ord} : |\mu^\alpha X.\phi|_{(A,R),I}\,(v)) \\ |\nu X.\phi|_{(A,R),I}\,(v) & = & (\inf \alpha \in \mathit{Ord} : |\nu^\alpha X.\phi|_{(A,R),I}\,(v)) \end{array}$$

where

$$\begin{array}{lcl} |\mu^0 X.\phi|_{(A,R),I}\,(v) & = & \emptyset \\ |\mu^{\alpha+1} X.\phi|_{(A,R),I}\,(v) & = & \mid \phi \mid_{(A,R),I} ((X := |\mu^\alpha X.\phi|_{(A,R),I}\,(v)) + v) \\ |\mu^\lambda X.\phi|_{(A,R),I}\,(v) & = & (\sup \alpha < \lambda : \mid \mu^\alpha X.\phi \mid_{(A,R),I}\,(v)) \text{ where } \lambda \text{ is a limit ordinal} \\ |\nu^0 X.\phi|_{(A,R),I}\,(v) & = & A \\ |\nu^{\alpha+1} X.\phi|_{(A,R),I}\,(v) & = & |\phi|_{(A,R),I} ((X := |\nu^\alpha X.\phi|_{(A,R),I}\,(v)) + v) \\ |\nu^\lambda X.\phi|_{(A,R),I}\,(v) & = & (\inf \alpha < \lambda : \mid \nu^\alpha X.\phi \mid_{(A,R),I}\,(v)) \text{ where } \lambda \text{ is a limit ordinal} \end{array}$$

For examples that illustrate the necessity of transfinite induction see [Bra92]. However, our μ-formulas are quite simple and we restrict ourselves to classes of μ-formulas and transition systems, where "normal" induction is sufficient to compute a fixpoint.

Definition 4.2 [Bra92] For a formula $\Phi = \sigma Z.\phi$ and a transition system (A, R), the *closure ordinal* $\mathrm{cl}\Phi$ of Φ with respect to (A, R) and v is the least ordinal α such that

$$\mid \Phi^{\alpha+1} \mid_{(A,R),I} (v) = \mid \Phi^\alpha \mid_{(A,R),I} (v)$$

A formula is an *unnested fixpoint* iff all its proper fixpoint subformulae are fix-subsentences. For a formula ϕ, a fixpoint subformula $\sigma X.\psi$ is a *fix-subsentence* of ϕ if, for every variable X in ψ, either X is bound in $\sigma X.\psi$ or X is free in ϕ.

Proposition 4.3 [Bra92] If (A, R) is a finite-branching transition system and $\Phi = \sigma Z.\phi$ is unnested then $\mathrm{cl}\Phi \leq \omega$.

4.2 Consistency of States

In this section, we define two classes of properties: configuration and robustness invariants. With these two classes of properties, we address one severe problem of Maude specifications: a Maude specification does not exclude states (configurations) which are not sensible from the state space of the algebra. Whether a state is sensible or not is a safety property, and typically a specification has quite a number of such very basic safety properties. What we require is that these properties are fairly easy to prove and that there is no transition from a state which has these safety properties to a successor state, which does not have them. Later we are only interested in the (dynamic) properties of states in which such configuration invariants hold or in properties of transition systems for which the configuration invariant holds in all states.

Configuration invariants [WNL95] and robustness invariants are two particular classes of formulas which are characterized in the following definition.

Definition 4.4 (Robustness invariant, configuration invariant) Let *Sp* be a specification and $\phi(c)$ a predicate well-defined over *Sp*, where c is a variable of sort Configuration. Then $\phi(c)$ is called a *configuration invariant* w.r.t. *Sp* if

1. for every rule in *Sp* of the form `m c => d if cond` (where `cond` is a formula, `m` is an atomic message and `c` is a configuration term not containing a configuration variable):
   $Sp \models \texttt{cond} \Rightarrow (\phi(\texttt{c}) \Rightarrow \phi(\texttt{d}))$
2. ϕ is compatible with multiset union, i.e.,
   $Sp \models \phi(c_1) \wedge \phi(c_2) \Rightarrow \phi(c_1 c_2)$
3. ϕ is subset-closed, i.e., $Sp \models (\phi(c) \wedge (c = c_1\ c_2)) \Rightarrow \phi(c_1)$

ϕ is a *robustness invariant* if it satisfies conditions 1 and 2.

We require the transition rules to comply with the configuration invariants: if the invariant holds in some state, then it must hold in all successor states. This closure property facilitates reasoning about substructures in which all states have some property.

We require as closure property that robustness and configuration invariants are compatible with multiset union. Together with the closure property for outgoing transitions, we obtain a subsystem, but not a subalgebra.

The next proposition states that the restriction to a configuration and robustness invariant yields a substructure. Thus, later, when we are interested in all states for which a configuration invariant holds, we do not have to care whether it holds for successor states in the initial model. Naturally, since all other models have more transitions, there might be transitions between states in which the invariant holds to states in which it does not hold.

Proposition 4.5 Let *Sp* be a Maude specification, (A, R) the initial transition system of *Sp* and $\phi(c)$ a robustness invariant for *Sp*. Then (A, R') is a substructure of (A, R), where

$$R' =_{\text{def}} \{(c^A, l^A, d^A) \mid (c^A, l^A, b^A) \in R \wedge R \models \phi(c) \wedge R \models \phi(d)\}$$

Proof. (Sketch) Substructures are closed w.r.t. outgoing transitions and a robustness invariant must hold for all successor states, if it holds in a state.

□

Invariants can be composed to more complex invariants by conjunction.

Lemma 4.6 Let ϕ_1, ϕ_2 be robustness (configuration) invariants. Then $\phi_1 \wedge \phi_2$ is a robustness (configuration) invariant.

Proof. We need the following implication for the proof of (1), (2) and (3): $(\star)$ $((a \Rightarrow b) \wedge (c \Rightarrow d)) \Rightarrow ((a \wedge c) \Rightarrow (b \wedge d))$

1. $Sp \models \texttt{cond} \Rightarrow \phi_1(c) \Rightarrow \phi_1(d)$ and $Sp \models \texttt{cond} \Rightarrow \phi_2(c) \Rightarrow \phi_2(d)$
$\Leftrightarrow$ { semantics of $\wedge$ }
$Sp \models (\texttt{cond} \Rightarrow \phi_1(c) \Rightarrow \phi_1(d)) \wedge (\texttt{cond} \Rightarrow \phi_2(c) \Rightarrow \phi_2(d))$
$\Rightarrow$ { Boolean algebra }
$Sp \models \texttt{cond} \Rightarrow ((\phi_1(c) \Rightarrow \phi_1(d)) \wedge (\phi_2(c) \Rightarrow \phi_2(d)))$
$\Rightarrow$ { $(\star)$ }
$Sp \models \texttt{cond} \Rightarrow ((\phi_1(c) \wedge \phi_2(c) \Rightarrow \phi_1(d) \wedge \phi_2(d)))$

2. $Sp \models \phi_1(c_1) \wedge \phi_1(c_2) \Rightarrow \phi_1(c_1\ c_2)$ and $Sp \models \phi_2(c_1) \wedge \phi_2(c_2) \Rightarrow \phi_2(c_1\ c_2)$
$\Leftrightarrow$ { semantics of $\wedge$ }
$Sp \models (\phi_1(c_1) \wedge \phi_1(c_2) \Rightarrow \phi_1(c_1\ c_2)) \wedge (\phi_2(c_1) \wedge \phi_2(c_2) \Rightarrow \phi_2(c_1\ c_2))$
$\Rightarrow$ { Boolean algebra }
$Sp \models (\phi_1(c_1) \wedge \phi_2(c_1)) \wedge (\phi_1(c_2) \wedge \phi_2(c_2)) \Rightarrow (\phi_1(c_1\ c_2) \wedge \phi_2(c_1\ c_2))$

3. $Sp \models \phi_1(c) \wedge (c = c_1\ c_2) \Rightarrow \phi_1(c_1)$ and $Sp \models \phi_2(c) \wedge (c = c_1\ c_2) \Rightarrow \phi_2(c_1)$
$\Leftrightarrow$ { semantics of $\wedge$ }
$Sp \models (\phi_1(c) \wedge (c = c_1\ c_2) \Rightarrow \phi_2(c_1)) \wedge (\phi_2(c) \wedge (c = c_1\ c_2) \Rightarrow \phi_2(c_1))$
$\Rightarrow$ { $(\star)$ }
$Sp \models \phi_1(c) \wedge \phi_2(c) \wedge (c = c_1\ c_2) \Rightarrow \phi_2(c_1) \wedge \phi_2(c_1)$

□

The following example illustrates the role of configuration invariants with two different implementations of the bounded buffer. The configuration invariant used in this example is an invariant for single objects. For robustness invariants, which involve more objects, see Sect. 4.3.2.

Example. Let us consider two Maude specifications of a bounded buffer. Both specifications implement the bounded buffer with the four attributes `in`, `out`, `max` and `cont`. An invariant for a bounded buffer and a configuration invariant of the specification of a bounded buffer is that the length of the contents is equal to the difference between the values of the attributes `in` and `out` and that the number of elements stored is not greater than the value of attribute `max`. Formally, we define the configuration invariant ϕ by

$$\begin{array}{ll}\phi(c) =_{\text{def}} & (\forall \texttt{B} \in \texttt{ObjectId}, \texttt{I}, \texttt{O} \in \texttt{Nat}, \texttt{M} \in \texttt{NzNat} : \\ & \texttt{< B : BdBuffer | in = I, out = O , cont = L , max = M >} \in c \Rightarrow \\ & (0 \leq (\texttt{I} - \texttt{O}) \leq \texttt{M}) \wedge (\texttt{I} - \texttt{O} = \texttt{length(L)}))\end{array}$$

We give two different implementations of a rule for `put`. In the first rule with label `1`, the condition of the transition rule ensures that a transition may happen if `I-O` is smaller than the maximum number of elements. The precondition of the second implementation with label `P2` additionally requires that the invariant of the transition has to hold:

```
module BD-BUFFER-DEMO-INVARIANTS {
  import {
    protecting (NAT)
    protecting (LIST-ENRICHED)
    protecting (EXT-ACZ-CONFIGURATION)
  }

  signature {
    class BdBuffer {
      in   : Nat
      out  : Nat
      max  : NzNat
      cont : List
    }

     op put _ into _  : Elem ObjectId -> Message
   }

  axioms {
    var  B    : ObjectId
    var  E    : Elem
    var  L    : List
    vars I O  : Nat
    var  M    : NzNat
    var  ATTS : Attributes

                                                   -- First version without invariants
   crl [1]:
       (put E into B)
       < B : BdBuffer | in = I, out = O, max = M, cont = L, ATTS >
       => < B : BdBuffer | in = I + 1, out = O, max = M, cont = E L, ATTS >
          if (sd(I,O) < M)  .
```

```
                                        -- Second version with invariants
   crl [2]:
       (put E into B)
       < B : BdBuffer | in = I, out = O, max = M, cont = L, ATTS >
       => < B : BdBuffer | in = I + 1, out = O, max = M, cont = E L, ATTS >
          if sd(I,O) < M and
             0 <= sd(I,O) and sd(I,O) <= M and
             length(L) == sd(I,O) .
  }
}
```

Note that we have to require that the specification includes function `length`, as defined in specification `LIST-ENRICHED` given in App. A.1.

Clearly, one can prove that, in both cases, if the configuration invariant holds in the present state, then it will hold in the successor states as well. ϕ is a configuration invariant for both specifications (where `get` is implemented analogously).

The behavior of the two specifications differs in "inconsistent" states: rule `1` allows a bounded buffer to accept a message while rule `2` does not. However, we cannot exclude inconsistent states from the algebra with Maude's language constructs.

$\diamondsuit$

This example illustrates the role of configuration invariants: they exclude states that we do not consider as well-formed and states that would never be reachable from well-formed states in the set of states and in the algebra we consider. They replace, in some sense, the well-formedness that is established, e.g., in the functional part of Maude, OBJ3, by the typing discipline: typically, any well-typed term represents a sensible value. At the level of configurations, each combination of objects with any values is well-typed. Configuration invariants allow to describe which configurations are well-formed.

Meseguer uses sort constraints in his version of Maude to describe well-formed elements of sorts. For him the state invariant of a bounded buffer would be a sort constraint, i.e., only objects for which the sort constraint holds are considered to be bounded buffers. This construct makes Maude a very abstract language, but it does not suit an executable language, rather a language to phrase theorems for a theorem prover. Moreover, such constraints can hardly be inherited and they lead to severe problems with partiality. We use states as classes to cope with the partiality of methods. Already this makes the sort and class hierarchy more complex and induces problems with properties of signatures. Sort constraints would add more complexity to this class hierarchy, and would make it impossible to ensure the properties of regularity and coherence which are paramount for the semantics. Moreover, sort constraints suit the object-oriented programming paradigm only at the intra-object level. However, at the inter-object level, it is even more important to reason about the consistency of the state, since Maude's states can be composed in an arbitrary way to form a state according to the object-oriented paradigm. Reasonable properties of specifications will only hold for consistent states. Since we would like to keep our language as simple and as small as possible, we have decided to use only one concept and have chosen invariants, since they apply to both the inter- and the intra-object level.

There is another way to restrict the set of all states, namely by introducing initial states and by considering only those states which are reachable from the initial states as in [WK96]. However, initial states are mostly of concern when discussing reachability, an issue that is marginal in this work. When one uses initial states and considers only reachable states, one must prove that the states one is interested in are reachable, and vice versa, that all inconsistent states are not reachable. We argue that the concept of considering only states which are reachable from initial states suit neither Maude nor object orientation, in general:

1. In Maude, objects can be created quite "casually", i.e., without any proto-object involved, since an object present at the right-hand side, but not at the left-hand side, is created when the rule is applied. Thus, we have no notion of distinguished initial states for Maude objects and would not like to have global distinguished initial states either.

2. Considering only states which are reachable from a set of initial states does not suit the object oriented design philosophy. Classes, more precisely, class hierarchies are designed to be reused and extended. Thus, it is paramount that the designer of a class ensures the correctness of the class. However, it is unknown in which context objects of this class are going to be embedded later, in the second phase of the design process, in which a system is composed from objects. Only at this point of the design process, the global correctness of the state becomes an issue. Thus, for us, the correctness of the global state is a proof obligation, which can only be phrased and met when a system is being composed from objects and not during the design phase, in which the class hierarchy with the description of the behavior of the objects is designed.

We have chosen to consider invariants as a means to phrase and prove consistency in a way that is (1) adequate for Maude and (2) adequate in object orientation.

4.3 Abstraction and Verification

The framework of abstraction and verification we use in this section is based on the duality of μ-formulas and bisimulation relations. μ-formulas characterize finitely branching transition systems. This means that two states of a transition system satisfy the same sets of μ-formulas if and only if they are bisimilar. When verifying a Maude specification, i.e., when proving that the initial model of a Maude specification satisfies a μ-formula, we try to find a simpler bisimilar transition system, verify the property at this level and infer from this that the original transition system satisfies the formula. For the formal framework—in particular, for the relations and the notation—we rely on [LGS+95, Bru93]. As a relation for the state spaces we use Galois connections and as relations for the transition systems we use (bi)simulation relations which are parameterized with Galois connections.

4.3.1 Galois Connections and Simulation Relations

The first step towards a framework which allows abstraction in verification is a relation between the computation structures, more precisely, between substructures representing the well-formed states. Following [LGS+95], we use Galois connections to relate the algebras that are models of the concrete and the abstract specification.

Let us introduce some abbreviations and notation:

- $\overline{X}$ is the complement of X in the domain of X: $\overline{X} = dom(X) \setminus X$.
- $Id^Q : \wp(Q) \rightarrow \wp(Q)$ is the identity function on a set Q, i.e., $Id^Q(X) = X$ for $X \subseteq Q$.
- The dual of a function α is $\tilde{\alpha}$, defined by $\tilde{\alpha}(X) =_{\text{def}} \overline{\alpha(\overline{X})}$.
- Let Q be a set of states, $X \subseteq Q$, L a set of labels and R a relation;

  $$pre(R)(L)(X) =_{\text{def}} \{c \mid c \in Q, (\exists l \in L, d \in X : (c, l, d) \in R)\}$$

  represents the set of predecessors in a labeled transition relation R by transitions with a label in the label set L.

  For an unlabeled relation, we define analogously the set of predecessors by:

  $$pre(R)(X) =_{\text{def}} \{c \mid c \in Q, (\exists d \in X : (c, d) \in R)\}$$

  Dually, we define the set of successors of a set of states by:

  $$post(R)(X) =_{\text{def}} \{d \mid d \in Q, (\exists c \in X : (c, d) \in R)\}$$

- Let S_1, S_2 be two sets of configurations: $S_1 \uplus S_2 =_{\text{def}} \{C_1\ C_2 \mid C_1 \in S_1, C_2 \in S_2\}$. (Remember that the multiset union of configurations is written $C_1\ C_2$.)

A Galois connection is a relation between two sets, which is determined by two functions α and γ. As the two names of functions α and γ suggest, we refer to the two functions as abstraction and concretion function and to the two sets as the abstract and the concrete set. α is the abstraction function, which maps concrete to abstract values, and γ the concretion function, which maps abstract to concrete values.

Definition 4.7 (Galois connection) Let Q_1 and Q_2 be two sets. A *Galois connection* (α, γ), from $\wp(Q_1)$ to $\wp(Q_2)$ is a pair of continuous functions $\alpha : \wp(Q_1) \rightarrow \wp(Q_2)$, $\gamma : \wp(Q_2) \rightarrow \wp(Q_1)$ such that

$Id^{Q_1} \subseteq \gamma \circ \alpha$ and $\alpha \circ \gamma \subseteq Id^{Q_2}$

Note that a relation between two sets can be lifted to a function between powersets. Note also that continuity implies monotonicity in this setting.

In an algebraic setting, usually, one uses Galois insertions, a special class of Galois connections, to relate two algebras. Hereby, α and γ have to obey additionally $\alpha \circ \gamma = Id^{Q_2}$.

Example. (Galois connection) Let us give a first small example of a Galois connection.

Let `NAT` denote specification of of natural numbers including `0`, defined by `succ` and `zero`. This is our concrete specification.

The abstract specification comprises only as set of two operators, $\{\texttt{0}, \texttt{+}\}$. The symbol `0` is the abstraction of the first natural number `zero` and `+` the abstraction of all positive numbers.

We consider two initial term-generated algebras of those two specifications, We denote these two algebras by $I(\texttt{NAT})$ and $I(\{0,+\})$.

We define (α, γ) to be a Galois connection from $\wp(I(\texttt{NAT}))$ to $\wp(I(\{0,+\}))$ by

$$
\begin{array}{lcl}
\alpha : \wp(I(\texttt{NAT})) \rightarrow \wp(I(\{\texttt{0},\texttt{+}\})) & & \\
\alpha(\emptyset) & = & \emptyset \\
\alpha(\{0\}) & = & \{0\} \\
\alpha(\{n\}) & = & \{+\} \text{ for all } n \in I(\texttt{NAT})_{\mathrm{Nat}}, n > 0 \\
\alpha(S_1 \cup S_2) & = & \alpha(S_1) \cup \alpha(S_2)
\end{array}
$$

$$
\begin{array}{lcl}
\gamma : \wp(I(\{0,+\})) \rightarrow \wp(I(\texttt{NAT})) & & \\
\gamma(\emptyset) & = & \emptyset \\
\gamma(\{0\}) & = & \{0\} \\
\gamma(\{+\}) & = & I(\texttt{NAT})_{\mathrm{Nat}} \backslash \{0\} \\
\gamma(\{0,+\}) & = & \texttt{NAT}_{\mathrm{Nat}}
\end{array}
$$

$\diamondsuit$

This is an example of a Galois connection between two initial term-generated algebras. In the next example, we extend this Galois connection to one between algebras containing objects.

Example. (Galois connection (continued)) Let $\texttt{OBJNAT} = (\Sigma_{\texttt{OBJNAT}}, E_{\texttt{OBJNAT}})$ be an algebraic specification.

```
module OBJNAT {
 import {
   protecting (ACZ-CONFIGURATION)
 }

 signature {
   [Nat]
```

```
   class ObjNat {
      val : Nat
   }

   op zero : -> Nat
   op succ : Nat -> Nat
   }
}
```

Specification OBJNAT defines an algebraic datatype Nat and a class ObjNat of objects that contain an attribute of sort Nat. Algebra A-OBJNAT contains a carrier set $\texttt{A-OBJNAT}_{\mathrm{Nat}}$, a carrier set $\texttt{A-OBJNAT}_{\mathrm{ObjNat}}$ and a function succ on the carrier set Nat. Furthermore, it contains a carrier set for $\texttt{A-OBJNAT}_{\mathrm{ObjectId}}$ for the standard sort ObjectId. All object-oriented sorts and symbols of specification ACZ-CONFIGURATION are part of a Maude specification by default. For brevity, we give the mapping here only for sort ObjectId.

The abstract specification OBJSIGNUM is defined by:

```
module OBJSIGNUM {
  import {
   protecting (ACZ-CONFIGURATION)
  }

  signature {
    [Signum]

    class ObjSignum {
      val : Signum
    }

    op 0 : -> Signum
    op + : -> Signum
  }
}
```

We denote the initial term-generated algebras of the two specifications with A-OBJNAT and A-OBJSIGNUM and refrain from adorning the interpretation of a term in the algebra with the name of the algebra.

We extend (α, γ), as it is defined in the first part of this example, to a Galois connection from $\wp(\texttt{A-OBJNAT})$ to $\wp(\texttt{A-OBJSIGNUM})$.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\alpha : \wp(\texttt{A-OBJNAT}) \to \wp(\texttt{A-OBJSIGNUM})} \\
\alpha(\{\texttt{O}\}) & = & \{\texttt{O}\}\ \texttt{for all}\ \texttt{O} \in \texttt{A-OBJNAT}_{\text{ObjectId}} \\
\alpha(\{\texttt{V}\}) & = & \{\texttt{V}\} \text{ for all } \texttt{V} \in \texttt{A-OBJNAT}_{\text{Nat}} \\
\alpha(\texttt{< O : ObjNat | val = V >}) & = & \{\texttt{< O' : ObjSignum | val = V' >} \mid \\
 & & \quad \texttt{O}' \in \alpha(\{\texttt{O}\}), \texttt{V}' \in \alpha(\{\texttt{V}\})\} \\
\alpha(\{C_1\ C_2\}) & = & \alpha(\{C_1\}) \uplus \alpha(\{C_2\}) \\
\alpha(S_1 \cup S_2) & = & \alpha(S_1) \cup \alpha(S_2)
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\gamma : \wp(\texttt{A-OBJSIGNUM}) \to \wp(\texttt{A-OBJNAT})} \\
\gamma(\{\texttt{O}\}) & = & \{\texttt{O}\} \text{ for all } \texttt{O} \in \texttt{A-OBJSIGNUM}_{\text{ObjectId}} \\
\gamma(\{\texttt{V}\}) & = & \{\texttt{V}\} \text{ for all } \texttt{V} \in \texttt{A-OBJSIGNUM}_{\text{Signum}} \\
\gamma(\texttt{< O : ObjSignum | val = V > }) & = & \{\texttt{< O' : ObjNat | val = V' >} \mid \\
 & & \quad \texttt{O'} \in \gamma(\{\texttt{O}\}), \texttt{V'} \in \gamma(\{\texttt{V}\})\} \\
\gamma(\{C_1\ C_2\}) & = & \gamma(C_1) \uplus \gamma(C_2) \\
\gamma(S_1 \cup S_2) & = & \gamma(S_1) \cup \gamma(S_2)
\end{array}
$$

$\diamondsuit$

We give this mapping schematically. The mapping of objects could be defined in a more elegant way. Note that it is "almost" a homomorphism, i.e., $\alpha(op^{\texttt{A-OBJNAT}}(p)) = op^{\texttt{A-OBJSIGNUM}}(y)$ if $y = \alpha(p)$. This is typical for the mappings we employ. In contrast to Σ-homomorphisms, the mappings relate algebras with different signatures and they are relations, not functions. Typical for the mappings we use henceforth is that they are defined on basic data types.

From now on, we allow ourselves to abbreviate the notation when giving mappings between transition systems. We omit braces around single configurations, e.g., we write $\alpha(C)$ for $\alpha(\{C\})$ and $\alpha(S) = C$ for $\alpha(S) = \{C\}$ (analogously for γ).

With the next corollary, we demonstrate that it is sufficient, at least for α, to give the mapping for singleton sets since α distributes over union of sets. Later, also γ will distribute over set union and, in this case, we give the mapping γ also for singleton sets only. Thus, omitting the set braces will not lead to misunderstandings.

Lemma 4.8 (Properties of Galois connections) Let Q_1 and Q_2 be two sets and (α, γ) a Galois connection from from $\wp(Q_1)$ to $\wp(Q_2)$.

1. $(\tilde{\gamma}, \tilde{\alpha})$ is a Galois connection from $\wp(Q_2)$ to $\wp(Q_1)$.

2. α distributes over union of sets, i.e., $\alpha(S_1 \cup S_2) = \alpha(S_1) \cup \alpha(S_2)$.

3. γ distributes over intersection of sets, i.e., $\gamma(S_1 \cap S_2) = \gamma(S_1) \cap \gamma(S_2)$.

4. $\tilde{\alpha}$ distributes over intersection of sets, i.e., $\tilde{\alpha}(S_1 \cap S_2) = \tilde{\alpha}(S_1) \cap \tilde{\alpha}(S_2)$.

5. $\tilde{\gamma}$ distributes over union of sets, i.e., $\tilde{\gamma}(S_1 \cup S_2) = \tilde{\gamma}(S_1) \cup \tilde{\gamma}(S_2)$.

Proof. See App. B. $\square$

Galois connections provide the formal framework for relating sets of states. Let us now define a simulation relation between transition systems whose states are in a Galois connection. In contrast to [LGS+95], we say "S_2 simulates S_1", when S_2 offers more transitions than S_1, as it is common in concurrent languages.

Definition 4.9 ($\sqsubseteq_{(\alpha,\gamma)}$ and $\simeq_{(\alpha,\gamma)}$) Let $S_1 = (Q_1, R_1)$ and $S_2 = (Q_2, R_2)$ be two transition systems, L_1 the set of labels of S_1 and (α, γ) a Galois connection from $\wp(Q_1)$ to $\wp(Q_2)$. S_2 is an (α, γ)-*simulation* of S_1, written $S_1 \sqsubseteq_{(\alpha,\gamma)} S_2$, if and only if, for any $L \subseteq L_1$,

$$\alpha \circ \mathit{pre}(R_1)(L) \circ \gamma \subseteq \mathit{pre}(R_2)(\alpha(L))$$

S_1 and S_2 are (α, γ)-*bisimilar*, written $S_1 \simeq_{(\alpha,\gamma)} S_2$, if and only if

$$S_1 \sqsubseteq_{(\alpha,\gamma)} S_2 \text{ and } S_2 \sqsubseteq_{(\tilde{\gamma},\tilde{\alpha})} S_1.$$

Let us illustrate this relation with a picture (Fig. 4.2).

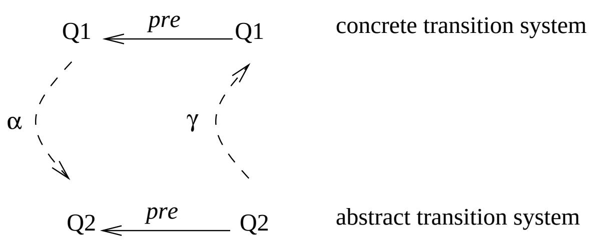

Figure 4.2: Simulation relation

Example. Assume we have two transition systems.

The concrete transition system, (A_1, R_1), consists of an algebra containing four states, c_1, c_2, d_1, d_2, two labels, l_1 and l_2, and a relation between the states:

$$(A_1, R_1) =_{\text{def}} (\{c_1, c_2, d_1, d_2, l_1, l_2\}, \{(c_1, l_1, d_1), (c_2, l_2, d_2)\})$$

The abstract transition system, (A_2, R_2), consists of three states, e_1, f_1 and f_2, two labels, m_1 and m_2, and a relation:

$$(A_2, R_2) =_{\text{def}} (\{e_1, f_1, f_2, m_1, m_2\}, \{(e_1, m_1, f_1), (e_1, m_2, f_2)\})$$

The transition systems and the abstraction function α are depicted in Fig. 4.3.

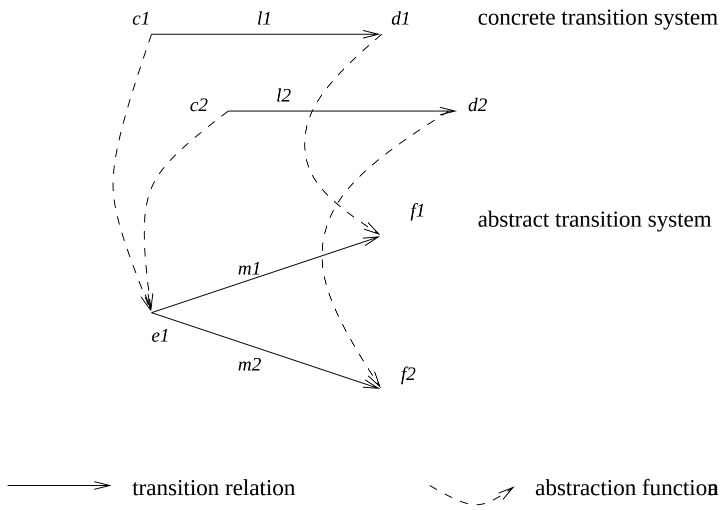

Figure 4.3: Abstract and concrete transition systems with abstraction function

We define α and γ by:

$$\begin{array}{lcll}
\alpha(c_i) & = & e_1 & \text{for} \quad i \in \{1,2\} \\
\alpha(d_i) & = & f_i & \text{for} \quad i \in \{1,2\} \\
\alpha(l_1) & = & m_i & \text{for} \quad i \in \{1,2\} \\
\\
\gamma(e_1) & = & \{c_1, c_2\} & \\
\gamma(f_i) & = & d_i & \text{for} \quad i \in \{1,2\} \\
\gamma(m_i) & = & l_i & \text{for} \quad i \in \{1,2\}
\end{array}$$

Demonstrandum 4.10
The abstract transition system (α, γ)-simulates the concrete transition system.

Proof. For all i $(i \in \{1,2\})$:

$$\begin{array}{ll}
& \alpha(pre(R_1)(l_i)(\gamma(e_i))) \\
= & \emptyset \\
\subseteq & pre(R_2)(\alpha(l_i))(e_i)
\end{array}$$

For all i $(i \in \{1,2\})$:

$$\begin{array}{ll}
& \alpha(pre(R_1)(l_i)(\gamma(f_i))) \\
= & \alpha(pre(R_1)(l_i)(d_i)) \\
= & \alpha(c_i) \\
= & e_i \\
= & pre(R_2)(m_i)(f_i) \\
= & pre(R_2)(\alpha(l_i))(f_i)
\end{array}$$

For all i, j $(i, j \in \{1,2\}, i \neq j)$:

$$\begin{array}{ll} & \alpha(pre(R_1)(l_j)(\gamma(e_i))) \\ = & \emptyset \\ \subseteq & pre(R_2)(\alpha(l_j))(e_i) \end{array}$$

For all i, j $(i, j \in \{1,2\}, i \neq j)$:

$$\begin{array}{ll} & \alpha(pre(R_1)(l_j)(\gamma(f_i))) \\ = & \alpha(pre(R_1)(l_j)(d_i)) \\ = & \alpha(\emptyset) \\ = & \emptyset \\ = & pre(R_2)(m_j)(f_i) \\ = & pre(R_2)(\alpha(l_j))(f_i) \end{array}$$

□

Note that the two transition systems are even (α, γ)-bisimilar. We omit the second direction of the proof here.

◇

Later, we use definitions of simulation relations, each using only one of the two functions α and γ forming the Galois connection.

Lemma 4.11 (Alternative simulation relations) Let (Q_1, R_1) (Q_2, R_2) and L be as in Def. 4.9. Then

$$\begin{array}{llcl} & \alpha\circ pre(R_1)(L)\circ\gamma & \subseteq & pre(R_2)(\alpha(L)) \\ \text{iff} & \alpha\circ pre(R_1)(L) & \subseteq & pre(R_2)(\alpha(L))\circ\alpha \\ \text{iff} & pre(R_1)(L)\circ\gamma & \subseteq & \gamma\circ pre(R_2)(\alpha(L)) \\ \text{iff} & pre(R_1)(L) & \subseteq & \gamma\circ pre(R_2)(\alpha(L))\circ\alpha \end{array}$$

Proof. See App. B.

□

Let us relate this definition of Galois connections to the concept of Σ-homomorphisms introduced in Chap. 3. Let A and B be two Σ-algebras and $h : A \to B$ a Σ-homomorphism. The Σ-homomorphism h can be extended to a function on powersets which distributes over set union. Then h is the abstraction function α of a Galois connection and h^{-1} is the concretion function γ. When relating this to an (α, γ)-simulation relation, α is compatible with the predecessor function, i.e., for a homomorphism we would require that $pre(R)(\alpha(h)) = \alpha(pre(R))$, for a simulation relation that $pre(R)(\alpha(h)) \subseteq \alpha(pre(R))$. The abstraction function preserves the "existence" of transitions, i.e., $(c, l, d) \in R_1 \Rightarrow (\alpha(c), \alpha(l), \alpha(d)) \in R_2$. The transition relation R_2 may contain transitions which are not in the image of R_1 under α. Let us formalize this.

Lemma 4.12 (Simulation relation induced by a Σ-homomorphism) Let Σ be a coherent order-sorted signature. Let (A, R) and (B, S) be two Σ-transition systems and $h : A \to B$ an order-sorted Σ-homomorphism.

We extend $h : A \to B$ to $\alpha : \wp(A) \to \wp(B)$ by

$$\begin{array}{lcl} \alpha(S_1 \cup S_2) & = & \alpha(S_1) \cup \alpha(S_2) \\ \alpha(\{C\}) & = & \{h(C)\} \end{array}$$

$\gamma : \wp(B) \to \wp(A)$ is the extension of $h^{-1} : B \to \wp(A)$ by

$$\begin{array}{lcl} \gamma(S_1 \cup S_2) & = & \gamma(S_1) \cup \gamma(S_2) \\ \gamma(\{C\}) & = & h^{-1}(C) \end{array}$$

Then (α, γ) is a Galois connection and $(A, R) \sqsubseteq_{(\alpha,\gamma)} (B, S)$.

Proof.

1. (α, γ) is a Galois connection:

   (a) α, γ are continuous.

   (b) $Id^A \subseteq \gamma \circ \alpha$

$$\begin{array}{ll} & \gamma(\alpha(X)) \\ = & \quad \{ \text{ rephrasing } \} \\ & \gamma(\{t_1 \mid t_1 \in \alpha(X)\}) \\ = & \quad \{ \text{ definition of } \alpha \text{ } \} \\ & \gamma(\{t_1 \mid (\exists t_2 \in X : t_1 = h(t_2))\}) \\ = & \quad \{ \text{ set theory } \} \\ & \{t_3 \mid (\exists t_1, t_2 : t_3 \in \gamma(\{t_1\}), t_1 = h(t_2), t_2 \in X)\} \\ = & \quad \{ \text{ definition of } \gamma \text{ } \} \\ & \{t_3 \mid (\exists t_1, t_2 : t_3 \in h^{-1}(t_1), t_1 \in h(t_2), t_2 \in X)\} \\ = & \quad \{ \ t_2 \in h^{-1}(h(t_2)) \ \} \\ & X \end{array}$$

   (c) $\alpha \circ \gamma \subseteq Id^B$

$$\begin{array}{ll} & \alpha(\gamma(X)) \\ = & \quad \{ \text{ definition of } \gamma \text{ } \} \\ & \alpha(\{t_2 \mid (\exists t_1 \in X : t_2 \in h^{-1}(t_1))\}) \\ = & \quad \{ \text{ set theory } \} \\ & \{\alpha(t_2) \mid (\exists t_1 \in X : t_2 \in h^{-1}(t_1))\} \\ = & \quad \{ \text{ definition of } \alpha \text{ } \} \\ & \{h(t_2) \mid (\exists t_1 \in X : t_2 \in h^{-1}(t_1))\} \\ = & \quad \{ \text{ set theory } \} \\ & \{t \mid (\exists t_1, t_2 : t_1 \in X, t \in h(t_2), t_2 \in h^{-1}(t_1))\} \\ = & \quad \{ \text{ inverse } \} \end{array}$$

$$
\begin{array}{ll}
 & \{t \mid t \in X\} \\
= & \quad \{\text{ set theory }\} \\
 & X
\end{array}
$$

2. $h : A \to B$ Σ-homomorphism implies $(A, R) \sqsubseteq_{(\alpha,\gamma)} (B, S)$

$$
\begin{array}{ll}
 & h : A \to B \ \Sigma\text{-homomorphism} \\
\Leftrightarrow & \quad \{\text{ property of } \Sigma\text{-homomorphism }\} \\
 & (c, l, d) \in R \text{ implies } (h(c), h(l), h(d)) \in S \text{ for all } c, l, d \in A \\
\Leftrightarrow & \quad \{\text{ definition of } \mathit{pre} \ \} \\
 & c \in \mathit{pre}(R)(l)(d) \text{ implies } h(c) \in \mathit{pre}(S)(h(l))(h(d)) \text{ for all } c, l, d \in A \\
\Rightarrow & \quad \{\ \alpha(\{t\}) = \{h(t)\}, \text{ notational convention }\} \\
 & c \in \mathit{pre}(R)(l)(d) \text{ implies } \alpha(c) \subseteq \mathit{pre}(S)(\alpha(l))(\alpha(d)) \text{ for all } c, l, d \in A \\
\Rightarrow & \quad \{\text{ set theory }\} \\
 & \alpha(\mathit{pre}(R)(l)(d)) \subseteq \mathit{pre}(S)(\alpha(l))(\alpha(d)) \text{ for all } l, d \in A \\
\Rightarrow & \quad \{\ \alpha, \mathit{pre} \text{ distribute over } \cup \ \} \\
 & \alpha(\mathit{pre}(R)(L)(D)) \subseteq \mathit{pre}(S)(\alpha(L))(\alpha(D)) \text{ for all } L, D \subseteq A \\
\Leftrightarrow & \quad \{\text{ function notation }\} \\
 & \alpha \circ \mathit{pre}(R)(L) \subseteq \mathit{pre}(S)(\alpha(L)) \circ \alpha \text{ for all } L \subseteq A_{\text{Message}} \\
\Leftrightarrow & \quad \{\text{ definition of } (\alpha, \gamma)\text{-simulation relation }\} \\
 & (A, R) \sqsubseteq_{(\alpha,\gamma)} (B, S)
\end{array}
$$

□

The Galois connection is a generalization of the concept of a Σ-homomorphism: Σ-homomorphisms are mappings between algebras of the same signature and Galois connections define also mappings between algebras with different signatures.

Dually, a reduct can be extended to a concretion function, while the abstraction function α is induced by a signature morphism. An algebra and a reduct are in a simulation relation. Let us phrase this more precisely.

Lemma 4.13 (Simulation relation induced by a reduct) Let Σ_A and Σ_B be two coherent order-sorted signatures and $\sigma : \Sigma_A \to \Sigma_B$ be an order-sorted signature morphism. Let (A, R) be a term-generated Σ_A-transition system and (B, S) a term-generated Σ_B-transition system such that $(B, S)|_\sigma = (A, R)$. Let $\sigma^\star$ be the extension of σ to terms.

The signature morphism $\sigma : \Sigma_A \to \Sigma_B$ induces a function $\alpha : \wp(A) \to \wp(B)$ by

$$
\begin{array}{lcl}
\alpha(\emptyset) & = & \emptyset \\
\alpha(\{t^A\}) & = & \{(\sigma^\star(t))^B\} \\
\alpha(S_1 \cup S_2) & = & \alpha(S_1) \cup \alpha(S_2)
\end{array}
$$

The reduct $|_\sigma$ is extended to $\gamma : \wp(B) \to \wp(A)$ by

$$
\begin{array}{lcl}
\gamma(\emptyset) & = & \emptyset \\
\gamma(\{t^B\}) & = & \{t' \mid \sigma^\star(t') = t, t' \in T(\Sigma_A)\} \\
\gamma(S_1 \cup S_2) & = & \gamma(S_1) \cup \gamma(S_2)
\end{array}
$$

Then (α, γ) is a Galois connection and $(A, R) \sqsubseteq_{(\alpha,\gamma)} (B, S)$.

Proof.

1. (a) α and γ are continuous.

   (b) $Id^A \subseteq \gamma \circ \alpha$

$$
\begin{array}{ll}
& \gamma(\alpha(X)) \\
= & \quad \{ \text{ definition of } \alpha \ \} \\
& \gamma(\alpha(\{t^A \mid t^A \in X\})) \\
= & \quad \{ \text{ definition of } \alpha, \ \alpha \text{ distributes over } \cup \ \} \\
& \gamma(\{s^B \mid (\exists t : t^A \in X, s \in T(\Sigma_B), (\sigma^\star(t))^B = s^B)\}) \\
= & \quad \{ \text{ definition of } \gamma \ \} \\
& \{c^A \mid (\exists t : t^A \in X, s \in T(\Sigma_B), c^A \in \gamma(\sigma^\star(t))^B)\} \\
= & \quad \{ \text{ set theory } \} \\
& \{c^A \mid (\exists t : t^A \in X, s \in T(\Sigma_B), c^A \in {\sigma^\star}^{-1}(\sigma^\star(t)))\} \\
\subseteq & \quad \{ \text{ set theory } \} \\
& \mathrm{X}
\end{array}
$$

   (c) $\alpha \circ \gamma \subseteq Id^B$

$$
\begin{array}{ll}
& \alpha(\gamma(X)) \\
= & \quad \{ \text{ set theory } \} \\
& \alpha(\gamma(\{t^B \mid t^B \in X\})) \\
= & \quad \{ \text{ definition of } \gamma \ \} \\
& \alpha(\{t^A \mid (\exists s \in T(\Sigma_A) : t^B \in X, \sigma^\star(s) = t)\}) \\
= & \quad \{ \text{ definition of } \alpha, \ \alpha \text{ distributes over } \cup \ \} \\
& \{(\sigma^\star(t))^B \mid (\exists s \in T(\Sigma_A) : (\sigma^\star(s))^B \in X)\} \\
\subseteq & \quad \{ \text{ set theory } \} \\
& X
\end{array}
$$

2.

$$
\begin{array}{ll}
& (A, R) = (B, S)|_\sigma \\
\Rightarrow & \quad \{ \text{ definition of the reduct } \} \\
& (c|_\sigma, l|_\sigma, d|_\sigma) \in R \Rightarrow (c, l, d) \in S \\
\Rightarrow & \quad \{ \text{ definition } \gamma \ \} \\
& (c', l', d') \in R, c' \in \gamma(c), l' \in \gamma(l), d' \in \gamma(d) \text{ implies } (c, l, d) \in S \\
\Leftrightarrow & \quad \{ \text{ notation } \} \\
& pre(R)(\gamma(L)) \circ \gamma \subseteq \gamma \circ pre(S)(L) \\
\Leftrightarrow & \quad \{ \text{ monotonicity of } pre, \ L = \alpha(L') \ \} \\
& pre(R)(\gamma(\alpha(L'))) \circ \gamma \subseteq \gamma \circ pre(S)(\alpha(L')) \\
\Leftrightarrow & \quad \{ \text{ monotonicity of } pre, \ L' \subseteq \gamma(\alpha(L')) \ \} \\
& pre(R)(L') \circ \gamma \subseteq \gamma \circ pre(S)(L') \\
\Leftrightarrow & \quad \{ \text{ Lemma 4.11, substitute variable name } L' \text{ by } L \ \} \\
& \alpha \circ pre(R)(L) \circ \gamma \subseteq pre(S)(\alpha(L))
\end{array}
$$

□

In Sect. 4.3.2, we discuss how a Galois connection, a mapping between algebras of different signatures, can be modeled by several specification-building operations.

In this section and Sect. 3.3, we have introduced simulation and bisimulation relations. Let us show that Defs. 4.9 and 3.27 are equivalent.

Lemma 4.14 (Equivalence of simulation relations) Let (A, R) and (B, S) be two transition systems.

1. Let $\rho \subseteq A \times B$ such that $(A, R) \lesssim_\rho (B, S)$, then $(A, R) \sqsubseteq_{(post(\rho), \widetilde{pre}(\rho))} (B, S)$.
2. Let $(A, R) \sqsubseteq_{(\alpha,\gamma)} (B, S)$, then $(A, R) \lesssim_\rho (B, S)$ for some $\rho \subseteq A \times B$.
3. Let $\rho \subseteq A \times B$ such that $(A, R) \approx_\rho (B, S)$, then $(A, R) \simeq_{(post(\rho), \widetilde{pre}(\rho))} (B, S)$.
4. Let $(A, R) \simeq_{(\alpha,\gamma)} (B, S)$, then $(A, R) \approx_\rho (B, S)$ for some $\rho \subseteq A \times B$.

Proof.

1. $\rho \subseteq A \times B$ implies that $(post(\rho), \widetilde{pre}(\rho))$ is a Galois connection.
   (a) $post(\rho)$ is monotonic. $\widetilde{pre}(\rho)$ is monotonic, since the dual of a monotonic function is monotonic (see proof of Lemma 4.8 in App. B.1) and *pre* is monotonic in its second and third argument.
   (b) $Id^A \subseteq \widetilde{pre}(\rho) \circ post(\rho)$

$$
\begin{array}{ll}
 & \widetilde{pre}(\rho)(post(\rho)(X)) \\
= & \quad \{ \text{ definition of } post \ \} \\
 & \widetilde{pre}(\rho)(\{d \mid (\exists c \in X, d \in A : (c, d) \in \rho)\}) \\
= & \quad \{ \text{ definition of the dual } \} \\
 & \overline{pre(\rho)(\overline{\{d \mid (\exists c \in X, d \in A : (c, d) \in \rho)\}})} \\
= & \quad \{ \text{ complement } \} \\
 & \overline{pre(\rho)(\{d \mid (\exists c \in X, d \in A : (c, d) \notin \rho)\})} \\
= & \quad \{ \text{ definition of } pre \ \} \\
 & \overline{\{c' \mid (\exists c \in X, d \in A : (c, d) \notin \rho \wedge (c', d) \in \rho)\}} \\
= & \quad \{ \text{ set theory } \} \\
 & \overline{\overline{X}} \\
= & \quad \{ \text{ set theory } \} \\
 & X
\end{array}
$$

   (c) $post(\rho) \circ \widetilde{pre}(\rho) \subseteq Id^B$

$$
\begin{array}{ll}
 & post(\rho)(\widetilde{pre}(\rho)(X)) \\
= & \quad \{ \text{ definition of the dual } \} \\
 & post(\rho)(\overline{(pre(\rho)(\overline{X}))}) \\
= & \quad \{ \text{ definition of } pre \ \}
\end{array}
$$

$$
\begin{array}{ll}
& post(\rho)(\overline{\{c \mid c \in B, (\exists d \in \overline{X}, (c,d) \in \rho)\}}) \\
= & \quad \{ \text{ set theory } \} \\
& post(\rho)(\{c \mid c \in B, (\exists d \in \overline{X}, (c,d) \notin \rho)\}) \\
= & \quad \{ \text{ definition of } post \ \} \\
& \{d' \mid (\exists c \in B, d \in \overline{X}, (c,d) \notin \rho, (c,d') \in \rho)\} \\
= & \quad \{ \text{ set theory } \} \\
& \{d' \mid d' \in \overline{\overline{X}}\} \\
= & \quad \{ \text{ set theory } \} \\
& X
\end{array}
$$

$(A,R) \precsim_\rho (B,S)$ implies $(A,R) \sqsubseteq_{(post(\rho),\widetilde{pre}(\rho))} (B,S)$

$$
\begin{array}{ll}
& (A,R) \precsim_\rho (B,S) \\
\Rightarrow & \quad \{ \text{ Def. 3.27 } \} \\
& (\forall c,l,d,e : (c,e) \in \rho \wedge (c,l,d) \in R \Rightarrow (\exists f : (e,l,f) \in S \wedge (d,f) \in \rho)) \\
\Rightarrow & \quad \{ \text{ abstract notation } \} \\
& post(\rho) \circ pre(R)(l) \subseteq pre(S)(l) \circ post(\rho) \\
\Leftrightarrow & \quad \{ \text{ Lemma 4.11 } \} \\
& (A,R) \sqsubseteq_{(post(\rho),\widetilde{pre}(\rho))} (B,S)
\end{array}
$$

2. $(A,R) \sqsubseteq_{(\alpha,\gamma)} (B,S)$ implies $(A,R) \precsim_\rho (B,S)$ for some $\rho \subseteq A \times B$.

   We define ρ by: $(c,e) \in \rho$ iff $e \in \alpha(c)$

$$
\begin{array}{ll}
& (A,R) \sqsubseteq_{(\alpha,\gamma)} (B,S) \\
\Leftrightarrow & \quad \{ \text{ Def. 4.9 } \} \\
& \alpha \circ pre(R)(L) \circ \gamma \subseteq pre(S)(\alpha(L)) \\
\Leftrightarrow & \quad \{ \text{ Lemma 4.11 } \} \\
& \alpha \circ pre(R)(L) \subseteq pre(S)(\alpha(L)) \circ \alpha \\
\Leftrightarrow & \quad \{ \text{ notation } \} \\
& (\forall d \in A : \alpha(pre(R)(L)(d)) \subseteq pre(S)(\alpha(L))(\alpha(d))) \\
\Leftrightarrow & \quad \{ \text{ definition of } pre, \ \alpha(l) = l \text{ for all messages } l \ \} \\
& (\forall c,l,d \in A : (c,l,d) \in R \Rightarrow \alpha(c) \subseteq \{c \mid (\exists d' : (c,l,d') \in S \wedge d' \in \alpha(d))\}) \\
\Leftrightarrow & \quad \{ \ \alpha(c) = e \text{ iff } (c,e) \in \rho, \ \alpha(c) \subseteq \{e \mid \phi(e)\} \text{ implies } \rho(c,e) \Rightarrow \phi(e) \ \} \\
& (\forall c,l,d \in A : (c,l,d) \in R \Rightarrow (c,e) \in \rho \Rightarrow (\exists d' : (c,l,d') \in S \wedge d' \in \alpha(d))) \\
\Leftrightarrow & \quad \{ \ d' \in \alpha(d) = \text{iff } (d,d') \in \rho \ \} \\
& (\forall c,l,d \in A : (c,l,d) \in R \Rightarrow (c,e) \in \rho \Rightarrow (\exists d' : (c,l,d') \in S \wedge (d,d') \in \rho)) \\
\Rightarrow & \quad \{ \text{ Boolean algebra } \} \\
& (\forall c,l,d \in A : (c,l,d) \in R \wedge (c,e) \in \rho \Rightarrow (\exists d' : (c,l,d') \in S \wedge (d,d') \in \rho)) \\
\Rightarrow & \quad \{ \text{ definition of simulation relation } \} \\
& (A,R) \precsim_\rho (B,S)
\end{array}
$$

3. ρ is a bisimulation relation

$$\begin{array}{ll}
& \rho \text{ is a bisimulation relation} \\
\Rightarrow & \{ \text{ definition of bisimulation relation } \} \\
& \rho \text{ and } \rho^{-1} \text{ are simulation relations} \\
\Rightarrow & \left\{ \begin{array}{l} \text{Case (1), where } (post(\rho), \widetilde{pre}(\rho)) \text{ is a Galois connection implies} \\ (pre(\rho), \widetilde{post}(\rho)) \text{ is a Galois connection (Lemma 4.8)} \end{array} \right\} \\
& (A,R) \sqsubseteq_{(post(\rho),\widetilde{pre}(\rho))} (B,S) \text{ and } (B,S) \sqsubseteq_{(pre(\rho),\widetilde{post}(\rho))} (A,R) \\
\Rightarrow & \{ \text{ definition of bisimulation relation } \} \\
& (A,R) \simeq_{(post(\rho),\widetilde{pre}(\rho))} (B,S) \text{ (and } (B,S) \simeq_{(pre(\rho),\widetilde{post}(\rho))} (A,R))
\end{array}$$

4. analogous to (3).

□

We have established the connection between homomorphisms and reducts on the one hand and Galois connections and simulation relations on the other hand. In the next lemma, we establish the relation of simulation relations to specification-building operations in order to be able to use the concepts of simulation relations and the properties that they preserve for relations between classes of algebras.

Lemma 4.15 (Specification morphism and simulation relation) Let Σ_1 and Σ_2 be two coherent order-sorted specifications. Let $\sigma : \Sigma_1 \to \Sigma_2$ be an order-sorted signature morphism and $\sigma^\star$ its extension to terms. Let (Σ_1, E, T) be an order-sorted specification.

For all $(B,S) \in Mod(\Sigma_2, \sigma^\star(E), \sigma^\star(T))$ there exists $(A,R) \in Mod(\Sigma_1, E, T)$ such that $(A,R) \sqsubseteq_{(\alpha,\gamma)} (B,S)$ for some α, γ.

Proof.

$$\begin{array}{ll}
& (B,S) \in Mod(\Sigma_2, \sigma^\star(E), \sigma^\star(T)) \\
\Leftrightarrow & \{ \textit{ translate } \langle \Sigma_1, Mod(\Sigma_1, E, T)\rangle \textit{ by } \sigma = Mod(\Sigma_1, \langle \Sigma_1, \sigma^\star(E), \sigma^\star(T)\rangle) \ \} \\
& (B,S)|_\sigma \in Mod(\Sigma_1, E, T) \\
\Leftrightarrow & \{ \text{ set theory } \} \\
& (B,S)|_\sigma = (A,R) \text{ for } (A,R) \in Mod(\Sigma_1, E, T) \\
\Leftrightarrow & \{ \text{ Lemma 4.13, } \alpha, \gamma \text{ accordingly } \} \\
& (A,R) \sqsubseteq_{(\alpha,\gamma)} (B,S) \text{ for some } (A,R) \in Mod(\Sigma_1, E, T)) \text{ and some } \alpha, \gamma
\end{array}$$

□

Thus, we have a way to obtain an abstract specification for a transition system "automatically". For combining Galois connections and simulation relations to model relations, induced by a combination of specification-building operations, we need the next lemma on transitivity.

Lemma 4.16 (Transitivity of Galois simulation relations) Let (Q_1, R_1), (Q_2, R_2) and (Q_3, R_3) be transition systems, (α_1, γ_1) and (α_2, γ_2) Galois connections and

$(Q_1, R_1) \sqsubseteq_{(\alpha_1,\gamma_1)} (Q_2, R_2)$ and $(Q_2, R_2) \sqsubseteq_{(\alpha_2,\gamma_2)} (Q_3, R_3)$. Then $(Q_1, R_1) \sqsubseteq_{(\alpha_2\circ\alpha_1,\gamma_1\circ\gamma_2)} (Q_3, R_3)$.

Proof.

1. We show that, if (α_1, γ_1) and (α_2, γ_2) are Galois connections, then $(\alpha_2 \circ \alpha_1, \gamma_1 \circ \gamma_2)$ is a Galois connection.

   (a) If α_1,α_2,γ_1,γ_2 are continuous then $\alpha_2 \circ \alpha_1$ and $\gamma_1 \circ \gamma_2$ are continuous.

   (b)
   $$
   \begin{array}{ll}
   & Id^{Q_2} \subseteq \gamma_2 \circ \alpha_2, \alpha_1 \circ \gamma_1 \subseteq Id^{Q_2} \\
   \Rightarrow & \{ \text{ monotonicity } \} \\
   & \gamma_1 \circ Id^{Q_2} \circ \alpha_1 \subseteq \gamma_1 \circ \gamma_2 \circ \alpha_2 \circ \alpha_1, \alpha_2 \circ \alpha_1 \circ \gamma_1 \circ \gamma_2 \subseteq \alpha_2 \circ Id^{Q_2} \circ \gamma_2 \\
   \Rightarrow & \left\{ \ Id^{Q_1} \subseteq \gamma_1 \circ \alpha_1, \alpha_2 \circ \gamma_2 \subseteq Id^{Q_3} \ \right\} \\
   & Id^{Q_1} \subseteq \gamma_1 \circ \gamma_2 \circ \alpha_2 \circ \alpha_1, \alpha_2 \circ \alpha_1 \circ \gamma_1 \circ \gamma_2 \subseteq Id^{Q_3}
   \end{array}
   $$

2. We prove the simulation relation according to Def. 4.9.

$$
\begin{array}{ll}
& (Q_1, R_1) \sqsubseteq_{(\alpha_1,\gamma_1)} (Q_2, R_2) \\
\Leftrightarrow & \{ \text{ Def. 4.9 } \} \\
& \alpha_1 \circ pre(R_1)(L) \circ \gamma_1 \subseteq pre(R_2)(\alpha_1(L)) \\
\Rightarrow & \{ \text{ Def. 4.9 and Lemma 4.11 } \} \\
& \alpha_1 \circ pre(R_1)(L) \circ \gamma_1 \subseteq \gamma_2 \circ pre(R_3)(\alpha_2(\alpha_1(L))) \circ \alpha_2 \\
\Rightarrow & \{ \text{ monotonicity } \} \\
& \alpha_2 \circ \alpha_1 \circ pre(R_1)(L) \circ \gamma_1 \circ \gamma_2 \subseteq \alpha_2 \circ \gamma_2 \circ pre(R_3)(\alpha_2(\alpha_1(L))) \circ \alpha_2 \circ \gamma_2 \\
\Rightarrow & \left\{ \ \alpha_2 \circ \gamma_2 \subseteq Id^{Q_3} \ \right\} \\
& \alpha_2 \circ \alpha_1 \circ pre(R_1)(L) \circ \gamma_1 \circ \gamma_2 \subseteq pre(R_3)(\alpha_2(\alpha_1(L))) \\
\Leftrightarrow & \{ \text{ Def. 4.9 } \} \\
& (Q_1, R_1) \sqsubseteq_{(\alpha_2\circ\alpha_1,\gamma_1\circ\gamma_2)} (Q_3, R_3)
\end{array}
$$

□

We give an example of a simulation relation between two transition systems. This continues the example of a Galois connection between the transition system of specifications `OBJNAT` and `OBJSIGNUM`.

Example. (Continued) We extend specification `OBJNAT` and `OBJSIGNUM` by transition rules and obtain the specifications `OBJNAT+` and `OBJSIGNUM+`.

In `OBJNAT+` an object with object identifier `O` responds to a message `status?` with a message `status of O is V`, where `V` is the value of attribute `val` if `V` is an even number:

```
module OBJNAT+ {
  import { extending (OBJNAT) }

  signature {
```

```
   op to _ status?     : ObjectId -> Message
   op status of _ is _ : ObjectId Nat -> Message
   op iseven           : Nat -> Bool
  }

  axioms {
    var  O    : ObjectId
    vars V N  : Nat
    var  ATTS : Attributes

  crl [status]: (to O status?)
                 < O : ObjNat | val = V, ATTS >
                 => < O : ObjNat | val = V, ATTS >
                    (status of O is V)
                if iseven(V) .

  eq iseven(zero)             = true .
  eq iseven(succ(zero))       = false .
  eq iseven(succ(succ(N)))    = iseven(N) .
  }
}
```

In specification `OBJSIGNUM+` we have the same two messages. An object of class `ObjSignum` may always respond to a `status?` message:

```
module OBJSIGNUM+ {
  import { extening (OBJSIGNUM) }

  signature {
    op to _ status?     : ObjectId -> Message
    op status of _ is _ : ObjectId Signum -> Message
  }

  axioms {
    var O : ObjectId
    var V : Signum
    var ATTS : Attributes

  rl [status]: (to O status?)
               < O : ObjSignum | val = V, ATTS >
               => < O : ObjSignum | val = V, ATTS >
                 (status of O is V) .
  }
}
```

We extend α and γ to cope with the messages:

$$\begin{array}{lcl}
\alpha(\{\texttt{(to O status?)}\}) & = & \{\texttt{(to O' status?)} \mid \texttt{O'} \in \alpha(\{\texttt{O}\})\} \\
\alpha(\{\texttt{(status of O is V)}\}) & = & \{\texttt{(status of O' is V')} \mid \texttt{O'} \in \alpha(\{\texttt{O}\}), \texttt{V'} \in \alpha(\{\texttt{V}\})\}
\end{array}$$

$$\begin{array}{lcl}
\gamma(\{\texttt{(to O status?)}\}) & = & \{\texttt{(to O' status?)} \mid \texttt{O'} \in \gamma(\{\texttt{O}\})\} \\
\gamma(\{\texttt{(status of O is V)}\}) & = & \{\texttt{(status of O' is V')} \mid \texttt{O'} \in \gamma(\{\texttt{O}\}), \texttt{V'} \in \gamma(\{\texttt{V}\})\}
\end{array}$$

We consider the initial term-generated algebras of the two specifications, i.e., $I(\texttt{OBJNAT+} = (\texttt{A-OBJNAT+}, R_{\texttt{OBJNAT+}})$ and $I(\texttt{OBJSIGNUM+} = (\texttt{A-OBJSIGNUM+}, R_{\texttt{OBJSIGNUM+}})$

Demonstrandum 4.17
Then $I(\texttt{OBJNAT+}) \sqsubseteq_{(\alpha,\gamma)} I(\texttt{OBJSIGNUM+})$

We consider a state in algebra `A-OBJNAT+` with one object and one message. In the abstract specification, this state will also consist of one object and one message. The class name and the value stored in the object as well as the parameter to the message differ in the abstract and the concrete specification.

Case 1: The value of attribute `val` is even. Thus, the object may have responded to a `status?` message according to the transition rule given above.

$$\begin{array}{ll}
 & \alpha(\mathit{pre}(R_{\texttt{OBJNAT+}})(\{\texttt{(to O status?)}\}) \\
 & \qquad \texttt{(< O : ObjNat | val = 2 * n > (status of O is 2 * n)))} \\
= & \quad \{ \text{ definition of } \mathit{pre} \text{ and relation } R_{\texttt{OBJNAT+}} \ \} \\
 & \alpha(\{\texttt{(O status?)} \ \ \texttt{< O : ObjNat | val = 2 * n >}\}) \\
= & \quad \{ \text{ definition of } \alpha \ \ \} \\
 & \{\texttt{(O status?)} \ \ \texttt{< O : ObjSignum | val = + >}\} \\
= & \quad \{ \text{ definition of } \mathit{pre} \text{ and relation } R_{\texttt{OBJNAT+}} \ \} \\
 & \mathit{pre}(R_{\texttt{OBJSIGNUM+}})(\{\texttt{(to O status?)}\}) \\
 & \qquad \texttt{(< O : ObjSignum | val = + > (status of O is +))} \\
= & \quad \{ \text{ definition of } \alpha \ \} \\
 & \mathit{pre}(R_{\texttt{OBJNAT+}})(\alpha(\{\texttt{(to O status?)}\})) \\
 & \qquad (\alpha\texttt{(< O : ObjNat | val = 2 * n > (status of O is 2 * n)))}
\end{array}$$

Case 2: The value of attribute `val` is odd. Thus, the object of class `ObjNat` cannot have responded to a `status?` message according to the transition rule given above. The set of predecessor states is empty in the concrete specification, while it is not empty in the abstract specification.

$$\begin{array}{ll}
 & \alpha(\mathit{pre}(R_{\texttt{OBJNAT+}})(\{\texttt{(to O status?)}\}) \\
 & \quad \texttt{(< O : ObjNat | val = (2 * n + 1) > (status of O is (2 * n + 1))))} \\
= & \quad \{ \text{ definition of } \mathit{pre} \text{ and relation } R_{\texttt{OBJNAT+}} \ \} \\
 & \alpha(\emptyset) \\
= & \quad \{ \text{ definition of } \alpha \ \}
\end{array}$$

$\emptyset$
$\subseteq$ { definition of $\emptyset$ }
{`(O status?) < O : ObjSignum | val = + >`}
$=$ { definition of pre and relation $R_{\texttt{OBJNAT+}}$ }
$pre(R_{\texttt{OBJNAT+}})$({`(to O status?)`})
`(< O : ObjSignum | val = + > (status of O is +))`
$=$ { definition of α }
$pre(R_{\texttt{OBJNAT+}})(\alpha($ {`(to O status?)`}$))$
$(\alpha($`< O : ObjNat | val = (2 * n + 1) > (status of O is (2 * n + 1))`$))$

$\diamondsuit$

This example illustrates a simulation relation between two transition systems. Several applications will also have non-homomorphic functions α and γ (see the example in Sect. 4.3.2).

Preservation of a formula by a (monotonic) function α means that, if a formula holds for a set of states, then it holds for the image of this set under α as well. I is the interpretation of a formula with basic proposition from a specification *Sp*, i.e., where the primitive propositions "o" and "m" use terms o and m from the signature of *Sp* and where the labels are terms of sort `Message` of the signature of *Sp*. The image of f is an algebra of a specification with a (possibly) different signature Σ'. $f \circ I$ is the interpretation of a formula in the image of f, i.e., in an algebra in which the elements have denotations from Σ'.

Definition 4.18 (Preservation) Let $S_1 = (Q_1, R_1)$ and $S_2 = (Q_2, R_2)$ be two transition systems, $\phi \in \mathcal{P}$ a formula, and $I : \wp(Q_1) \rightarrow \wp(Q_2)$ an interpretation function. f *preserves* ϕ for I if and only if, for $q \in Q_1$, $q \in |\phi|_{S_1,I}\ v$ implies $f(q) \subseteq |\phi|_{S_2,f\circ I}\ f(v)$

Instead of proving preservation for all states, it is sufficient to prove it for all sets of states that are interpretations of formulas.

Lemma 4.19 (Preservation of formulas) [LGS+95] Let S_1, S_2, f and I be as in Def. 4.18. If, for all formulas $\phi \in \mathcal{P}$, $f(|\ \phi\ |_{S_1,I}\ \ v) \subseteq |\ \phi\ |_{S_2,f\circ I}\ f(v)$ then f preserves $\mathcal{P}$ for I.

Proof.

$f(|\phi|_{S_1,I})v \subseteq |\phi|_{S_2,f\circ I}\ f(v)$
$\Rightarrow$ { $|\phi|_{S_1,I}\ v \supseteq q$, monotonicity }
$f(q) \subseteq |\phi|_{S_2,f\circ I}\ f(v)$

$\square$

The property that a formula is preserved by a function is often not strong enough. The property that a function f is consistent with an interpretation function I means that both the image of a formula and the image of its negation are disjoint. Thus, we can infer that, for all elements in the image of an abstraction of the interpretation of a formula, the formula but not its negation is valid. We need this property later in order to get preservation results for formulas containing negation.

Definition 4.20 (Consistency) [LGS+95] A function f is *consistent* with an interpretation function I if, for all formulas ϕ, $f(\overline{I(\phi)}) \cap f(I(\phi)) = \emptyset$.

A function f is consistent if, for any formula, the image of the interpretation of the formula and of its negation are disjoint.

Up to now, we have defined (α, γ)-(bi)simulations between transition systems. These transition systems are models of Maude specifications. We have also defined what it means for a μ-formula to be valid in a state of such a transition system, or in all states of the transition system. Furthermore, we have defined what it means for a formula to be preserved by a mapping between transition systems. In the following theorem, we state that certain classes of μ-formulas are preserved by (α, γ)-simulation relations. Note that this result for a subset of our μ-formulas can be found at [LGS+95].

Theorem 4.21 (Preservation of properties) Let (Σ_1, E_1, T_1) and (Σ_2, E_2, T_2) be two specifications and (A_1, R_1) and (A_2, R_2) two of their state transition systems. Let $P_1 \subseteq P(\Sigma_1)$ and $P_2 \subseteq P(\Sigma_2)$ be two languages of basic propositions. Let $I_1 : \mathcal{L}_\mu(P_1) \to A_1$ and $I_2 : \mathcal{L}_\mu(P_2) \to A_2$ be two interpretation functions.

1. If $(A_1, R_1) \sqsubseteq_{(\alpha,\gamma)} (A_2, R_2)$ then α preserves $\langle\,\rangle\mathcal{L}_\mu^+(P_1)$ for I_1 and, if α is consistent with I_1, then α preserves $\langle\,\rangle\mathcal{L}_\mu(P_1)$.

2. If $(A_1, R_1) \sqsubseteq_{(\alpha,\gamma)} (A_2, R_2)$ then $\tilde{\gamma}$ preserves $[\,]\mathcal{L}_\mu^+(P_2)$ for I_2 and, if $\tilde{\gamma}$ is consistent with I_2, then $\tilde{\gamma}$ preserves $[\,]\mathcal{L}_\mu(P_2)$ for I_2.

3. If $(A_1, R_1) \simeq_{(\alpha,\gamma)} (A_2, R_2)$ then α preserves $\mathcal{L}_\mu^+(P_1)$ for I_1 and, if α is consistent with I_1, then α preserves $\mathcal{L}_\mu(P_1)$ for I_1.

Proof. See App. B.2.

□

We apply this theorem to a formula and two models of specifications. Say, we would like to infer the validity of a formula in the concrete transition system (A_1, R_1) from its validity in the abstract specification (A_2, R_2). The formula uses basic propositions that are interpreted by an interpretation $\alpha \circ I_1$ as sets of elements of the abstract algebra. The validity of the formula has to be proven for the abstract specification. Provided that the formula contains only appropriate modal connectives, one of the functions, say $\tilde{\gamma}$, preserves these properties. Assume this formula holds in state C_2 of (A_2, R_2). The property holds in state C_1 if C_1 is in the image of C_2 under $\tilde{\gamma}$.

Let us discuss this preservation result. Say, again, we are interested in the preservation of properties of the abstract transition system under $\tilde{\gamma}$. Then the power of the preservation result depends on whether α and $\tilde{\gamma}$ are consistent with the interpretation function of the abstract transition system and also whether $\tilde{\gamma}$ is onto in the concrete specification. Let us explain this in more detail. If γ is not consistent with I, then the codomain of $\tilde{\gamma}$ becomes rather small. This is crucial when we are interested in whether a property holds for the whole specification. Furthermore, if $\tilde{\gamma}$ is not consistent, we may not use negation and, thus, not implication in the formulas. This reduces the power of the approach as well.

Thus, whether this theorem can be applied in a sensible way depends on whether α and γ, or more precisely α and $\tilde{\gamma}$, and the interpretation functions fit well with each other.

We have given two equivalent notions of simulation relations. Naturally, the concept of preservation of properties applies not only to the (α, γ)-simulation relation.

Corollary 4.22 Let (A_1, R_1) and (A_2, R_2) be two transition systems, ϕ a μ-formula and I_1 and I_2 two interpretation functions for μ-formulas in (A_1, R_1) and (A_2, R_2).

1. Let $\rho \subseteq A_1 \times A_2$ be a simulation relation. Then $post(\rho)$ preserves all formulas in $\langle\,\rangle\mathcal{L}_\mu^+(P(\Sigma))$ and, if $post(\rho)$ is consistent with I_1, then ρ preserves formulas in $\langle\,\rangle\mathcal{L}_\mu(P(\Sigma))$.

2. Let $\rho \subseteq A_1 \times A_2$ be a simulation relation. Then $pre(\rho)$ preserves all formulas in $[\,]\mathcal{L}_\mu^+(P(\Sigma))$ and, if $pre(\rho)$ is consistent with I_2, then $pre(\rho)$ preserves formulas in $[\,]\mathcal{L}_\mu(P(\Sigma))$.

3. Let $\rho \subseteq A_1 \times A_2$ be a bisimulation. Then $post(\rho)$ preserves all formulas in $\mathcal{L}_\mu^+$ and, if $post(\rho)$ is consistent with I_1 then $post(\rho))$ preserves formulas in $\mathcal{L}_\mu$.

Proof. The proof is an immediate consequence of Lemma 4.14 and Thm. 4.21.

1. $(A_1, R_1) \precsim_\rho (A_2, R_2)$
   $\Rightarrow$ { Lemma 4.14 }
   $(A_1, R_1) \sqsubseteq_{(post(\rho), \widetilde{pre}(\rho))} (A_2, R_2)$
   $\Rightarrow$ { definition of preservation, Thm. 4.21 }
   $c \in \mid \phi \mid_{(A_1,R_1),I_1} v$ implies $d \in \mid \phi \mid_{(A_2,R_2),post(\rho)\circ I_1} post(\rho)(v)$
   for all $c \in A_1$, $d \in A_2$ and $(c, d) \in \rho$.

2. $(A_1, R_1) \precsim_\rho (A_2, R_2)$
   $\Rightarrow$ { Lemma 4.14 }
   $(A_1, R_1) \sqsubseteq_{(post(\rho), \widetilde{pre}(\rho))} (A_2, R_2)$
   $\Rightarrow$ $\left\{ \text{definition of preservation, Thm. 4.21, } \widetilde{\widetilde{pre}(\rho)} = pre(\rho) \right\}$
   $d \in \mid \phi \mid_{(A_2,R_2),I_2} v$ implies $c \in \mid \phi \mid_{(A_1,R_1),pre(\rho)\circ I_2} pre(\rho)(v)$
   for all $c \in A_1$, $d \in A_2$ and $(c, d) \in \rho$.

3. analogous.

□

Let us illustrate our results with an example.

Example. (Continued) We continue the example of specification `OIDNAT+` and `OIDSIGNUM+`, where $I(\texttt{OIDNAT+}) \sqsubseteq_{(\alpha,\gamma)} I(\texttt{OIDSIGNUM+}))$. Let I_{N} be the interpretation for μ-formulas in $I(\texttt{OIDNAT+})$.

Let us give as example the abstraction of the basic proposition "`< O : ObjNat | val = 1 >`":

$$
\begin{array}{ll}
 & \alpha(|\text{“}\texttt{< O : ObjNat | val = 1 >}\text{”}|_{\texttt{OIDNAT},I_{\mathrm{N}}}\ v \\
= & \quad \{ \text{ semantics of “}\texttt{< O : ObjNat | val = 1 >}\text{” } \} \\
 & \alpha(\{C \mid C \in \texttt{A-OIDNAT}_{\mathrm{Cf}}, \texttt{< O : ObjNat | val = 1 >} \in C\}) \\
= & \quad \{ \ \alpha \text{ distributes over } \cup \ \} \\
 & \{\alpha(C) : C \in \texttt{A-OIDNAT}_{\mathrm{Cf}}, \texttt{< O : ObjNat | val = 1 >} \in C) \\
= & \quad \{ \text{ definition of } \alpha : \alpha(\{C_1 C_2\}) = \alpha(\{C_1\})\ \alpha(\{C_1\})\ \} \\
 & \{C \mid C \in \alpha(\texttt{A-OIDNAT}_{\mathrm{Cf}}), X \in \alpha(\ \texttt{< O : ObjNat | val = 1 >}), X \in C\} \\
= & \quad \left\{ \begin{array}{l} \alpha(\texttt{A-OIDNAT}_{\mathrm{Cf}}) \subseteq \texttt{A-OIDSIGNUM}_{\mathrm{Cf}}, \\ \alpha(\texttt{< O : ObjNat | val = 1 >}) = \texttt{< O : ObjSig | val = + >} \end{array} \right\} \\
 & \{C \mid C \in \texttt{A-OIDSIGNUM}_{\mathrm{Cf}}, \texttt{< O : ObjSig | val = + >} \in C\}
\end{array}
$$

◊

After introducing this framework of abstraction and verification, we explain the use of (α, γ)-simulation relations with a variation of the bounded buffer.

Example. [Relation between a bounded buffer and a unbounded buffer] A bounded buffer is simulated by an unbounded buffer. Specification `BUFFER` of an unbounded buffer is given by:

```
module BUFFER {
  import {
    protecting (LIST)
    protecting (ACZ-CONFIGURATION)
    protecting (NAT)
  }

  signature {
    class Buffer {
      in   : Nat
      out  : Nat
      cont : List
    }
```

```
    op to _ get to _           : ObjectId ObjectId -> Message
    op to _ put _              : ObjectId Elem -> Message
    op to _ answer to get is _ : ObjectId Elem -> Message
  }

  axioms {
    vars B U  : ObjectId
    vars I O  : Nat
    var  L    : List
    var  E    : Elem
    var  ATTS : Attributes

  rl [put]: (to B put E)
            < B : Buffer | in = I, cont = L, ATTS >
            => < B : Buffer | in = I + 1, cont = E L, ATTS > .

  rl [get]: (to B get to U)
            < B : Buffer | out = O, cont = L E, ATTS >
            => < B : Buffer | out = O + 1, cont = L, ATTS >
               (to U answer to get is E) .
  }
}
```

We establish the Galois connection between initial term-generated models of `BD-BUFFER` and `BUFFER`, `A-BD-BUFFER` and `A-BUFFER` . α abstracts from the capacity of a bounded buffer, which is stored in attribute `max`.

$$
\begin{array}{lcl}
\alpha : \wp(\texttt{A-BD-BUFFER}) \to \wp(\texttt{A-BUFFER}) & & \\
\alpha(\texttt{eps}) & = & \texttt{eps} \\
\multicolumn{3}{l}{\alpha(\texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L >})} \\
 & = & \texttt{< B : Buffer | in = I, out = O, cont = L >} \\
\alpha(m) & = & m \text{ for all messages } m \\
\alpha(\{C_1\ C_2\}) & = & \alpha(\{C_1\}) \uplus \alpha(\{C_2\})
\end{array}
$$

The image of an unbounded buffer comprises all bounded buffers γ whose attributes `in` and `out` have equal values and which have a capacity greater or equal to the number of elements stored inside the (bounded) buffer.

$$
\begin{array}{lcl}
\gamma : \wp(\texttt{A-BUFFER}) \rightarrow \wp(\texttt{A-BD-BUFFER}) & & \\
\gamma(\texttt{eps}) & = & \texttt{eps} \\
\gamma(\texttt{< B : Buffer| in = I, out = O, cont = L >}) & & \\
 & = & \{\texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L >} \mid \\
 & & \qquad \texttt{M} \in \texttt{NzNat}\} \\
\gamma(m) & = & m \text{ for all messages } m \\
\gamma(\{C_1\ C_2\} & = & \alpha(\{C_1\}) \uplus \alpha(\{C_2\})
\end{array}
$$

Note that the Galois connections relate also bounded buffers with states that are not sound.

Let `BD-BUFFER` be the specification given in Sect. 2.1.1 and let (α, γ) and `BUFFER` be defined as in this example.

1. (α, γ) is a Galois connection from $\wp(I(\texttt{BD-BUFFER}))$ to $\wp(I(\texttt{BUFFER}))$.
2. $I(\texttt{BD-BUFFER}) \sqsubseteq_{(\alpha,\gamma)} I(\texttt{BUFFER})$

$\diamondsuit$

The example illustrates that the simulation and bisimulation relations could be used as a construct for the reuse of object-oriented specifications.

4.3.2 Verification of a Mutual Exclusion Property

The properties that are directly expressible in Maude specifications are liveness properties. This makes it necessary to verify certain safety properties of Maude specifications. Typically, safety properties have to hold for all paths of an execution and they involve the use of the box operator and greatest fixpoints. An example of a safety property is mutual exclusion. We demonstrate with a proof of the mutual exclusion of accesses to runways in the airport specification how the framework of abstraction and verification can be applied to a specification in Maude. It takes seven steps to prove the mutual exclusion property for runways:

1. Identification of sensible states for which we prove the mutual exclusion property.
2. Statement of the abstract specification: the sentinel.
3. Specification of the mutual exclusion property.
4. Definition of the Galois connection between the concrete specification `AIRPORT` and the abstract specification `SENTINEL`.
5. Mapping between transition rules.
6. Proof of the mutual exclusion property for the specification of the sentinel.

7. Inference of the mutual exclusion for runways from the mutual exclusion of the sentinel.

In this section, we use the subscripts Sen and `Airp` for specifications, interpretation functions and transition systems.

Before verifying the mutual exclusion property, we would like to make a few remarks on how algebraic specifications and Galois connections are combined and why we actually can employ the Galois connection as a specification-building operation.

In the previous section, we have established (α, γ)-simulation relations as mappings between transition systems. These simulation relations are used to verify properties phrased in the μ-calculus. We use them for the verification of algebraic structures. Thus, we would like the mappings to be homomorphisms and we would like to use specification morphisms to obtain the abstract specification. In this section, we establish the connection between algebraic specifications, signature morphisms and Σ-homomorphisms and the framework of abstraction and verification based on (α, γ)-bisimulation. We use the results of this section to motivate why we use the Galois connection to obtain the abstract specification. The proof of the property in the example uses the framework developed in Sect. 4.3.1.

The signature morphisms, the homomorphisms and the specification-building operators of Sect. 3.1.3 allow to establish relations between specifications and to translate specifications. We use this in order to obtain the abstract specification `SENTINEL` from the concrete specification `AIRPORT`. We proceed as follows:

- We enrich the concrete specification with a specification of a state, such that the state can be mapped under a signature morphism to the state of the abstract specification.
- We export the new state from the original specification, such that a signature morphism can be applied as a map of the new state to the abstract specification.
- We establish the signature morphism between the exported signature and the signature of the abstract specification.

The signature morphisms can be extended to specification-building operations according to Sect. 3.1.3. The mappings of this example are depicted in Fig. 4.4.

$$\text{AIRPORT} \xrightarrow{\textit{translate}} \text{AIRPORT+STATUS} \xrightarrow{\textit{export}} \text{STATUS} \xrightarrow{\textit{translate}} \text{SENTINEL}$$

Figure 4.4: Mappings

We use two signature morphisms:

$$\begin{array}{lcl} \sigma_1 & : & \Sigma_{\texttt{AIRPORT}} \rightarrow \Sigma_{\texttt{AIRPORT+STATUS}} \\ \sigma_2 & : & \Sigma_{\texttt{STATUS}} \rightarrow \Sigma_{\texttt{SENTINEL}} \end{array}$$

The signature morphisms induce specification-building operations which map the specification AIRPORT in several steps to the specification SENTINEL:

$$\begin{array}{lcl} \texttt{AIRPORT+STATUS} & = & \textit{translate}\ \texttt{AIRPORT}\ \textit{with}\ \sigma_1 \\ \texttt{STATUS} & = & \textit{export}\ \Sigma_{\texttt{STATUS}}\ \textit{from}\ \texttt{AIRPORT+STATUS} \\ \texttt{SENTINEL} & = & \textit{translate}\ \texttt{STATUS}\ \textit{with}\ \sigma_2 \end{array}$$

Thus, the abstract specification SENTINEL is defined by:

$$\textit{translate}\ (\textit{export}\ \Sigma_{\texttt{STATUS}}\ \textit{from}\ (\textit{translate}\ \texttt{AIRPORT}\ \textit{with}\ \sigma_1))\ \textit{with}\ \sigma_2$$

This expression can be simplified according to rules given in [Wir90]:

$$\begin{array}{ll} & \textit{translate}(\textit{export}\ \Sigma_{\texttt{STATUS}}\ \textit{from}\ (\textit{translate}\ [\texttt{AIRPORT}]\ \textit{with}\ \sigma_1))\ \textit{with}\ \sigma_2 \\ = & \left\{ \begin{array}{l} \textit{translate}\ (\textit{export}\ \Sigma\ \textit{from}\ Sp))\ \textit{with}\ \sigma = \textit{export}\ \Sigma\text{'}\ \textit{from}(\textit{translate}\ Sp\ \textit{with}\ \sigma), \\ \text{for } \sigma : \Sigma \to \Sigma' \end{array} \right\} \\ & \textit{export}\ \Sigma_{\texttt{SENTINEL}}\ \textit{from}\ (\textit{translate}\ (\textit{translate}\ [\texttt{AIRPORT}]\ \textit{with}\ \sigma_1)\ \textit{with}\ \sigma_2) \\ = & \{\ \textit{translate}\ (\textit{translate}\ Sp\ \textit{with}\ \sigma_1)\ \textit{with}\ \sigma_2 = \textit{translate}\ Sp\ \textit{with}\ \sigma_2 \circ \sigma_1\ \} \\ & \textit{export}\ \Sigma_{\texttt{SENTINEL}}\ \textit{from}\ (\textit{translate}\ [\texttt{AIRPORT}]\ \textit{with}\ \sigma_2 \circ \sigma_1) \\ = & \{\ \sigma_2 \text{ onto }\ \} \\ & \textit{translate}\ [\texttt{AIRPORT}]\ \textit{with}\ \sigma_2 \circ \sigma_1 \end{array}$$

Thus, the mapping between the concrete specification AIRPORT and the abstract specification SENTINEL is a combination of two translations. According to Lemma 4.12, we construct the abstract specification by a translation.

We establish a Galois connection between the models of AIRPORT and SENTINEL:

$$\begin{array}{lcll} \alpha(t^{\text{Airp}}) & = & (\sigma_2(\sigma_1(t)))^{\text{Sen}} & \text{for all } t \in \texttt{AIRPORT+STATUS}_{\text{Status}} \\ a^{\text{Airp}} & \in & \gamma(t^{\text{Sen}}) & \text{if } \alpha(a^{\text{Airp}}) = t^{\text{Sen}} \end{array}$$

$f : \texttt{Cf} \to \texttt{Status}$ maps a configuration of AIRPORT to a configuration in AIRPORT+STATUS representing only the status. Since $\alpha(s) = \alpha(f(c))$, for $f(c) = s$, we can use $\alpha \circ f$ and $f^{-1} \circ \gamma$ as Galois connection.

We have achieved (1) that we can use the Galois connection as specification-building operation to obtain the abstract specification automatically and (2) that the specification-building operation establish a Galois connection and—since α is a homomorphism—a simulation relation between transition systems.

Step 1:

We reduce the state space from all possible states to sensible states. Hereby, we use the state space determined by the robustness invariants given in Sect. 4.2.

Lemma 4.23 (Example (without Proof)) Let $\Sigma_{\texttt{Airp}}$ be the signature of the airport.

- Each runway belongs to a tower:
  $\phi_1(c) =_{\text{def}} (\forall \texttt{< R : Runway >} \in c \Rightarrow$
  $(\exists \texttt{T}, \texttt{R}, \texttt{N} : \texttt{< T : Tower | free = F, nfree = N >} \in c \wedge (\texttt{R} \in \texttt{F} \vee \texttt{R} \in \texttt{N})))$

- A runway is either free or not free:
  $\phi_2(c) =_{\text{def}} (\forall \texttt{T}, \texttt{F}, \texttt{N} : \texttt{< T : Tower | free = F, nfree = N >} \in c \Rightarrow \texttt{F} \cap \texttt{N} = \emptyset)$

- If a plane selected a runway, then this runway exists:
  $\phi_3(c) =_{\text{def}} (\forall \texttt{P}, \texttt{R} : \texttt{< P : Plane | runway = R >} \in c \Rightarrow \texttt{< R : Runway >} \in c)$

- If a plane in take-off has selected a runway, then the runway is registered at a tower as not free in attribute `nfree`.
  $\phi_4(c) =_{\text{def}} (\forall \texttt{P}, \texttt{R}, \texttt{T}, \texttt{N} : \texttt{< P : StartingPl | runway = R >} \in c$
  $\Rightarrow \texttt{< T : Tower | nfree = N >} \in c \wedge \texttt{R} \in \texttt{N})$

Let RI_{Airp} be defined by:

$$RI_{\text{Airp}}(c) =_{\text{def}} \phi_1(c) \wedge \phi_2(c) \wedge \phi_3(c) \wedge \phi_4(c)$$

Then $RI_{\texttt{Airp}}$ is a robustness invariant of specification `AIRPORT`.

To ensure the correctness of the mapping, we need to ensure that object identifiers are unique in each configuration. Naturally, this property is neither a robustness nor a configuration invariant since it is not compatible with multiset union. We can safely assume that it is an invariant for our specifications, i.e., that all transition rules preserve this property from states to all successor states. We need this property, in particular, for the object identifiers of towers in a configuration.

Step 2:

The first step in applying the framework of abstraction and verification to a Maude specification is to find a suitable abstract model. From an abstract point of view, a runway can be modeled by a sentinel, an airport with several runways by a set of sentinels.

Let us briefly explain the specification `AIRPORT` and its abstract model, the sentinel.

A plane sends a request for a runway, and the data necessary for the selection of a runway from the set of free runways are exchanged in a synchronous communication. Then the plane taxis to the runway and takes off subsequently. The airborne plane sends a message to the tower, letting the tower know that the runway is free again. The specification of landings of planes on airports is specified analogous.

The signature of the sentinel is given by:

```
module SIG-SENTINEL {
  import {
    extending (ACZ-CONFIGURATION)
    protecting (NAT)
  }
```

```
  signature {
    class Sentinel {
      status : Nat
    }

    op P : ObjectId -> Message
    op V : ObjectId -> Message
    op T : ObjectId -> Message
  }
}
```

A sentinel reacts to three messages. The message `P` marks the entering of the critical section and `V` indicates leaving of the critical section governed by the sentinel. A third message is `T`, which the sentinel can accept at any time and which does not change its state. `T` serves as the abstraction of all messages which have to do with a runway but do not change its state. `P` and `V` are only accepted by a sentinel if it is in the appropriate state: if the state of the sentinel indicates that the critical section is free it accepts a `P`, if the state of the critical section indicates it is occupied it accepts a `V`.

A sentinel guards a critical section. The sentinel allows only one process to enter the critical section at a time. A sentinel is somewhat similar to a semaphore, which also guards a critical section. But different from a semaphore, whose implementation includes typically also queues of processes waiting to enter a semaphore and mechanisms to stop and activate processes trying to enter a critical section, the functionality of a sentinel comprises only the guard mechanism itself.

Step 3:

The mutual exclusion property is given by a formula which describes which sequences of actions are allowed: after a `P` (entering the critical section governed by a sentinel) there has to be a `V` (leaving the critical section) before the next `P` action is allowed. All actions apart from `P(R)` and `V(R)` do not change the state of a sentinel `R`:

$$
\begin{aligned}
&ME =_{\mathrm{def}} (\forall \mathtt{R} \in T_{\mathrm{ObjectId}}(\Sigma) : \text{``}\mathtt{< R : Sentinel \mid status = 0 >}\text{''} \Rightarrow \\
&(\nu X_1.\ [\mathtt{(V(R))}]\ \mathtt{false} \wedge [T_{\mathtt{Message}}(\Sigma_{\mathrm{Sen}}) \backslash \{\mathtt{(V(R))}, \mathtt{(P(R))}\}]\ X_1 \wedge [\mathtt{(P(R))}]\ (\nu X_2. \\
&\qquad [\mathtt{(P(R))}]\ \mathtt{false} \wedge [T_{\mathtt{Message}}(\Sigma_{\mathrm{Sen}}) \backslash \{\mathtt{(V(R))}, \mathtt{(P(R))}\}]\ X_2 \wedge [\mathtt{(V(R))}]\ X_1)))
\end{aligned}
$$

This formula belongs to the μ-fragment $[\ ]\mathcal{L}_\mu$. The class of properties one wants to infer determines the relation between the concrete and the abstract structure. According to Thm. 4.21, $\tilde{\gamma}$ preserves properties in $[\ ]\mathcal{L}_\mu$ like the mutual exclusion property. To infer a property from `SENTINEL` for `AIRPORT`, specification `SENTINEL` has to be such that `AIRPORT` $\sqsubseteq_{(\alpha,\gamma)}$ `SENTINEL`.

Step 4:

Function $\alpha : \wp(A_{\mathrm{Airp}}) \rightarrow \wp(A_{\mathrm{Sen}})$ maps an airport, which consists of a tower and a number of runways and planes, to a collection of sentinels. If a runway is contained in the list of free runways, then the sentinel is in state $\texttt{0}$, otherwise it is in state $\texttt{1}$.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\alpha : \wp(A_{\mathrm{Airp}}) \rightarrow \wp(A_{\mathrm{Sen}})} \\
\alpha(t) & = & t \text{ for all } t \notin \texttt{Airp}_{\mathrm{Cf}} \\
\alpha(C_1\ C_2) & = & \alpha(C_1) \uplus \alpha(C_2) \\
\alpha(\texttt{< P : Plane >}) & = & \texttt{eps} \\
\alpha(\texttt{(P taxis to R)}) & = & \texttt{(P(R))} \\
\alpha(\texttt{(P takeoff at R)}) & = & \texttt{(V(R))} \\
\alpha(\texttt{(P airborne at R)}) & = & \texttt{(T(R))} \\
\multicolumn{3}{l}{\alpha(\texttt{(req runway from T for R to take off)}) = \texttt{(T(R))}} \\
\alpha(\texttt{< R : Runway | ATTS >}) & = & \texttt{eps} \\
\multicolumn{3}{l}{\alpha(\texttt{< T : Tower | free = F1..Fn, nfree = N1..Nm, ATTS >})} \\
 & = & \texttt{< F1 : Sentinel | status = 0 >} \\
 & & \texttt{...} \\
 & & \texttt{< Fn : Sentinel | status = 0 >} \\
 & & \texttt{< N1 : Sentinel | status = 1 >} \\
 & & \texttt{...} \\
 & & \texttt{< Nm : Sentinel | status = 1 >}
\end{array}
$$

Function $\gamma : \wp(A_{\mathrm{Sen}}) \rightarrow \wp(A_{\mathrm{Airp}})$ maps a collection of sentinels and messages to a collection of airports. Since we have abstracted from the structure of the airports (the number of airports and their runways, the planes, the tower and its data), the image of all states with n runways, which have the property $\phi_{\texttt{Airp}}$, is a state with n sentinels:

$$
\begin{array}{l}
\gamma : \wp(A_{\mathrm{Sen}}) \rightarrow \wp(A_{\texttt{Airp}}) \\
\gamma = \alpha^{-1}
\end{array}
$$

Lemma 4.24 (α, γ) is a Galois connection.

Proof. An immediate consequence of Lemma 4.12.

□

To check whether the abstract model suits our needs, we have to check whether $\tilde{\gamma}$ is consistent with the interpretation function of the abstract model for the interpretation of ME.

Note that α is not consistent with the interpretation of basic propositions like, e.g., “$\texttt{< P : Plane >}$” since, e.g., the configurations $\texttt{< P : Plane > C}$ and $\texttt{C}$ are mapped onto the same abstract configuration and, thus, $\alpha(I_{\mathrm{Airp}}(\text{“}\texttt{< P : Plane >}\text{”}))$ and $\alpha(\overline{I_{\mathrm{Airp}}(\text{“}\texttt{< P : Plane >}\text{”})})$ are not disjoint.

Lemma 4.25 $\tilde{\gamma}$ is consistent with I_{Sen}.

Proof.

$$
\begin{array}{ll}
& \tilde{\gamma}(I_{\mathrm{Sen}}(\phi)) \cap \tilde{\gamma}(\overline{I_{\mathrm{Sen}}(\phi)}) = \emptyset \\
\Leftrightarrow & \quad \{ \text{ definition of the dual } \} \\
& \overline{\gamma(\overline{I_{\mathrm{Sen}}(\phi)})} \cap \overline{\gamma(\overline{\overline{I_{\mathrm{Sen}}(\phi)}})} = \emptyset \\
\Leftrightarrow & \quad \{ \text{ set theory } \} \\
& \overline{\gamma(\overline{I_{\mathrm{Sen}}(\phi)})} \cap \overline{\gamma(I_{\mathrm{Sen}}(\phi))} = \emptyset \\
\Leftrightarrow & \quad \{ \text{ set theory } \} \\
& \gamma(\overline{I_{\mathrm{Sen}}(\phi)}) \cup \gamma(I_{\mathrm{Sen}}(\phi)) = A_{\mathrm{Airp}} \\
\Leftrightarrow & \quad \{ \ \gamma = \alpha^{-1} \ \} \\
& \alpha^{-1}(\overline{I_{\mathrm{Sen}}(\phi)}) \cup (I_{\mathrm{Sen}}(\phi)) = A_{\mathrm{Airp}} \\
\Leftrightarrow & \quad \{ \text{ definition of } \alpha^{-1} \ \} \\
& \{ Y \mid \alpha(Y) \in \overline{I_{\mathrm{Sen}}(\phi)} \} \cup \{ Y \mid \alpha(Y) \in (I_{\mathrm{Sen}}(\phi)) \} = A_{\mathrm{Airp}} \\
\Leftrightarrow & \quad \{ \text{ set theory, } \mathrm{card}(\alpha(C)) = 1 \ \} \\
& \{ Y \mid \alpha(Y) \in \overline{I_{\mathrm{Sen}}(\phi)} \vee \alpha(Y) \in (I_{\mathrm{Sen}}(\phi)) \} = A_{\mathrm{Airp}} \\
\Leftrightarrow & \quad \{ \ \alpha \text{ surjective } \} \\
& A_{\mathrm{Airp}} = A_{\mathrm{Airp}}
\end{array}
$$

□

Step 5:

We use function α to find, for each transition rule of `AIRPORT`, a transition rule for `SENTINEL`. The corresponding transition rules are given in Fig. 4.5. The mapping of the transition rules is given in Fig. 4.3.2.

Note that we map rule `A1`, which is responsible for mobility, to `S1`, which "arbitrarily" generates `P` messages, i.e., messages for an arbitrary sentinel. Thus, the selection of a runway is modeled by the possibility of entering any critical section governed by its sentinel.

We obtain as transition rules of specification `SENTINEL`:

```
module SENTINEL {
  import {
    extending (SIG-SENTINEL)
  }

  axioms {
    var R : ObjectId
    var X : Nat
    var ATTS : Attributes

  rl [S1]: (T(R)) < R : Sentinel | status = X, ATTS >
               => < R : Sentinel | status = X, ATTS > (P(R)) .
  rl [S2]: (P(R)) < R : Sentinel | status = 0, ATTS >
```

<table>
<tr><th>Rules of AIRPORT</th><th>Rules of SENTINEL</th></tr>
<tr><td>crl [A1]:
(req runway from T for P to take off)
(< P : onGroundPl | Plane-ATTS >
< T : Tower | free = F,
nfree = N, Tower-ATTS >)
=> (< P : StartingPl | runway = A:ObjectId,
Plane-ATTS >
< T : Tower | free = F,
nfree = N, Tower-ATTS >)
(P taxis to A:ObjectId)
if select(F) == A:ObjectId .</td><td>rl [S1]:
(T(R))
< R : Sentinel |
status = X,
ATTS >
=> < R : Sentinel |
status = X,
ATTS >
(P(R)) .</td></tr>
<tr><td>rl [A2]:
(P taxis to R)
< P : StartingPl | runway = R, Plane-ATTS >
< R : Runway | Runway-ATTS >
< T : Tower | free = F,
nfree = N, Tower-ATTS >
=> < P : TakeoffPl | Plane-ATTS >
< R : Runway | Runway-ATTS >
< T : Tower | free = remove(R,F),
nfree = R N, Tower-ATTS >
(P takeoff at R) .</td><td>rl [S2]:
(P(R))
< R : Sentinel |
status = 0,
ATTS >
=> < R : Sentinel |
status = 1,
ATTS >
(V(R)) .</td></tr>
<tr><td>rl [A3]:
(P takeoff at R)
< P : TakeoffPl | tower = T, Plane-ATTS >
< R : Runway | Runway-ATTS >
< T : Tower | free = F, nfree = N,
Tower-ATTS >
=> < P : FlyingPl | tower = T, height = 100,
Plane-ATTS >
< R : Runway | Runway-ATTS >
< T : Tower | free = R F,
nfree = remove(R,N),
Tower-ATTS >
(to T, P airborne at R) .</td><td>rl [S3]:
(V(R))
< R : Sentinel |
status = 1,
ATTS >
=> < R : Sentinel |
status = 0,
ATTS >
(T(R)) .</td></tr>
<tr><td>rl [A4]:
(to T, P airborne at R)
< T : Tower | free = R F,
Tower-ATTS >
=> < T : Tower | free = R F,
Tower-ATTS > .</td><td>rl [S4]:
(T(R))
< R : Sentinel |
ATTS >
=> < R : Sentinel |
ATTS > .</td></tr>
</table>

Figure 4.5: Mapping of transition rules

```
            => < R : Sentinel | status = 1, ATTS > (V(R)) .
  rl [S3]: (V(R)) < R : Sentinel | status = 1, ATTS >
            => < R : Sentinel | status = 0, ATTS > (T(R)) .
  rl [S4]: (T(R)) < R : Sentinel | status = X, ATTS >
            => < R : Sentinel | status = X, ATTS > .
 }
}
```

Step 6

Lemma 4.26 *ME* holds for $(A_{\mathrm{Sen}}, R_{\mathrm{Sen}})$.

Proof. See App. 4.26.

$\square$

Step 7

To prove that *ME* holds for specification `AIRPORT`—more precisely, for the initial model of this specification—we may apply Thm. 4.21.

Proposition 4.27 *ME* holds for $I($`AIRPORT`$)$.

Proof. We check the requirements of Thm. 4.21.

1. (α, γ) is a Galois connection from A_{Airp} to A_{Sen}. $\surd$ (Lemma 4.24)
2. $(A_{\mathrm{Airp}}, R_{\mathrm{Airp}}) \sqsubseteq_{(,\alpha)} \gamma(A_{\mathrm{Sen}}, R_{\mathrm{Sen}})$. $\surd$
3. $ME \in [\ ]\mathcal{L}_\mu(P(\Sigma))$. $\surd$ (Definition of $[\ ]\mathcal{L}_\mu(P(\Sigma))$)
4. *ME* holds for A_{Airp}. $\surd$ (Lemma 4.26)
5. $\widetilde{\gamma}$ is consistent with I_{Sen}. $\surd$ (Lemma 4.25).

$(A_{\mathrm{Airp}}, R_{\mathrm{Airp}}) \models ME$ if $(A_{\mathrm{Airp}}, R_{\mathrm{Airp}}), C \models ME$ for all $C \in A_{\mathrm{Airp}}$.
Thus, it remains to prove that $\widetilde{\gamma}$ is onto such that $\widetilde{\gamma}$ preserves *ME* for all states of A_{Airp}.

(α, γ) is a Galois connection
$\Rightarrow \quad \{\ \alpha \text{ and } \gamma \text{ are monotonic }\}$
$\gamma(\emptyset) = \emptyset$
$\Rightarrow \quad \{\ \text{definition of the dual }\}$
$\overline{\widetilde{\gamma}(\overline{\emptyset}) = \emptyset}$
$\Rightarrow \quad \left\{\ \overline{\emptyset} = A_{\mathrm{Sen}}\ \right\}$

$$\Rightarrow \left\{ \begin{array}{l} \overline{\tilde{\gamma}(A_{\text{Sen}})} = \emptyset \\ \overline{\emptyset} = A_{\text{Airp}} \\ \tilde{\gamma}(A_{\text{Sen}}) = A_{\text{Airp}} \end{array} \right\}$$

□

Let us briefly discuss this result. We relate specifications with quite different states. The state of a sentinel is determined by a local attribute. The state of a runway depends not on an attribute of the runway, but on the state of the tower and on the state of a plane. While in specification `SENTINEL` an object of class `Sentinel` controls, via attribute `state`, which messages it accepts, a runway is passive in the sense that it cannot influence which planes take off or land on it, and the planes and the tower have control over who gets permission to use a runway. The consistency of the state, e.g., that runway and the information on its state kept in the tower are consistent is ensured by the robustness invariant. In this example, we have only proven the mutual exclusion property for take-offs, not for landings. If the landing is specified in a style similar to the take-off, we can reuse the specification of sentinels. This decreases considerably the amount of work necessary for the proof.

4.4 Inheritance of Properties

In Sect. 4.3.1, we have introduced a framework for relating transition systems. These transition systems are models of specifications or models of μ-formulas. In this section, we investigate particular kinds of relations between transition systems, namely, the relations modeling the reuse by the concepts inheritance, subconfigurations and message algebras. We identify the classes of formulas and, thus, the properties which can be inherited via our reuse concepts. We define a set of μ-schemata with which we specify the behavior of objects in a schematic way. Each of the μ-formulas covers a certain aspect of the behavior of objects of a class. The three levels are depicted in Fig. 4.6.

We use the term “inheritance of properties” not only for Maude’s inheritance relation but also for the two other concepts.

Let us sketch briefly our design scenario and the role of our results for the object-oriented specification of distributed systems. A complex system is developed step by step according to the object-oriented paradigm. Hereby, the class hierarchy is extended and new classes are derived from existing classes. Since we work at a property-oriented level, we are interested in which properties are preserved when deriving new classes. The design of the class hierarchy is the first phase: only in the second phase a system is modeled as a collection of objects. Thus, in the first phase, one is interested in the properties of single classes or objects while, in the second phase, properties of collections of objects like the mutual exclusion property of Sect. 4.3.2 are of interest. Thus, the properties (and proofs) whose inheritability one is interested in are properties (and proofs) of single classes. However, the properties have to reflect the particular view of Maude at the inter-object

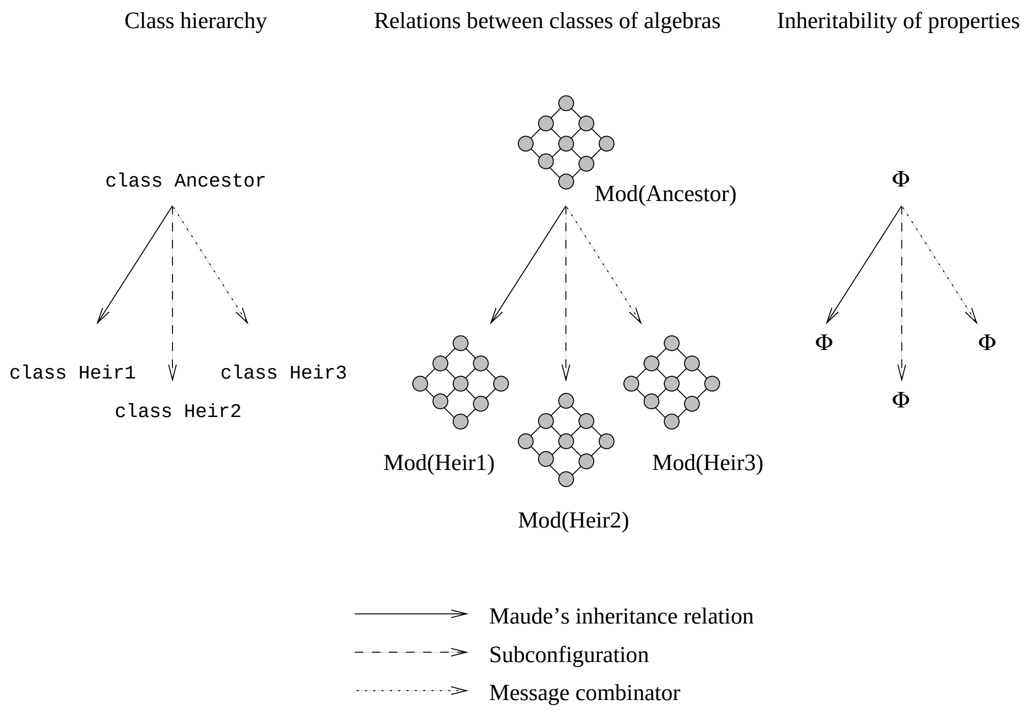

Figure 4.6: Views in refinement

level and, thus, the formula schemata describe the observations made by an observer at the inter-object level.

The relations which we establish here are relations between classes of (term-generated) algebras. Thus, we work at the semantic level of classes of algebras and not at the syntactic level of specifications. We have different motivations to deals not only with the syntactical constructs but with the concepts:

1. Our first motivation is that we are interested in the properties and the classes of algebras for which these properties hold, and not in the representation of these properties in specifications. Accordingly, we are not mainly interested in the syntactical constructs of reuse, but in the relations between classes of algebras. At the level of a specification language, it is important to represent the properties in a concise and comprehensible way and to indicate reuse relations between the classes of algebras. At a more "code-oriented" level, this reuse relation can be implemented by a reuse of code.

2. Our second motivation is that in the process of modeling an object-oriented concurrent system, the relations between the classes have to be established. Thus, one begins at an informal or property-oriented level. After identifying the objects and classes, the class hierarchy has to be determined. Our relation between classes of algebras is a means to identify, on a formal basis, the classes that can be in a "reuse

relation" later in the design process.

Since our level of specification, Maude, is operational, we introduce with the μ-calculus a more property-oriented level of specification.

Let `C` be a classname and `atts` resp. `atts'` denote the attributes with their values of class `C`. Let $SI(\texttt{< B : C | atts}_i\texttt{ >})$, $\phi_i(\texttt{< B : C | atts}_i\texttt{ >})$ and $\psi_i(\texttt{< B : C | atts}_i\texttt{ >})$ be propositions on the state of an object `B` of class `C`. SI is the state invariant of class `C`. Let $\texttt{a}_i$ be a message and let p be all free variables in the formulas with the range P.

We define five formula schemata for a class `C` with n methods:

$$
\begin{aligned}
&\mathit{Persistence}(B) = (\nu X.(\forall p \in P :\\
&\quad \text{“}\texttt{< B : C >}\text{”} \Rightarrow [-](\text{“}\texttt{< B : C >}\text{”} \wedge X)))\\
&\mathit{State}(B) = (\nu X.(\forall p \in P :\\
&\quad \mathit{SI}(\text{“}\texttt{< B : C | atts >}\text{”}) \Rightarrow [-](\mathit{SI}(\text{“}\texttt{< B : C | atts' >}\text{”}) \wedge X)))\\
&\mathit{Synchronization}(B) = (\wedge i : 1 \le i \le n : (\forall p \in P :\\
&\quad \text{“}m_i\text{”} \wedge \text{“}\texttt{< B : C | atts >}\text{”} \wedge \psi_i(\texttt{< B : C | atts >}) \wedge \langle m_i \rangle \texttt{true})\\
&\mathit{StateChange}(B) = (\nu X.(\forall p \in P : (\wedge i : 1 \le i \le n :\\
&\quad \text{“}\texttt{< B : C | atts >}\text{”} \wedge \mathit{SI}(\texttt{ < B : C | atts >}) \wedge \psi_i(\texttt{< B : C | atts >})\\
&\qquad \Rightarrow [m_i](\phi_i(\texttt{< B : C | atts}_i\texttt{' >}) \wedge X))))\\
&\mathit{AnswerMessages}(B) = (\nu X.(\forall p \in P : (\wedge i : 1 \le i \le n :\\
&\quad \text{“ }\texttt{< B : C | atts >}\text{”} \wedge \mathit{SI}(\texttt{< B : C | atts >}) \wedge \psi_i(\texttt{ < B : C | atts >})\\
&\qquad \Rightarrow [m_i](\text{“}\texttt{a}_i\text{”} \wedge X))))
\end{aligned}
$$

For an example of an application of the formula schemata, see Sect. 5.1.3. Note that the formula schema *Synchronization* differs from the formula schema given in 5.1.3. We are interested in whether an action actually happens, while in Sect. 5.1.3 one is interested in a property that holds for all states. Thus, the formula `Synchronization`, used in this section holds only for states in which messages are accepted, while in Sect. 5.1.3 the corresponding formula holds for all states.

The results on inheritability of properties, which will be given in Prop. 4.29, Prop. 4.31 and Prop. 4.33, are summarized in the following table. (The fact that a property is inheritable is marked by $\surd$. The fact that a property is inherited to some of the heirs is marked by $\surd^\star$.)

	Persistence	*State*	*Synchronization*	*Answer-Messages*	*State-Change*
Inheritance			$\surd$		
Subconfiguration	$\surd$	$\surd$		$\surd$	$\surd$
MessageAlgebra	$\surd$	$\surd$	$\surd^\star$	$\surd$	$\surd$

We run into the usual trade-off between the power of the reuse constructs and the quality of the properties that are inherited: the more powerful the reuse concept, the more it allows to change the reused class or object, and the fewer properties both of the class alone and of the global system are preserved in this reuse. In our work, we have decided to give preference to inheritance of properties.

4.4.1 Maude's Inheritance Relation

In Maude an heir inherits from its ancestor the implementation of the state and the ability to react to messages. Particular to Maude is that not only new transitions may be added in the inheriting specification but also new equations may be added that change the semantics of the data types such that new transitions are induced.

Definition 4.28 (Inheritance) Let $\Sigma_A = (S_A, \leq_A, OP_A)$ and $\Sigma_H = (S_H, \leq_H, OP_H)$ be two coherent order-sorted signatures and $Sp_A = (\Sigma_A, E_A, T_A)$ and $Sp_H = (\Sigma_H, E_H, T_H)$ be order-sorted specifications. Let $\sigma : \Sigma_A \to \Sigma_H$ be the canonical injection.

Let C_{A_i} be classes in Σ_A and C_H a class in Σ_H such that $C_H \leq_H C_{A_i}$ for all i, $1 \leq i \leq n$. Sp_H is an *heir of* Sp_A via $C_H \leq_H C_{A_i}$, or Sp_A is an *ancestor of* Sp_H via $C_H \leq_H C_{A_i}$ if

$$\begin{array}{l}(\forall\,(H,S) \in Mod(Sp_H) : \\ \quad (\exists\,(A,R) \in Mod(Sp_A), \rho \subseteq A \times H : \\ \qquad H|_\sigma = A \wedge (A,R) \precsim_\rho (H,S)))\end{array}$$

where $(C,D) \in \rho$ if and only if $C \in \text{filter}(D)$ and

$$\begin{array}{lcll}
\text{filter}(D_1 D_2) & = & \text{filter}(D_1) \uplus \text{filter}(D_2) & \\
\text{filter}(\texttt{eps}) & = & \texttt{eps} & \\
\text{filter}(\texttt{<}\, o : C_H \mid atts \texttt{>}) & = & \{ \texttt{<}\, o : C_{A_i} \mid atts_{A_i} \texttt{>} \mid & \\
 & & \quad C_H \leq C_{A_i}, & \\
 & & \quad (\forall a \texttt{=} w \in atts,\ a \text{ attribute of } C_{A_i}, v \in \text{filter}(w) : & \\
 & & \quad a \texttt{=} v \in atts_{A_i})\} & \\
\text{filter}(\texttt{<}\, O : C \texttt{>}) & = & \texttt{<}\, O : C \texttt{>} & \text{for} \quad C \in \Sigma_A, C \not\leq_H C_H \\
\text{filter}(\texttt{<}\, O : C \texttt{>}) & = & \texttt{eps} & \text{for} \quad C \in \Sigma_H \setminus \Sigma_A, C \not\leq_H C_H \\
\text{filter}(m(p_1 \ldots p_n)) & = & m(\text{filter}(p_1) \ldots \text{filter}(p_n)) & \text{for} \quad m \in \Sigma_A \\
\text{filter}(m(p_1 \ldots p_n)) & = & \texttt{eps} & \text{for} \quad m \in \Sigma_H \setminus \Sigma_A \\
\text{filter}(v) & = & v & \text{for} \quad v : s, s \not\leq \texttt{Cf}
\end{array}$$

Let us explain and motivate this inheritance relation. It is a relation between classes of algebras, not a relation between classes of a specification. Thus, we abstract from the specifications itself and consider for the inheritance relation the classes of algebras for which the axioms of the specification hold. Our inheritance relation is defined by a reduct on the algebras and a simulation relation between the transition systems.

We relate an Sp_H-algebra to an Sp_A-algebra. We require that the reduct of the Sp_H-algebra is an Sp_A-algebra. This ensures that the heir specification contains at least all the basic data types of the ancestor specification. It allows the heir specification to impose more properties on the data structures, i.e., that $(Mod(Sp_H))|_\sigma \subseteq Mod(Sp_A)$. This part of the definition covers data types and the construction of elements of the data types. Note that this notion of inheritance has been used in OS [Bre91] to model inheritance at the intra-object level with the constructive view of an algebraic specification.

The second part of the inheritance relation is a simulation relation between transition systems. We relate states whose parts belonging to the ancestor specification are equal.

Function filter provides this abstraction for the heir configurations and induces a simulation relation ρ on states. A configuration of the heir may have more transitions than the simulated configuration of the ancestor. The transitions may be triggered from messages inherited from the ancestor as well as messages particular to the heir. Thus, a heir inherits from its ancestors the ability to react to all messages the ancestor reacts to.

The abstraction filters the "new observations", which are particular to the heir specification, while the reduct excludes "new elements". This difference in the treatment of the inheritance relation reflects the difference in the construction in algebras and observation in transition systems. filter links the two concepts by abstracting in a way such that behaviorally equal configurations, which are constructed differently, can be related in the inheritance relation.

Note that our inheritance relation does not allow to introduce new operation symbols for basic sorts, which are sorts of an attribute of the ancestor specification. The transition rules specify for which states, and in particular, for which values of attributes, transitions for objects are possible. This property cannot be inherited to new values of some basic sorts, constructed with new operation symbols. Thus, at this point, we treat basic sorts and classes differently in the inheritance relation. We provide function for the abstraction of heir classes to ancestor classes. This abstraction function cannot be found in general for elements of basic sorts.

Maude's object model is the reason why we cannot abstract from the values and consider only the sorts. The synchronization code determines when an object accepts a message and this synchronization code takes the values of objects and messages into account. Thus, we cannot extend the domain of basic values as, e.g., in the inheritance relation in OS.

The inheritance relation allows multiple inheritance. We establish multiple relations between classes of the heir and classes of the ancestor specification. Note, furthermore, that our inheritance relation allows to consider multiple inheritance. This suits the specification style of Maude, with its inter-object view, and the specification method introduced in Sect. 2.4.

In Maude, the behavior of objects is described in an operational style. Let us lift the inheritance relation from Maude specifications to the more property-oriented level of the μ-calculus.

Proposition 4.29 (Inheritance of properties) Let Sp_A, Sp_H, σ, ρ, C_{A_i} and C_H be as in Def. 4.28. Choose $(A,R) \in Mod(Sp_A)$ and $(H,S) \in Mod(Sp_H)$ such that $(A,R) \precsim_\rho (H,S)$.

Let *Synchronization* be the formula schema for class C_{A_i}. *Synchronization* is inherited from C_{A_i} to C_H by inheritance via $C_H \leq_H C_{A_i}$, i.e.,

$$(A,R), C, v \models \mathit{Synchronization}(B) \text{ implies } (H,S), D, w \models \mathit{Synchronization}(B)$$

for all $C \in A_{\mathrm{Cf}}$, $D \in H_{\mathrm{Cf}}$, $(C,D) \in \rho$ and $w \in post(\rho)(v)$.

Proof.

$(A,R) \precsim_\rho (H,S)$

$\Rightarrow \quad \left\{ \text{ Cor. 4.21, } \mathit{Synchronization} \in \langle\,\rangle \mathcal{L}^{+}_{\mu}(P_1) \right\}$
$(A,R), C, v \models \mathit{Synchronization}(B)$ implies $(H,S), D, w \models \mathit{Synchronization}(B)$
for $(C,D) \in \rho$, $w \in \mathit{post}(\rho)(v)$

$\square$

Note that $\mathit{post}(\rho)$ is not consistent with I_A.

This result on the inheritability of properties of specifications justifies the choice of asynchronous message passing in Maude: the transition rules allow only to express that "sometime" an action will happen, which corresponds to the classes of formulas and properties that can be inherited by Maude's inheritance relation.

In the relations between specifications which we define to be an inheritance relation we require the two signatures of ancestor and heir to be coherent, and the algebras to be monotonic. This guarantees, that the ground term algebra is (up to isomorphism) the initial model of the specification. We require that the signature morphism that relates the specifications is the canonical embedding. In Sect. 3.2 be have defined properties for signature morphisms, which ensures that the image of a coherent signature is coherent as well. In this section we are interested in the relations between classes of models, not the the relations between classes of models induced by a signature morphism and thus, we require the property of coherence for both specifications and not, as in Sect. 3.2 for the ancestor specification only.

Let us illustrate the inheritance relation with an example. Note that the inheritance relation actually reuses the specification in the example. Thus, this is an example for an inheritance relation between reusing and reused specification.

Example. In this example, we study the inheritance relation between two specifications `BD-BUFFER` and `BD-BUFFER-X-2` via `XBdBuffer` $\leq$ `BdBuffer`.

Let `BD-BUFFER` be the the specification given in Sect. 2.1 and let `BD-BUFFER-X-SECOND` be the specification given by:

```
module BD-BUFFER-X-SECOND {
  import {
    protecting (BD-BUFFER)
  }

  signature {
    class XBdBuffer [BdBuffer] {
  }

  op last _ replyto _          : ObjectId ObjectId  -> Message
  op to _ answer to last is _ : ObjectId Elem -> Message
  }
```

```
  axioms {
    vars B U       : ObjectId
    var  E         : Elem
    var  C         : List
    vars I O I' O' : Nat
    var  M         : NzNat
    var  ATTS      : Attributes

  ceq [X]: < B : XBdBuffer | (cont = C), (out = O),  (in = I),  (max = M),
                             ATTS >
        = < B : XBdBuffer | (cont = C), (out = O'), (in = I'), (max = M),
                            ATTS >
          if (sd(I,O) == sd(I',O')) .

rl [L]:  (last B replyto U)
         < B : XBdBuffer | cont = E C, out = O, ATTS >
        => < B : XBdBuffer | cont = C, out = O + 1, ATTS >
          (to U answer to last is E) .
  }
}
```

An object of class `XBdBuffer` accepts a message `last`, while a bounded buffer does not. Specification `BD-BUFFER-X-SECOND` abstracts from the values of the two attributes `in` and `out`.

Consider a configuration C_0 containing a bounded buffer and a `put` and a `get` message.

$C_0 =$ `< B1 : BdBuffer | in = 1, out = 0, max = 2, cont = E1 >`
`(put E2 into B1)`
`(get B1 replyto U)`

A number of configurations of an $\mathtt{Sp}_H$-algebra inherit from configuration C_0, i.e., they are in the inheritance relation to C_0. Let us give four examples for such configurations.

- C_0 itself:

  $$D_0 = C_0$$

- A configuration containing a buffer of sort `XBdBuffer`:

  $D_1 =$ `< B1 : XBdBuffer | in = 1, out = 0, max = 2, cont = E1 >`
  `(put E2 into B1)`
  `(get B1 replyto U)`

- A configuration containing an additional message `last`:

$D_2 =$ `< B1 : XBdBuffer | in = 1, out = 0, max = 2, cont = E1 >`
`(put E2 into B1)`
`(get B1 replyto U)`
`(last B1 replyto U)`

- All configurations in which the difference of the values of the attributes `in` and `out`, `in - out`, is equal to 1, as e.g., a bounded buffer with the same capacity and the same contents but a different history:

$D_3 =$ `< B1 : XBdBuffer | in = 2, out = 1, max = 2, cont = E1 >`
`(put E2 into B1)`
`(get B1 replyto U)`

The formula *Synchronization* of class `XBdBuffer` is given by:

Synchronization(B) =
($\forall$`I`, `O` $\in$ `Nat`, `M` $\in$ `NzNat`, `L` $\in$ `List`, `E` $\in$ `Elem` :
$\wedge$ "`(put E into B)`",
$\wedge$ "`< B : BdBuffer | in = I, out = O, max = M, cont = L >`",
$\wedge$ $\mathtt{I} - \mathtt{O} < \mathtt{M}$,
$\wedge$ $\langle$`(put E into B)`$\rangle$`true`)
$\wedge$ ($\forall$`I`, `O` $\in$ `Nat`, `M` $\in$ `NzNat`, `L` $\in$ `List`, `E` $\in$ `Elem` :
$\wedge$ "`(get B replyto U)`"
$\wedge$ "`< B : BdBuffer | in = I, out = O, max = M, cont = L E >`"
$\wedge$ $\mathtt{I} - \mathtt{O} > 0$:
$\wedge$ $\langle$`(get B replyto U)`$\rangle$`true`)

Note that the basic proposition "`< B : BdBuffer | ... >`" holds both for objects of class `BdBuffer` and for objects of class `XBdBuffer`, according to the semantics of "o".

We have to establish a simulation relation between the ancestor and the heir transition system. In this example this is particularly simple, since the heir specification reuses the ancestor specification and has additional equations and transition rules.

Let (A, R) be a term-generated model of the ancestor specification and (H, S) a term-generated model of the heir specification such that $H|_{\Sigma_A} = A$ and $(A, R) \precsim_\rho (H, S)$.

$(A, R), C_0 \models$ *Synchronization*(B) implies $(H, S), D_i \models$ *Synchronization*(B)
for $0 \leq i \leq 3$

$\diamondsuit$

In this example, specification `BD-BUFFER` is reused in specification `BD-BUFFER-X-SECOND`. In specification `BD-BUFFER-X-SECOND`, the subclass relation between classes `BdBuffer` and class `XBdbuffer` is established. For the inheritance relation, a subclass relation between classes is mandatory. However, since the inheritance relation is based on classes of models, one could think about a relation between specifications which do not reuse the ancestor specification. Establishing the simulation relation would follow the concepts introduced in Sect. 4.3.2. The heir specification would be enriched with a function to compute a state which can be translated with a signature morphism to the ancestor specification. This signature morphism can be used to translate the heir specification to the ancestor specification.

In the process of stepwise refinement, such an inheritance relation, which does not employ reuse of the ancestor specification, can be implemented at a more concrete level with reuse of code. At the abstract and property-oriented level, a concise representation of code is necessary in order to keep specifications legible and proofs feasible, while reuse of specification code is less important. Thus, one might prefer to write a concise new specification over reusing and complicating an existing one, while having the reuse relation in mind for reuse of code at a more concrete level.

4.4.2 Subconfigurations

The subconfiguration is the reuse construct which restricts the possibility of an object belonging to the reused class to accept messages which are part of the global state. Particular to our approach is that we model the migration of messages into and out of subconfigurations by equations. Recall that we may use the same object identifier for the encapsulating and the encapsulated object.

Definition 4.30 (Inheritance via subconfigurations) Let Σ_A and Σ_H be coherent order-sorted signatures, $Sp_A = (\Sigma_A, E_A, T_A)$ and $Sp_H = (\Sigma_H, E_H, T_H)$ specifications and $\sigma : \Sigma_A \to \Sigma_H$ the canonical embedding.

Let C_{A_i} be classes in Σ_A for $1 \leq i \leq n$ and C_H a class in Σ_H. C_H has an attribute a of sort `Subconfiguration` and $C_{A_1} \dots C_{A_n} \leq$ `Subconfiguration`.

Sp_H is an *heir by subconfiguration via* (C_H `Subconfiguration of` $C_{A_1} \dots C_{A_n}$) of Sp_A or
Sp_A is an *ancestor by subconfiguration via* (C_H `Subconfiguration of` $C_{A_1} \dots C_{A_n}$) of Sp_H if

$$(\forall (H, S) \in Mod(Sp_H) : (\exists (A, R) \in Mod(Sp_A) : H|_\sigma = A \wedge (H, S) \overset{\sim}{\precsim}_\rho (A, R)))$$

where $(D, C) \in \rho$ if and only if $C = \text{filter}(D)$ and

$$\begin{array}{lcl}
\text{filter}(C_1\ C_2) & = & \text{filter}(C_1)\ \text{filter}(C_2) \\
\text{filter}(\texttt{<}O : C_H \mid atts_H, a \texttt{=} o_1 \dots m_n\texttt{>}) & = & \text{filter}(o_1) \dots \text{filter}(m_n) \\
\text{filter}(m(p_1, \dots, p_n)) & = & m(\text{filter}(p_1), \dots, \text{filter}(p_n)) \\
\text{filter}(\texttt{<}O : C\texttt{>}) & = & \texttt{<}O : C\texttt{>} \text{ for } C \not\leq C_H \\
\text{filter}(v) & = & v \text{ for primitive values } v.
\end{array}$$

Let us discuss this subconfiguration relation. It is—like the inheritance relation—a relation between classes of algebras. According to Maude's particular view of object-oriented systems, this subconfiguration relation covers both the intra- and the inter-object level.

The definition ensures that the heir specification reuses classes of the ancestor specification via a `Subconfiguration` construct. Again, we require that the reduct of all models of the heir specifications is a model of the ancestor specification to ensure that the heir contains all the basic sorts and function symbols of the ancestor. We establish a simulation relation between the ancestor and the heir transition system. Function filter abstracts from the `Subconfiguration` construct and induces a simulation relation. Thus, a configuration of the ancestor specification subsumes (at least) two configurations of the heir, namely one that is identical and one which contains a subconfiguration. In this relation, the ancestor configuration simulates these (at least) two configurations of the heir.

Note that we have to use the notation developed in Sect. 4.3 to express the inheritance relation, since the basic proposition on objects "o" resembles subsorts but not subconfigurations.

Proposition 4.31 (Inheritance of properties by subconfigurations) Let Sp_A, Sp_H, σ, ρ, C_{A_i} and C_H be as in Def. 4.30, I_A the interpretation of μ-formulas in (A, R) and ϕ one of the formula schemata *Persistence*, *StateChange*, *State* and *AnswerMessages* for class C_A. Let Sp_H be a heir by subconfiguration via C_H `Subconfiguration of` $C_{A_1} \ldots C_{A_n}$ from Sp_A.

Choose (A, R) and (H, S) such that $(H, S) \stackrel{\prec}{\sim}_\rho (A, R)$. Then

$$(A, R), C \models \phi$$

implies

$$D \in |\phi|_{(H,S),pre(\rho)\circ I_A}$$

for all $(A, R) \in Mod(Sp_A)$, $(H, S) \in Mod(Sp_H)$ and $C \in A_{\mathrm{Cf}}$, $D \in H_{\mathrm{Cf}}$, $(D, C) \in \rho$.

Proof. We have to prove that $pre(\rho)$ is consistent with I_A, the interpretation function for μ-formulas in (A, R).

$$
\begin{array}{ll}
 & pre(\rho) \text{ is consistent with } I_A \\
\Leftarrow & \{ \text{ definition of consistency, Def. 4.20 } \} \\
 & pre(\rho)(I_A(\phi)) \cap pre(\rho)(\overline{I_A(\phi)}) = \emptyset \\
\Leftarrow & \{ \text{ generalization } \} \\
 & pre(\rho)(X) \cap pre(\rho)(\overline{X}) = \emptyset \\
\Leftarrow & \{ \text{ definition of } pre(\rho) \ \} \\
 & \{h \mid \mathrm{filter}(h) \in X\} \cap \{h \mid \mathrm{filter}(h) \in \overline{X}\} = \emptyset \\
\Leftarrow & \{ \ \mathrm{card}(\mathrm{filter}(\{h\})) = 1 \ \ \} \\
 & \emptyset = \emptyset
\end{array}
$$

We prove our claim:

$(H, S) \precsim_\rho (A, R)$
$\Rightarrow \quad \{ \text{ Cor. } 4.22, \phi \in [\,]\mathcal{L}_\mu(P(\Sigma)) \ \}$
$(A, R), C, v \models \phi$ implies $D \in \mid \phi \mid_{(H,S),pre(\rho)\circ I_A} w$
for $(D, C) \in \rho$, $w \in pre(\rho)(c)$

□

The relation between two initial models is of particular interest, since we inherit formulas containing the box operator. This class of properties is not "inherited" from the initial algebras to other algebras belonging to the models of a specification in the loose approach.

Let us illustrate the reuse via subconfigurations with an example.

Example. We reuse a specification of an unbounded buffer to implement a bounded buffer. The specification of the unbounded buffer is given by:

```
module UNBOUNDED-BUFFER {
 import {
   protecting (ACZ-CONFIGURATION)
   protecting (LIST)
   protecting (NAT)
  }

  signature {
    class UBuffer {
      cont  : List
      in    : Nat
      out   : Nat
     }

  op get _ replyto _         : ObjectId ObjectId -> Message
  op to _ answer to get is _ : ObjectId Elem     -> Message
  op put _ into _            : Elem ObjectId     -> Message
  }

  axioms {
    vars B R U : ObjectId
    var  E     : Elem
    var  C     : List
    vars I O   : Nat
    var  ATTS  : Attributes

  rl [P]: (put E into B)
          < B : UBuffer | cont = C, in = I, ATTS >
          => < B : UBuffer | cont = E C, in = I + 1, ATTS > .
```

```
  rl [G]: (get B replyto R)
          < B : UBuffer | cont = C E, out = O, ATTS >
          => < B : UBuffer | cont = C, out = O + 1, ATTS >
             (to R answer to get is E) .
  }
}
```

The bounded buffer encapsulates an unbounded buffer in a subconfiguration and implements whether to accept a `put` or a `get` message. Hereby, only one message may enter a subconfiguration at a time. The migration into and out of a bounded buffer is implemented by equations.

```
module BD-U-BUFFER {
  import {
    extending (UNBOUNDED-BUFFER)
    protecting (SUBCONFIGURATION)
  }

  signature {
    class BdBuffer {
      max  : NzNat
      conf : Subconfiguration
    }
    [UBuffer < Subconfiguration]
  }

  axioms {
    vars B R : ObjectId
    var  E   : Elem
    vars I O : Nat
    var  M   : NzNat
    var  C   : List
    vars ATTS ATTS' : Attributes

ceq [put]:
    (put E into B)
    < B : BdBuffer | (max = M),
                     (conf = < B : UBuffer | in = I, out = O, ATTS >), ATTS' >
    = < B : BdBuffer | (max = M),
                       (conf = < B : UBuffer | in = I, out = O, ATTS >
                               (put E into B)), ATTS' >
    if sd(I,O) < M .
```

```
eq [get]:
   (get B replyto R)
   < B : BdBuffer | (conf = < B : UBuffer | cont = C E, ATTS >), ATTS' >
    = < B : BdBuffer | (conf = < B : UBuffer | cont = C E, ATTS >
                               (get B replyto R)), ATTS' > .

eq [answer]:
   < B : BdBuffer | (conf = (to R answer to get is E)
                            < B : UBuffer | ATTS >), ATTS' >
   = < B : BdBuffer | (conf =  < B : UBuffer | ATTS >), ATTS' >
     (to R answer to get is E) .
  }
}
```

Consider a configuration in the ancestor specification with an unbounded buffer containing two elements and a `put` and a `get` message.

$C_0 =$ `< B1 : UBuffer | cont = E2 E1, in = 2, out = 0 >`
`(put E3 into B1)`
`(get B1 replyto R)`

In C_0 both first a `put` then a `get`, or first a `get` and then a`put` message are accepted. In both cases the resulting configuration C_2 is given by

$C_2 =$ `< B1 : UBuffer | cont = E3 E2, in = 3, out = 1 >`
`(answer to get is E1)`

In the heir specification `Bd-U-Buffer`, there are several heirs of C_0. The first heir is C_0 itself.

$D_0 = C_0$

Let us give as an example of an heir configuration a bounded buffer with capacity 2.

```
D1 = < B1 : UBuffer | max = 2,
                      conf =  < B1 : UBuffer | cont = E2 E1,
                                               in = 2, out = 0 > >
     (put E3 into B1)
     (get B1 replyto R)
```

In D_0 only one sequence of actions is possible, namely, first a `get` and then a `put`. The final configuration is

```
E1 = < B1 : UBuffer | max = 2,
                      conf =  < B1 : UBuffer | cont = E3 E2,
                                               in = 3, out = 1 > >
     (answer to get is E1)
```

Note that, due to the equations, D_0 is equivalent to a configuration in which message `put` is migrated into the subconfiguration:

```
D1 = < B1 : UBuffer | max = 2,
                      conf = (to B1 put E3)
                             < B1 : UBuffer | cont = E2 E1,
                                              in = 2, out = 0 > >
     (get B1 replyto R)
```

and analogously:

```
D1 = < B1 : UBuffer | max = 2,
                      conf =  (get B1 replyto R)
                              < B1 = UBuffer | cont = E2 E1,
                                               in = 2, out = 0 > >
     (put E3 into B1)
```

Due to function filter, C_0 subsumes the configurations C_1 and C_2.

Let us discuss as an example the inheritance of the formula *StateChange*. The state invariant *SI* of an unbounded buffer is given by

$$\texttt{length(L)} = \texttt{I} - \texttt{O} \wedge 0 \leq \texttt{I} - \texttt{O}$$

where the state of the buffer is $\texttt{< B : BdBuffer} \mid \texttt{in = I, out = O, cont = L >}$. The formula *StateChange* for an unbounded buffer is given by

```
StateChange(B) =
(νX.(∀I, O∈Nat, L∈List, E, E'∈Elem :
       ("< B : UBuffer | in = I, out = O, cont = L >"
    ∧  SI(< B : UBuffer >)
    ⇒  [(put E into B)]
           ("< B : UBuffer | in = I + 1, out = O, cont = E L >"
           ∧X))
 ∧     ("< B : UBuffer | in = I, out = O, cont = L E' >"
    ∧  SI(< B : UBuffer >)
    ∧  I − O > 0
    ⇒  [(get B replyto U)]
           ("< B : UBuffer | in = I, out = O + 1, cont = L >"
           ∧X))))
```

Thus,

$$(A, R), C_0 \models \mathit{StateChange}(B) \text{ implies } D_i \in |\, \mathit{StateChange}(B)\,|_{(H,S),pre(\rho)\circ I_A} \text{ for } 0 \leq i \leq 1$$

$\diamondsuit$

4.4.3 Message Algebras

The rewriting calculus provides us with the expressiveness to specify how messages are processed and how they are constructed. In the message algebra in Sect. 2.2.4, several message combinators have been introduced. We are interested in the relations between classes of transition systems established by a reuse via message combinators. The message combinators introduced in Sect. 2.2.4 compose messages but do not affect the state changes triggered by these single messages. Maude provides us with the flexibility to combine less benign message combinators. Let us illustrate this with an example.

Example. We define a message combinator `|||` as follows:

```
module EXAMPLE-MESSAGE-COMBINATOR {
 import {
   protecting (ACZ-CONFIGURATION)
   protecting (RWL)
 }

 signature {
   op _ ||| _ : Message Message   -> Message
   op f       : ACZ-Configuration -> ACZ-Configuration
 }

 axioms {
  vars m1 m2           : Message
  vars c1 c c2 d1 d2 h : ACZ-Configuration

  crl [Par]: (m1 ||| m2) c1 c c2 => f(d1) f(d2)
             if ((m1 c1 c) ==> (d1 h))  and
                ((m2 c c2) ==> (d2 h)) .
  }
}
```

`|||` specifies a synchronous transition in which message `m1` and `m2` synchronize on some common part of a configuration (denoted by `c` in the transition rule). Some part of the resulting configuration, which is common to both transitions (denoted in the rule by `h`), may not be part of the resulting composed configuration. Hereby, we do not specify precisely how much of the common part is to be neglected in the composition of the resulting state, since we do not specify that `h` is, e.g., the maximal common part of the configurations. Moreover, the function f manipulate the state when the transition is a result from applying rule `Par`. Thus, the result of processing `m1|||m2` cannot be achieved by composing the processing of `m1` and `m2` by the rules of the calculus introduced in Sect. 2.1.3.

$\diamondsuit$

This example illustrates the expressiveness of Maude as well as the variety of relations between reused and reusing specification when using the reuse concepts of message combinators and algebras. We are able to specify message combinators which change the state of the objects in a way that cannot be achieved by a combination of applications of (uncombined) transition rules. Such message combinators alter the properties of the objects involved in an arbitrary way and, thus, we cannot make a general statement about which properties are inherited via message combinators. However, we are interested in message combinators which compose messages and transitions only, without altering their semantics. Examples of such message combinators are the ones of the specification `MSG-ALGEBRA`.

We require bisimilarity for the transitions and the states of messages. To relate a composed transition to a number of transitions, we introduce a transitivity rule in the calculus:

$$c \xrightarrow{a \cup b} e \text{ if } c \xrightarrow{a} d \text{ and } d \xrightarrow{b} e$$

Let a and b be multisets of labels (messages). The transitivity rule abstracts from the order, in which the messages have been processed by uniting the two multisets of labels to one multiset.

Definition 4.32 (Inheritance via message combinators) Let $\Sigma_A = (S_A, \leq_A, OP_A)$ and $\Sigma_H = (S_H, \leq_H, OP_H)$ be two coherent order-sorted signatures, $Sp_A = (\Sigma_A, E_A, T_A)$ and $Sp_H = (\Sigma_H, E_H, T_H)$ two order-sorted specifications and $\sigma : \Sigma_A \rightarrow \Sigma_H$ the canonical injection.

Let $f_1 :$ $\texttt{Message}^{n_1}$ `->` `Message`,
$\ldots,$
$f_n :$ $\texttt{Message}^{n_1}$ `->` $\texttt{Message} \in \Sigma_H \setminus \Sigma_A$
be message combinators.

Sp_H inherits via message combinators $f_1, \ldots, f_n$ from Sp_A, or Sp_A is *an ancestor via message combinators* $f_1, \ldots, f_n$ from Sp_H, if

$$\begin{aligned}
&(\forall (H,S) \in Mod(Sp_H) : (\exists (A,R) \in Mod(Sp_A), \rho, \rho' \subseteq H \times A : \\
&\qquad H|_\sigma = A \\
&\quad \wedge \quad (H,S) \approx_\rho (A,R) \\
&\quad \wedge \quad (H,S) \precsim_{\rho'} (A,R)))
\end{aligned}$$

$$\begin{array}{llll}
\text{where} & C = \text{filter}(D) & \text{if} & (D,C) \in \rho \quad \text{and} \\
\text{and} & C = \text{filter}(D) & \text{if and only if} & (D,C) \in \rho'
\end{array}$$

$$\begin{array}{lcl}
\text{filter}(\texttt{eps}) & = & \texttt{eps} \\
\text{filter}(C_1 C_2) & = & \text{filter}(C_1)\ \text{filter}(C_2) \\
\text{filter}(f_i(m_1, \ldots m_{n_i})) & = & \text{filter}(m_1)\ \ldots\ \text{filter}(m_{n_i}) \text{ for } f_i \in \{f_1, \ldots, f_n\} \\
\text{filter}(f(m_1, \ldots m_{n_i})) & = & f(\text{filter}(m_1)\ \ldots\ \text{filter}(m_{n_i})) \text{ for } \texttt{f} \notin \{f_1, \ldots, f_n\} \\
\text{filter}(\texttt{<}o : C \mid \{a = v\}\texttt{>}) & = & \texttt{<}o : C \mid \{a = \text{filter}(v)\}\texttt{>} \\
\text{filter}(m(p_1, \ldots, p_n)) & = & m(\text{filter}(p_1), \ldots, \text{filter}(p_n)) \text{ if } m \text{ atomic}
\end{array}$$

Let us motivate this relation between the reusing and the reused specification. We require that the heir specification contains at least the sorts and operation symbols of the ancestor specification and ensure this by the canonical injection σ at the level of signatures and by the reduct at the level of algebras.

We have two different relations, a simulation and a bisimulation relation, to model the inheritance relation via message algebras.

The simulation relation (ρ') abstracts from the message combinators and relates all states with the same objects and the same messages, regardless, whether they are part of a composed message or whether they are "simply" part of the configuration. The simulation relation inherits the properties which are modeled as $\langle\,\rangle$-free formulas.

The bisimulation relation (ρ) relates—like the simulation relation—also states, which consist of the same objects and messages. But, in contrast to simulation relation ρ', it takes also into account that the composed messages are accepted—provided the state without the operators accepts the uncomposed messages. This relation inherits all five formula schemata.

Function filter relates states of the ancestor and the heir specification, provided they consist of the same objects and the same messages. Note that filter itself establishes the simulation relation, while bisimulation relation ρ is a subset of simulation relation ρ'.

We relate configurations which are equal with respect to the objects and messages of the old signature and abstract from the message combinators which are particular to the heir specification. Note that we have to use the notation of Sect. 4.3 since the formula "`m`" does not apply to composed messages containing m.

Proposition 4.33 (Inheritance of properties via message combinators) Let (Σ_A, Σ_H, Sp_A, $\mathtt{Sp}_H$, $\sigma : \Sigma_A \to \Sigma_H$, ρ, ρ' and
f_1 : $\mathtt{Message}^{n_1}$ `-> Message`,
. . . ,
f_n : $\mathtt{Message}^{n_1}$ `-> Message`
be as in Def. 4.32.

Choose $(A, R) \in \mathrm{Mod}(Sp_A)$ and $(H, S) \in \mathrm{Mod}(Sp_H)$. Choose ρ and ρ' according to Def. 4.32.

1. Let ϕ be one of the five formula schemata *Persistence*, *State*, *StateChange*, *Synchronization* and *AnswerMessages*. Then

$$(H, S) \approx_\rho (A, R) \text{ implies } ((A, R), C \models \phi \text{ implies } D \in |\,\phi\,|_{(H,S),pre(\rho)\circ I_A})$$

   for $(D, C) \in \rho$ and $w \in pre(\rho)(v)$.

2. Let ϕ' be one of the four formula schemata *Persistence*, *State*, *StateChange* and *AnswerMessages*. Then

$$(H, S) \precsim_{\rho'} (A, R)$$

implies

$$(A, R), C, v \models \phi' \text{ implies } D \in |\phi'|_{(H,S),pre(\rho')\circ I_A}(w)$$

for $(D, C) \in \rho$ and $w \in pre(\rho)(v)$.

Proof.

1. The proof that $pre(\rho)$ is consistent with I_A follows the proof of Prop. 4.31.

$$\begin{array}{ll} & (H,S) \approx_\rho (A,R) \\ \Rightarrow & \{ \text{ Cor. 4.22 } \} \\ & (A,R), C, v \models \phi \Rightarrow D \in |\phi|_{(H,S),pre(\rho)\circ I_A}(pre(\rho)(v)) \text{ for } (D,C) \in \rho. \end{array}$$

2. The proof that $pre(\rho)$ is consistent with I_A follows the proof of Prop. 4.31.

$$\begin{array}{ll} & (H,S) \precsim_{\rho'} (A,R) \\ \Rightarrow & \{ \text{ Def. 4.9 } \} \\ & (H,S) \sqsubseteq_{(post(\rho'),\widetilde{pre}(\rho'))} (A,R) \\ \Rightarrow & \{ \text{ Thm. 4.21 } \} \\ & (A,D), C, v \models \phi' \Rightarrow D \in |\phi'|_{(H,S),pre(\rho')\circ I_A}(pre(\rho')(v)) \text{ for } (D,C) \in \rho'. \end{array}$$

$\square$

These results on the inheritance of properties by message combinators stress the expressiveness of message combinators and message algebras and their relevance in object-oriented concurrent languages. Provided that message combinators compose the messages without altering the state changes they trigger, they are a both powerful and benign construct for reusing properties in concurrent object-oriented languages.

Let us illustrate the inheritance via subconfigurations with an example.

Example. Consider specification `BD-BUFFER` and message algebra `MSG-ALGEBRA` of Sect. 2.2.4. In specification `BD-BUFFER-2+`, a bounded buffer reacts also to a message `get2`—similar to specification `BD-BUFFER-2` of Sect. 2.3.2. Note that we do not define a new subclass of class `BdBuffer`.

```
module BD-BUFFER-2+ {
  import {
    protecting (BD-BUFFER)
    protecting (MSG-ALGEBRA)
  }

  signature {
    op get2 _ replyto _                : ObjectId ObjectId -> Message
```

```
    op to _ answer to get2 is _ and _ : ObjectId Elem Elem -> Message
  }

  axioms {
   vars B U  : ObjectId
   vars E E' : Elem
   var  ATTS : Attributes

eq [E1]: (get2 B replyto U)
        < B : BdBuffer | ATTS >
        = < B : BdBuffer | ATTS >
          ((get B replyto U) ;; (get B replyto U)) .

eq [E2]: ((to U answer to get is E) ;; (to U answer to get is E'))
        = (to U answer to get2 is E' and E) .
  }
}
```

Consider again configuration C_0, which contains a bounded buffer with capacity 2 and two elements `E1` and `E2` being put into the buffer. There are two `get` messages and one `put` message in the configuration.

$C_0 =$ `< B1 : UBuffer | cont = E2 E1, in = 2, out = 0, max = 2 >`
`(put E3 into B1)`
`(get B1 replyto R)`
`(get B1 replyto R)`

There are again several configurations D_i which inherit from C_0. First of all, C_0 is a valid configuration and, thus, the first configuration D_0 is just C_0:

$D_0 = C_0$

The two `get` messages can be composed sequentially:

$D_1 =$ `< B1 : BdBuffer | cont = E2 E1, in = 2, out = 0, max = 2 >`
`(put E3 into B1)`
`(get B1 replyto R);;(get B1 replyto R)`

which is equivalent to:

$D_2 =$ `< B1 : BdBuffer | cont = E2 E1, in = 2, out = 0, max = 2 >`
`(put E3 into B1)`
`(get2 B1 replyto R)`

The message `put` and the two `get` messages can be composed sequentially. Note that, in this configuration no transition is possible since the bounded buffer is full:

$D_3 =$ `< B1 : BdBuffer | cont = E2 E1, in = 2, out = 0, max = 2 >`
`(put E3 into B1) ;; (get B1 replyto R) ;; (get B1 replyto R)`

The resulting configuration of all the states $D_0 \ldots D_3$ is either:

$F_0 = E_0$

or:

$F_1 =$ `< B1 : BdBuffer | cont = E2 E1, in = 2, out = 0, max = 2 >`
`(to U answer to get2 is E1 and E2)`

Let us consider two formula schemata, *Synchronization* and *StateChange*:

- *Synchronization* is inherited from C_0 to D_0, D_1 and D_2 but *not* to D_3.
- *StateChange* is inherited from C_0 to D_0, D_1, D_2 and to D_3.

Note that the basic proposition "`(get B replyto U)`" is interpreted by the standard interpretation as "a configuration containing a message `get`". $\rho(I_A($"`(get B replyto U)`"$))$ is the set of configurations which contain either a `get` message as an atomic message of a `get` message as a component of a composed message.

$\diamondsuit$

4.4.4 Inheritance of Properties—The Upshot

In the previous subsections, we have established a link from reuse at the syntactic level of Maude and the reuse constructs, to reuse at the semantic level and reuse at the property-oriented level of Maude.

We distinguish three kinds of reuse:

1. via inheritance,
2. via subconfiguration,
3. via a message combinator.

For each of these kinds of reuse, we have a relation between classes of algebras, a reuse construct of Maude and properties, phrased in the μ-calculus.

In Sect. 2.3, we have already discussed the expressiveness of these reuse constructs. Together they are *powerful* enough to circumvent the inheritance anomaly. The upshot of the previous three subsections is that the are also *safe* kinds of reuse, since we can reflect the syntactic reuse at the semantic level by an operation on the classes of algebras, which are the model of our specifications. Moreover, this relation can be lifted from the operational level of Maude to the property-oriented level of the modal μ-calculus. We know which classes of formulas are preserved by the three kinds of reuse. This suggests that these constructs are not only safe but also *adequate* for the property-oriented level of a specification language, when one would like to achieve presumably not reuse of specification text but reuse of properties.

The fact that we are able to establish a connection between reuse constructs, operations on classes of algebras and μ-formulas proves that Maude and this set of reuse concepts form a well-designed language. Moreover, the technique of abstract interpretation which we apply to the reuse concepts suggests (1) that the relations we use are the loosest ones which ensure that the properties which we consider to be relevant in verification and refinement are preserved and (2) that stronger properties cannot be preserved by these reuse concepts which we consider to be appropriate from the software engineering point of view. Our conclusion is that Maude with our concepts of reuse is as a language appropriate and well-designed. We suggest that the object model of Maude and our reuse concepts are also appropriate for a concurrent object-oriented programming language since the gap between our specification language, Maude, and a programming language is due to Maude's operational specification style quite small.

4.5 Related Work

We have to deal with verification of Maude specifications, since the properties that can be expressed directly by a Maude specification are limited. More property-oriented languages like, e.g., TROLL [HSJ+94, SF91, FM96] use temporal logics for the specification of the behavior of objects. Thus, TROLL is more expressive and abstract as well as more restricted in its concurrency and its communication patterns. Thus, there is much less necessity to verify TROLL specifications.

Verification of object-oriented languages focuses in most approaches on the intra-object level. One reason for this is that sequential languages—and even most concurrent languages—have a fixed communication scheme, which does not allow to specify individual communication and synchronization patterns. Thus, one is less interested in the properties of the behavior of a collection of objects. Verification at the intra-object level is done for OOSpectrum in [WNL95], for $\pi o \beta \lambda$ in [Jon93a, HJ96, NS96, Wed93] and for an actor-like language in [BMS96]. For verification at an inter-object level with a small amount of concurrency and fixed communication pattern, see [HJ96, NS96].

There is a second reason why most approaches focus on the intra-object level: objects are likely to be small. Verification at the inter-object level soon becomes very complex. Particular to our approach is the use of abstraction techniques from abstract interpretation [CC78, LGS+95, Ste93, SCK+95, SMC96, SKV96] in the verification of object-oriented concurrent entities.

We have used two different formalisms and techniques in object orientation: invariants as properties of states and the μ-calculus as a notation for properties of the behavior.

The configuration invariants were introduced in [WNL95] to describe well-formed states. Paricular to our approach is that we we address concistency of the global state and introduce invariants as a feasible means to phrase such properties. Particular is also that we separate the two issues of object-oriented specification and properties of the global state which cannot be phrased in the object-oriented specification style. In [Jon93a] properties of the global state are part of the precondition of messages. Assumption-commitment specifications use "global" properties for the specification of procedures [AM95]. In Sect. 5.2.5, we use a similar technique in refinement. For us the configuration invariants and the properties that they ensure are prerequisites for sophisticated techniques in verification.

As a notation for the properties of Maude specifications we have chosen the μ-calculus [Koz83, Bra92, Sti92] and our relation between abstract and concrete specifications is (bi)-simulation. This enables us to use the results of abstract interpretation for concurrent systems from [Bra92, Bru93, CC78, Dam93, DGG94, LGS+95, SCK+95, SMC96, SKV96] in our work. Particular to our approach are the results on the inheritance of propertiesi.e., the relation between the operational and the property-oriented level of description of object-oriented concurrent systems.

4.6 Remarks and Discussion

In most approaches verification of object-oriented programs is limited to the intra-object level. But, in the object-oriented concurrent paradigm, the inter-object level is both important and interesting, in particular, when the language admits as much concurrency as Maude does. Here, the object-oriented paradigm has drawbacks compared with other concurrent specification languages. A program consists of a collection of objects which are implemented independently. When a collection of objects is being composed, the behavior of these objects has to be investigated. We focus accordingly in our work on the inter-object level and the only "property" of our serialized objects are the data that form the state.

The property of the airport specification, which we have proved, is rather simple. It is obvious that it holds for the abstract specification. To prove more complex properties of Maude specifications model-checking algorithms [CES86, SCK+95, SMC96, SKV96] and support tools would be helpful [SCM95]. Generally, the verification of program properties is cumbersome. Thus, refinement is preferred. Note that the formal techniques we have used in verification will be used in refinement as well.

The first properties we have introduced are configuration and robustness invariants. The necessity for such properties stems from the object-oriented design principle. The classes are developed rather independently and, when their instances are put together it is not sufficient to assume soundness of the classes, which can be ensured by the implementors of the classes, but we need additionally the soundness of the global state. Thus, although these invariants as well as the sort constraints are not a part of the object-oriented design philosophy, they are tremendously useful in practice and often a prerequisite for the verification of complex properties of behavior, as all our examples show. We use the invariants also to ensure the correctness of single objects. This is not strictly necessary since the correctness of the classes could be ensured. But, on the other hand, invariants can hardly be inherited, since they are very vulnerable to extensions of existing code. Furthermore, both global and local invariants bring difficulties in the implementations, in particular, for the Rewrite Rule Machine. Let us explain this in more detail. The Rewrite Rule Machine is designed to execute rewriting rules. Rewriting rules, both equations and transition rules, can determine for a given set of objects and messages that they are able to perform a transition. Computing a value that depends on the global state like, i.e., an invariant, can hardly be done in parallel and inspecting the whole state would require to stop all concurrent rewrite processes. In case the state is considerably large, it would require also that parts of the state which are not in main memory would have to be swapped. Altogether, the concept of invariants would be very expensive with respect to the performance of the Rewrite Rule Machine. Thus, the invariants have been sacrificed to the object-oriented programming paradigm and the executability of Maude.

Verification is expensive and verification at the inter-object level deals with complex systems. We introduce an abstraction technique, which helps to diminish the complexity of the verification task considerably. Here, subconfigurations can be useful also in reducing the complexity of verification. Moreover, the use of subconfigurations helps to simplify a specification, as demonstrated in Sect. 2.4.3. In general, the concepts for structuring and reuse help to reduce the complexity of verification tasks and allow to inherit properties and, possibly, proofs as well.

In this chapter, we have characterized the classes of properties that are inherited via our reuse constructs. These classes of properties are naturally quite limited, as in all other extensions of existing object-oriented or conventional code. The important implication of inheritability of the property that “transitions may happen” via the inheritance relation and other properties via the other reuse concepts is that Maude is a well-designed language. We would like to stress that other, richer object-oriented concurrent specification or programming languages offer neither reuse concepts as rich as ours nor expose the correspondence of inheritance of properties and reuse of code.

Chapter 5

Refinement

Refinement is used in computer science for a variety of relations between descriptions, specifications or programs. In this chapter, we follow the philosophy of a refinement by which a program is derived from a specification. Here, it is paramount that one does not change the properties of this specification in the refinement process.

Refinement is dual to verification. While verification deals with proving abstract properties for concrete programs, in a refinement one tries to infer a program from its characteristic properties. The formal framework of (α, γ)-(bi)simulation relations developed in Chap. 4 is reused in this chapter.

We are interested in the refinement of an abstract specification of a concurrent system to an implementation in a concurrent object-oriented language. In Maude, we have very abstract communication and synchronization mechanisms as well as a very abstract view of objects. Typical object-oriented programming languages provide point-to-point communication and several object-oriented language constructs, which are difficult to model in a formal way [Wal94, Jon93d, Wed93]. Since formal methods for object-oriented programming languages demonstrate that dealing with the object-oriented constructs is very cumbersome, we do not become so concrete. Our most concrete level is a Maude specification, using the communication mechanisms typical for object-oriented programming languages.

Although formal refinement is considered to be less expensive than verification, it is still expensive. Thus, we believe that, in practice, only parts of a complex system will be specified, verified and finally refined formally to a program. The concepts developed in Sect. 4.3 provide a means for structuring a specification or program such that the parts for which formal methods are used and the other parts which are only validated do not interfere.

Remember that we have described object-oriented specification to consist of three different views: the specification of the properties and of the observable behavior of a class, the specification of communication and synchronization between objects, and the specification of a system consisting of objects. Accordingly, the refinement of object-oriented specifications can follow these three different views: refinement of a single class, refinement of a collection of objects, and refinement of a complete system modeled in an object-oriented language. We dedicate a section to each of these views on specification and refinement. Each of them needs a different, appropriate refinement relation and refinement techniques.

In the first section, we deal with the refinement of a single class. When refining a single class, we are interested in how object-oriented languages of different levels of abstraction express the properties of an object. It is paramount that a refinement of a single class does not change the behavior of the overall system. Thus, the refinement relation is very fine and not the properties, but the implementation is refined. We proceed in several steps from a property-oriented specification in the μ-calculus to a specification using formula schemata in the μ-calculus (see Sect. 4.4) and on to a specification in Maude. As an example, we use the bounded buffer.

The second section is dedicated to the refinement of communication patterns: here, not the language and the implementation of the classes, but the communication between the objects is of interest. The goal of such a refinement is, again, not to change the properties of the overall system. As an example, we use the specification of a protocol, which we refine from an abstract, hard-to-implement specification in Maude to a concrete, execution-efficient specification in Maude, which is also more realistic with respect to properties of the underlying hardware.

In the third section, we deal with the refinement of a whole system. This view is rather different from the refinement of a class and the refinement of communication patterns. The goal of a refinement at this level is an execution-efficient program. The kind of refinement depends solely on the properties to be preserved. Thus, very fine as well as very coarse refinement relations may be appropriate. Since abstraction is a necessity for refinement of complex systems, we use behavioral refinement. At this level, we are not interested in the implementation of single objects and use therefore Maude as our notation for abstract and concrete levels of specifications. Here, the example is the airport specification.

5.1 Refinement of Classes

Maude has been developed especially for the object-oriented specification of concurrent systems. The rewriting calculus and the underlying rewriting logic make Maude a very powerful and general specification language [Mes93a, Mes96]. Maude's advantages are its object model, its ability to combine the two paradigms of inheritance and concurrency in a sensible way and, particularly, its abstract way of specifying synchronization and communication between objects (see Sect. 2.3). But, on the other hand, Maude's semantics is operational and, thus, not really property-oriented, and the transition rules specifying the behavior of objects are not powerful enough to express, e.g., safety properties. The second, more property-oriented specification language is the μ-calculus, which is able to express safety and liveness properties of the behavior of a concurrent system [Bra92, Koz83, MPW93]. In Sect. 4.1, we have discussed the relation between the two languages.

When refining a specification of a class from a property-oriented specification to an executable specification or program, the refinement should ensure that the properties of a system in which such an object is used are not changed. Thus, we need a refinement relation that ensures substitutability. In Chap. 4, it is stated that only a bisimulation relation between finite transition systems preserves all properties, which the μ-calculus can express. Coarser relations between transition systems preserve only certain classes of μ-formulas. We characterize the refinement relation by implication. It abstracts from the internal states and preserves the observable properties.

This refinement from an abstract to a concrete specification is reflected at the semantic level by the loose approach [Wir90]. In each refinement step, the set of models of the specification becomes smaller. The last, most concrete step in such a refinement typically yields a singleton set of models—a program. In this chapter, all specifications are at a very high level of abstraction: our abstract, property-oriented specification language is the μ-calculus, our concrete, executable language the specification language Maude. Let us briefly explain the relation between the specifications, particularly, between specifications in different languages. The specifications have a basic signature and also basic data types, specified by equations in common. Thus, the order-sorted algebra A and, in particular, the states, i.e., the terms of sort `Configuration` (abbreviated `Cf`) are the same. Different is the level of abstraction in the language for specifying the behavior of objects, but common to these languages is the transition system: the semantics of a Maude specification is a transition system and the μ-properties are verified for the transition system.

We use three levels of specification with different degrees of abstraction:

1. At the *abstract* level, we use the language of modal μ-formulas for specification. A typical specification would describe, e.g., the sequences of messages an object or a collection of objects accepts. Invariants on states restrict the transition system such that all states obey certain well formedness conditions.

2. At the *intermediate* level, we use again the language of μ-formulas. The propositions on states we introduce describe the states of the objects. At this level, the formulas have a very rigid structure. The behavior of the objects belonging to a common class is a conjunction of five formulas specifying (1) that objects are persistent, (2) the consistent state of an object, (3) the synchronization code determining which messages are accepted depending on the local state of an object, (4) the state changes of the objects and (5) the answer messages generated.

3. At the *concrete* level, we use Maude as our specification language. At this point, Maude itself provides us with a choice of the degree of abstraction, using, e.g., implicit synchronous or explicit asynchronous communication.

We give three specifications, Φ_A, Φ_M and Sp_C, of a bounded buffer, each at a different level of abstraction, such that

$$\Phi_A \rightsquigarrow \Phi_M \rightsquigarrow Sp_C$$

5.1.1 The Refinement Relation

In the process of stepwise refinement, an abstract (requirement) specification is transformed into a concrete specification, which might be a program. We use the loose approach to refinement [Wir90]. In the process of stepwise refinement, implementation details of data types and algorithms are added to the specification. This reduces the class of models (in several steps) to a singleton set—a program.

Definition 5.1 Let $\Phi = \{\phi_1, \ldots \phi_n\}$ and $\Phi' = \{\phi'_1, \ldots \phi'_{n'}\}$ be μ-specifications.
Φ is refined by Φ', written $\Phi \rightsquigarrow \Phi'$, iff $\phi'_1 \wedge \ldots \wedge \phi'_{n'} \Rightarrow \phi_1 \wedge \ldots \wedge \phi_n$.
Let $\Phi' = \{\phi'_1, \ldots \phi'_{n'}\}$ be a μ-specification, and $Sp = (\Sigma, E, T)$ a Maude specification. Φ' is refined by Sp, written $\Phi' \rightsquigarrow Sp$, iff $Mod(Sp) \models \phi'_1 \wedge \ldots \wedge \phi'_{n'}$.

5.1.2 The Abstract Level—Invariants and the μ-Calculus

Relevant at the most abstract level in the specification of a single class are only two issues:

- *What is a bounded buffer?*
- *How does a bounded buffer behave?*

The question "What is a bounded buffer?" can be answered by specifying a property each bounded buffer has to satisfy:

$$State(\texttt{B}) = \text{``}\texttt{< B : BdBuffer >}\text{''} \Rightarrow 0 \leq \texttt{length(B.cont)} \leq \texttt{B.max}$$

The number of elements stored in a buffer is `length(B.cont)`. It is not larger than `B.max`, the maximal number of elements of a buffer.

At this abstract level, we do not give the implementation of the state but, instead, we require that a buffer has certain properties: it is able to determine the length of its contents and it stores only the maximum number of elements.

"What is a bounded buffer?" is answered by specifying properties of the state. This is usually not done in the μ-calculus. The specification of "How does a bounded buffer behave?" states properties that do not involve the state but the interactions of objects via messages. For specifying this aspect, we use the modal μ-calculus. (Recall the discussion on the construction of the state and the observation of objects in Sect. 3.6.)

So, a general approach to answering "what is ... "? is to give basic properties and functions (`cont`, `max`, `length`) as well as invariants for objects and configurations (here we have only an invariant *State* for one object). The answer to "how does ... behave?" gives possible actions (or messages) of an object together with allowed or required (sequences of) actions.

If predicate *State* holds in some state for a buffer `B`, then it holds in all subsequent states for `B`:

$$\phi_1(\texttt{B}) = State(\texttt{B}) \Rightarrow (\nu X.State(\texttt{B}) \wedge [-]X)$$

The notation $[-]$ is an abbreviation for $[L]$, where L is the set of all possible actions. We do not care what happens with buffers that are in an inconsistent state. ϕ_1 ensures that a consistent buffer remains consistent. A bounded buffer accepts a `put` or a `get` and possibly both messages:

$$\begin{aligned}\phi_2(\mathtt{B}) = \text{“}\mathtt{<\ B\ :\ BdBuffer\ >}\text{”} \Rightarrow\\ (\nu X.\, (\forall \mathtt{E} \in \mathtt{Elem}, \mathtt{U} \in \mathtt{ObjectId} :\\ (\langle \mathtt{(put\ E\ into\ B)} \rangle X \vee \langle \mathtt{(get\ B\ replyto\ U)} \rangle X)))\end{aligned}$$

After an element has been put into the buffer, there is a sequence of `get` messages such that an `answer` carries the element. The result of a `get` message is an answer message that is part of the configuration waiting to be processed from, maybe, the object that sent the `get` message. At this point, we apply already the asynchronous message passing mechanism of Maude to make a later refinement feasible:

$$\begin{aligned}\phi_3(\mathtt{B}) = \text{“}\mathtt{<\ B\ :\ BdBuffer\ >}\text{”} \Rightarrow\\ (\nu X.\, (\forall \mathtt{E} \in \mathtt{Elem}, \mathtt{U} \in \mathtt{ObjectId} :\\ [\mathtt{(put\ E\ into\ B)}]\\ (\mu Y.\, \langle \mathtt{(get\ B\ replyto\ U)} \rangle\\ (Y \vee \text{“}\mathtt{(to\ U\ answer\ to\ get\ is\ E)}\text{”})) \wedge X))\end{aligned}$$

After storing two elements in a buffer, the element stored first is retrieved before the one stored second (FIFO):

$$\begin{aligned}\phi_4(\mathtt{B}) = \text{“}\mathtt{<\ B\ :\ BdBuffer\ >}\text{”} \Rightarrow\\ (\nu X.\, (\forall \mathtt{E1}, \mathtt{E2} \in \mathtt{Elem}, \mathtt{U} \in \mathtt{ObjectId} :\\ [\mathtt{(put\ E1\ into\ B)}][\mathtt{(put\ E2\ into\ B)}]\\ (\mu Y.\, \langle \mathtt{(get\ B\ replyto\ U)} \rangle\\ (Y \vee (\text{“}\mathtt{(answer\ to\ get\ is\ E1)}\text{”}\\ \wedge \langle \mathtt{(get\ B\ replyto\ U)} \rangle\\ \text{“}\mathtt{(answer\ to\ get\ is\ E2)}\text{”}))) \wedge X))\end{aligned}$$

The abstract specification of a bounded buffer is $\Phi_A = \{\phi_1, \phi_2, \phi_3, \phi_4\}$. This set of formulas is just one suggestion for specifying the behavior of a bounded buffer. Naturally, one could consider entirely different sets of formulas.

5.1.3 Intermediate Level—Formula Schemata

At this level of abstraction, we make a decision about the implementation of the internal state of a buffer, namely, that the internal state can be observed by calls of the methods `in`, `out`, `max` and `cont`. Thus, an observation is represented by “`< B : BdBuffer | in = I, out = O, max = M, cont = L >`”. Implicitly, also the object model manifests itself in the structure of the formulas.

Five formulas determine the behavior of a class. Each formula corresponds to a certain view. We have two internal views, which specify consistent states' properties, and the state changes induced by the object. We have a property stating that objects are persistent and views for two interfaces: the answer messages that are produced and the link between the incoming messages and the state changes.

Note that the formula schemata were already given in Sect. 4.4. We repeat them here and instantiate them for the bounded buffer subsequently.

Definition 5.2 Let `C` be a classname and `atts` resp. `atts'` denote the attributes with their values of class `C`. Let $SI($`< B : C | atts`$_i$ `>`$)$, $\phi_i($`< B : C | atts`$_i$ `>`$)$ and $\psi_i($`< B : C | atts`$_i$ `>`$)$ be propositions on the state of an object `B` of class `C`. Let $\mathtt{a}_i$ be a message and let p be all free variables in the formulas with their range P.

We define five formula schemata for a class `C` with n methods:

$$\begin{array}{l} \mathit{Persistence}(B) = (\nu X.(\forall p \in P : \\ \quad \text{“}\mathtt{<\ B\ :\ C\ >}\text{”} \Rightarrow [-](\text{“}\mathtt{<\ B\ :\ C\ >}\text{”} \wedge X))) \end{array}$$

$$\begin{array}{l} \mathit{State}(B) = (\nu X.(\wedge p : p \in P : \\ \quad \mathit{SI}(\text{“}\mathtt{<\ B\ :\ C\ |\ atts\ >}\text{”}) \Rightarrow [-](\mathit{SI}(\text{“}\mathtt{<\ B\ :\ C\ |\ atts'\ >}\text{”}) \wedge X))) \end{array}$$

$$\begin{array}{l} \mathit{Synchronization}(B) = (\nu X.(\forall p \in P : (\wedge i : 1 \leq i \leq n : \\ \quad \text{“}\mathtt{<\ B\ :\ C\ |\ atts\ >}\text{”} \wedge \mathit{SI}(\mathtt{<\ B\ :\ C\ |\ atts\ >}) \wedge \psi_i(\mathtt{<\ B\ :\ C\ |\ atts\ >}) \\ \qquad \Rightarrow \langle m_i \rangle X))) \end{array}$$

$$\begin{array}{l} \mathit{StateChange}(B) = (\nu X.(\forall p \in P : (\wedge i : 1 \leq i \leq n : \\ \quad \text{“}\mathtt{<\ B\ :\ C\ |\ atts\ >}\text{”} \wedge \mathit{SI}(\mathtt{<\ B\ :\ C\ |\ atts\ >}) \wedge \psi_i(\mathtt{<\ B\ :\ C\ |\ atts\ >}) \\ \qquad \Rightarrow [m_i](\phi_i(\mathtt{<\ B\ :\ C\ |\ atts}_i\mathtt{'\ >}) \wedge X)))) \end{array}$$

$$\begin{array}{l} \mathit{AnswerMessages}(B) = (\nu X.(\forall p \in P : (\wedge i : 1 \leq i \leq n : \\ \quad \text{“}\mathtt{<\ B\ :\ C\ |\ atts\ >}\text{”} \wedge \mathit{SI}(\mathtt{<\ B\ :\ C\ |\ atts\ >}) \wedge \psi_i(\mathtt{<\ B\ :\ C\ |\ atts\ >}) \\ \qquad \Rightarrow [m_i](\text{“}\mathtt{a}_i\text{”} \wedge X)))) \end{array}$$

Note that the formula schemata as we present them here are not in the format of μ-formulas as presented in Sect. 4.1. However the predicates, which describe relations between values could be part of the description of the range of the variable.

A rather basic safety property of a bounded buffer is its persistence which says that, if a bounded buffer it is part of a configuration, it is also part of all successor states:

$$\begin{array}{l} \mathit{Persistence}(B) = \\ (\nu X.\text{“}\mathtt{<\ B\ :\ BdBuffer\ >}\text{”} \Rightarrow [\mathtt{-}]\text{“}\mathtt{<\ B\ :\ BdBuffer\ >}\text{”} \wedge X) \end{array}$$

The formula *State* specifies an invariant of the internal state of an object. If *State* holds for one state then it holds for all successor states. It ensures the consistency of the internal state. We do not consider bounded buffers which are in an inconsistent state. At this level, we make the design decision that the state of a bounded buffer is implemented by four attributes, namely `in`, `out`, `max`, and `cont`:

$$
\begin{array}{l}
\mathit{State}(B) = \\
(\nu X.(\forall \mathtt{I},\mathtt{I'},\mathtt{O},\mathtt{O'} \in \mathtt{Nat}, \mathtt{M} \in \mathtt{NzNatL}, \mathtt{L'} \in \mathtt{List} : \\
\quad (\text{“}\mathtt{<\ B\ :\ BdBuffer\ |\ in\ =\ I,\ out\ =\ O,\ max\ =\ M,\ cont\ =\ L\ >}\text{”} \\
\ \wedge\ \ \mathtt{length(L)} = \mathtt{I} - \mathtt{O} \\
\ \wedge\ \ \mathtt{0} \le \mathtt{I} - \mathtt{O} \le \mathtt{M}) \\
\Rightarrow [-]\ (\text{“}\mathtt{<\ B\ :\ BdBuffer\ |\ in\ =\ I',\ out\ =\ O',\ max\ =\ M,\ cont\ =\ L'>}\text{”} \\
\qquad \Rightarrow\ \mathtt{length(L')} = \mathtt{I'} - \mathtt{O'} \wedge \\
\qquad\quad \mathtt{0} \le \mathtt{I'} - \mathtt{O'} \le \mathtt{M'} \wedge X)))
\end{array}
$$

We refer to the formula $\mathtt{length(L)} = \mathtt{I} - \mathtt{O} \wedge \mathtt{0} \le \mathtt{I} - \mathtt{O} \le \mathtt{M}$ as *SI*, (state invariant, of a bounded buffer).

In formula *Synchronization*, we specify the so-called synchronization code. The synchronization code determines whether an object accept a message. In our approach, this synchronization code depends only on the state of the object:

$$
\begin{array}{l}
\mathit{Synchronization}(B) = \\
(\nu X.(\forall \mathtt{I},\mathtt{O} \in \mathtt{Nat}, \mathtt{M} \in \mathtt{NzNat}, \mathtt{L} \in \mathtt{List}, \mathtt{E} \in \mathtt{Elem}, \mathtt{U} \in \mathtt{ObjectId} : \\
\qquad (\text{“}\mathtt{<\ B\ :\ BdBuffer\ |\ in\ =\ I,\ out\ =\ O,\ max\ =\ M,\ cont\ =\ L\ >}\text{”} \\
\quad \wedge \quad \mathit{SI}(\mathtt{<\ B\ :\ BdBuffer\ >}) \\
\quad \wedge \quad \mathtt{I} - \mathtt{O} < \mathtt{M} \\
\quad \Rightarrow \quad \langle \mathtt{(put\ E\ into\ B)} \rangle X) \\
\wedge \qquad (\text{“}\mathtt{<\ B\ :\ BdBuffer\ |\ in\ =\ I,\ out\ =\ O,\ max\ =\ M,\ cont\ =\ L\ E\ >}\text{”} \\
\quad \wedge \quad \mathit{SI}(\mathtt{<\ B\ :\ BdBuffer\ >}) \\
\quad \wedge \quad \mathtt{I} - \mathtt{O} > 0 \\
\quad \Rightarrow \quad \langle \mathtt{(get\ B\ replyto\ U)} \rangle X)))
\end{array}
$$

The two formulas *StateChange* and *Synchronization* specify the internal behavior of an object. From the synchronization code, we obtain not only the conditions under which a method may be invoked but also under which preconditions this method and the functions on data types must operate correctly on the state of the object. In formula *StateChange*, we specify how methods change the state of an object. When a message is accepted, it *always* changes the state of the object in the same way. After a `put` action, the value of attribute `in` is incremented and the element which is parameter to the message is added to the contents. After a `get` message, the value of `out` is incremented and the element which is parameter to the message is added to the value of attribute `cont`:

$$
\begin{array}{l}
\mathit{StateChange}(B) = \\
(\nu X.(\forall \texttt{I}, \texttt{O}, \texttt{M}\in\texttt{Nat}, \texttt{M'}\in\texttt{Nat}, \texttt{L}\in\texttt{List}, \texttt{E}, \texttt{E'}\in\texttt{Elem}, : \\
\quad (\text{``}\texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L >}\text{''} \\
\quad \wedge\ \mathit{SI}(\texttt{< B : BdBuffer >}) \\
\quad \wedge\ \texttt{I} - \texttt{O} < \texttt{M} \\
\quad \Rightarrow [(\texttt{put E into B})] \\
\qquad \text{``}\texttt{< B : BdBuffer | in = I + 1, out = O, max = M, cont = E L >}\text{''} \\
\qquad \wedge X) \\
\wedge\ (\text{``}\texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L E' >}\text{''} \\
\quad \wedge\ \mathit{SI}(\texttt{< B : BdBuffer >}) \\
\quad \wedge\ \texttt{I} - \texttt{O} > 0 \\
\quad \Rightarrow [(\texttt{get B replyto U})] \\
\qquad (\text{``}\texttt{< B : BdBuffer | in = I, out = O + 1, max = M, cont = L >}\text{''} \\
\qquad \wedge X))))
\end{array}
$$

Messages not only change the internal state of the objects, they also trigger answer messages to be created as part of the global state:

$$
\begin{array}{l}
\mathit{AnswerMessages}(B) = \\
(\nu X.(\forall \texttt{I}, \texttt{O}\in\texttt{Nat}, \texttt{M}\in\texttt{NzNat}, \texttt{E}, \texttt{E'}\in\texttt{Elem}, \texttt{L}\in\texttt{List}, \texttt{U}\in\texttt{ObjectId} : \\
\qquad (\text{``}\texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L >}\text{''} \\
\quad \wedge \quad \mathit{SI}(\texttt{< B : BdBuffer >}) \\
\quad \wedge \quad \texttt{I} - \texttt{O} < \texttt{M} \\
\quad \Rightarrow \quad [(\texttt{put E into B})]X) \\
\wedge \qquad (\text{``}\texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L E' >}\text{''} \\
\quad \wedge \quad \mathit{SI}(\texttt{< B : BdBuffer >}) \\
\quad \wedge \quad \texttt{I} - \texttt{O} > 0 \\
\quad \Rightarrow \quad [(\texttt{get B replyto U})](\text{``}\texttt{(to U answer to get is E)}\text{''} \wedge X))))
\end{array}
$$

After message `(get B replyto U)`, message `(answer to get is E)` is part of the global state of the system. Again, we specify that after a method is invoked an answer message is *always* part of the (global) state.

The specification at the intermediate level is given by

$$
\begin{array}{ll}
\Phi_M(B) = \{ & \mathit{State}(B), \\
& \mathit{Persistence}(B), \\
& \mathit{StateChange}(B), \\
& \mathit{AnswerMessages}(B), \\
& \mathit{Synchronization}(B)\}
\end{array}
$$

5.1.4 Concrete Level—Maude

Maude's transition rules and rewriting calculus provide only the possibility to express sometime-properties on single actions, but not always-properties and not properties on sequences of actions. Each transition rule specifies one possible reaction of an object to a message. Thus, we have to refine the intermediate specification, which focuses on certain aspects of the behavior of a class, to a specification which focuses on the local and global reaction to a message.

There is one more, severe difference between the concrete level of Maude and the object-oriented μ-specification: the rules of Maude describe the global transition system, while the formula schemata in μ-calculus describe properties of a single class. Thus we abstract from the inter-object level in the specification. We implement this by abstraction from the presence of `put` and `get` messages in the configuration. We obtain the specification `BD-BUFFER-INTERN`.

```
module BD-BUFFER-INTERN {
  import {
    protecting (BD-BUFFER)
  }

  axioms {
    vars B U  : ObjectId
    var  E    : Elem

  eq [E1]: (get B replyto U) acz-empty
           = acz-empty .

  eq [E2]: (put E into B) acz-empty
           = acz-empty .
  }
}
```

Demonstrandum 5.3
$Sp_M \rightsquigarrow$`BD-BUFFER-INTERN`.

Proof. See App. C.1

□

5.2 Refinement of Communication Patterns

In the previous section, we have introduced a relation for the refinement of object-oriented specifications. The specifications and the program modeling a bounded buffer in Sect. 5.1

form an example of a refinement from an abstract level of specification to a concrete program of a single class. In this section, we deal with the refinement of a class with a particular application of the concept of refinement, the refinement of communication patterns: the synchronous transition rules of Maude provide a very abstract way of specifying synchronization and communication between a number of objects. This is convenient for specification but expensive at run time. The rules of Simple Maude [Mes93a], in which no synchronization between objects takes place, are much easier to implement and more efficient at run time, but less expressive. We go one step further and make the concrete specification not only more efficient than the abstract one, but also more realistic with respect to the properties of the hardware: messages may only contain small amounts of data, messages can be lost or duplicated and it is not guaranteed that communication preserves the order of messages being sent.

We develop a method for refining synchronous communication patterns to asynchronous communication patterns, which is based on the refinement relations of Sect. 5.1. To refine a synchronous communication pattern, to an asynchronous communication pattern, we proceed as depicted in Fig. 5.1.

Let us explain the four steps in the refinement and the five different levels of specification involved. A refinement of a single synchronous rule to a number of asynchronous rules transforms a single atomic transition to a number of transitions with possibly complex dependencies between them. An enrichment of the signature provides the capacity to store and carry the information necessary to synchronize the asynchronous transitions. This enrichment can be done when necessary—typically, it is already necessary in the first refinement step.

In the first refinement step, we enrich the signature of the abstract specification and introduce control structures (the triangle). At this level, we view the controlled transition as a single, atomic transition. These control structures help to enforce the desired relation, a combination of simulation and bisimulation between the abstract and the concrete specification.

In the second refinement step, we enrich the specification, which is the result of the first refinement step, by equations for behavioral equivalence to obtain transitions that are behaviorally equivalent to the controlled asynchronous transition of the first level. (In our example the actions `c` and `d` are commutative while `e` and `f` are not.)

In the third refinement step, we change the view, i.e., the implementation of the model of computation. We observe the controlled transition not any more as a single atomic composed transition but as a (sequence of) transitions. At this level, the actual action refinement takes place.

In the fourth refinement step, we abstract from the control terms and extract rely conditions for which the concrete specification is in a bisimulation relation with the abstract specification.

Formally, the refinement consists of two steps, a simulation and a bisimulation relation between the initial models of the specifications: $I(\mathtt{AP}) \stackrel{\prec}{\sim}_{(\alpha,\gamma)} I(Sp_{\mathrm{Control}})$ and $I(Sp_{\mathrm{Control}'}) \approx_{(\alpha,\gamma)} I(\mathtt{CP})$.

This section is organized as follows. Throughout, we use a specification of a transport

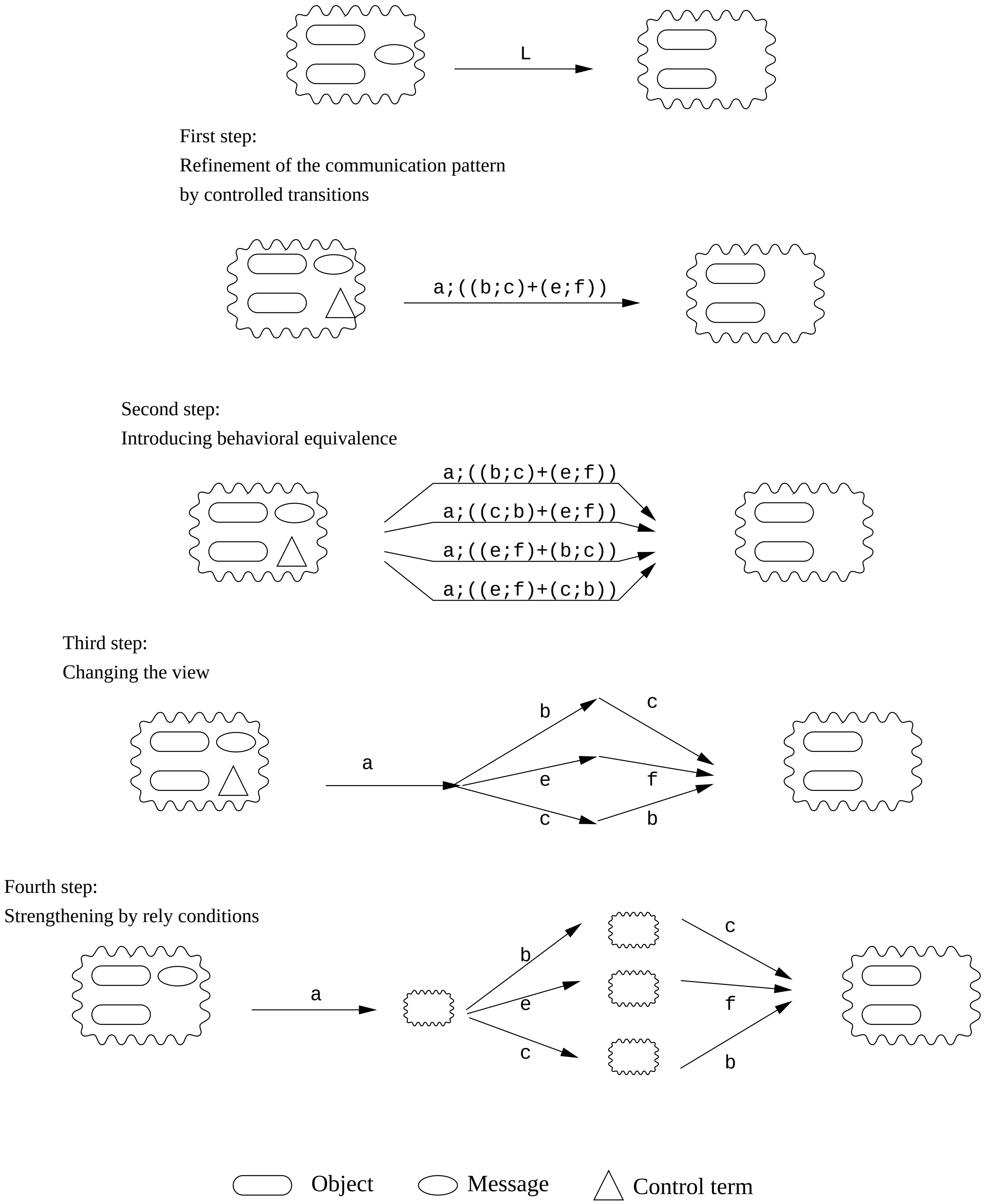

Figure 5.1: Steps in the refinement of communication patterns

protocol, as given in [Mes93a] and in a slightly modified version in [Lan96]. First, we give the specification of the two implementations of the protocol: the abstract protocol AP and the concrete protocol CP. Then, we develop the method of the refinement. Finally, we discuss how refinements of parts of a specification can be done in a large specification.

5.2.1 Abstract and Concrete Specifications of the Protocol

The information to be exchanged between a sender and a receiver is a list of contents.

At the abstract level, a sender and a receiver exchange the information in a single communication. Message (send from S to R) triggers the exchange of information.

```
module AP {
  import {
   protecting (LIST)
   protecting (ACZ-CONFIGURATION)
  }

  signature {
    class Sender {
      sendq : List
    }

    class Receiver {
      incoming : List
    }

   op send from _ to _ : ObjectId ObjectId -> Message
 }

 axioms {
  vars S R : ObjectId
  var  ML  : List
  vars ATTS ATTS' : Attributes

rl [send]: (send from S to R)
           < S : Sender | sendq = ML, ATTS >
           < R : Receiver | incoming = eps, ATTS' >
           => < S : Sender | sendq = eps, ATTS >
              < R : Receiver | incoming = ML, ATTS' > .
 }
}
```

In the concrete specification, we use asynchronous communication patterns. The protocol we use can be classified as a positive acknowledge and retransmission protocol [Sha94].

It is a version of the alternating bit protocol. It uses natural numbers to identify the packages. Our implementation is based on [Lan96, Mes93a]. Let us give the specification of the protocol first and explain it afterwards:

```
module CP {
  import {
   protecting (NAT)
   protecting (LIST)
   protecting (MESSAGELIST)
  }

  signature {
  class Sender {
    rec      : ObjectId
    sendq    : List
    sendbuff : Elem
    sendcnt  : Nat
  }

  class Receiver {
    sender   : ObjectId
    incoming : List
    reccnt   : Nat
  }

  op empty                          : -> Elem

  op initialize sender _ with _     : ObjectId ObjectId -> Message
  op initialize receiver _ with _ : ObjectId ObjectId -> Message
  op to _ ack _  from _             : ObjectId Nat ObjectId -> Message
  op to _ (_,_) from _              : ObjectId Elem Nat ObjectId -> Message
  }

  axioms {
    vars S S' R R'   : ObjectId
    vars M N N'      : Nat
    var  MSG         : Message
    vars L ML L' ML' : List
    var  E           : Elem
    var  ATTS        : Attributes
```

```
rl [initialize-sender]:
   (initialize sender S with R)
   < S : Sender | rec = R', sendq = ML,
                  sendbuff = E', sendcnt = N', ATTS >
   =>  < S : Sender | rec = R, sendq = ML,
                      sendbuff = empty, sendcnt = zero, ATTS > .

rl [produce]:
   < S : Sender | sendq = E L, sendbuff = empty, sendcnt = N, ATTS >
   => < S : Sender | sendq = L, sendbuff = E, sendcnt = N + 1, ATTS > .

crl [send]:
   < S : Sender | rec = R, sendbuff = E, sendcnt = N, ATTS >
   => < S : Sender | rec = R, sendbuff = E, sendcnt = N, ATTS >
      (to R (E,N) from S)
   if E =/= empty:Elem .


rl [rec-ack]:
   (to S ack N from R)
    < S : Sender | rec = R, sendbuff = E, sendcnt = N, ATTS >
    => < S : Sender | rec = R, sendbuff = empty, sendcnt = N, ATTS > .

crl [rec-ack-not-suitable]:
    (to S ack N from R)
    < S : Sender | rec = R, sendcnt = M, ATTS >
    => < S : Sender | rec = R, sendcnt = M, ATTS >
    if N =/= M .


rl [initialize-receiver]:
   (initialize receiver R with S)
   < R : Receiver | sender = S', incoming = L', reccnt = N', ATTS >
    => < R : Receiver | sender = S, incoming = empty, reccnt = 0, ATTS > .


crl [receive-successfully]:
    (to R (E,N) from S)
    < R : Receiver | sender = S, incoming = L, reccnt = M, ATTS >
    => < R : Receiver | sender = S, incoming = L E, reccnt = M + 1, ATTS >
       (to S ack N from R)
    if N == M + 1 .
```

```
crl [receive-wrong-cnt]:
    (to R (E,N) from S)
    < R : Receiver | sender = S, reccnt = M, ATTS >
    => < R : Receiver | sender = S, reccnt = M, ATTS >
       (to S ack M from R)
    if N =/= M + 1 .


crl [loose-message]:
    MSG
    => acz-empty
    if elemoflist(MSG, conc((to S ack N from R),
                   conc((to R  (E,N) from S), empty))) .

crl [duplicate-message]:
    MSG => MSG MSG
    if elemoflist(MSG, conc((to S ack N from R),
                       conc((to R (E,N) from S), empty))) .
 }
}
```

The sender stores the information to be sent to a receiver in form of a list. Each of the elements of the list is sent in a package. A package consists of an element of the list, which is a part of the information to be exchanged, and a message counter.

The sender keeps on sending the same package until it receives an acknowledgment with the appropriate message counter. The receiver keeps on sending an acknowledgment for the last package it received and which carried the appropriate message counter (rule `receive-successfully`). The receiver counts the packages and sends with each acknowledgment the number of those packages. As soon as the sender gets, for the first time, an acknowledgment with a message counter equal to its message counter, it knows that the receiver has received the package it has been sending and begins to send a new package (with an incremented message counter).

Messages may be duplicated and destroyed and the medium cannot guarantee that the order of the packages sent is preserved. Two transition rules model loss and duplication of messages. The asynchronous message passing mechanism of Maude does not ensure that objects accept the messages in the order in which they were sent; thus, we need no transition rule to model that the order is not preserved.

Note that some of the transition rules model autonomous behavior of the objects. E.g., there is no message necessary for triggering a sender to prepare itself for sending a new package (rule `produce`) and to decide to send a package (rule `send`).

Now we have established the abstract and the concrete specification of the protocol. In the next three sections, we develop the refinement relation and the method for the refinement between the two Maude specifications.

5.2.2 First Step: Refinement of the Communication Pattern with Controlled Transitions

We refine a single abstract synchronous transition rule to a number of asynchronous transition rules. Our goal is to establish an (α, γ)-bisimulation between an abstract specification and a specification using asynchronous rules combined with the control mechanism we introduce. Hereby, we have to enrich the abstract specification with new message names and with additional attributes for the classes of objects involved and, probably, also with new parameters for existing messages, which store or carry the information necessary to synchronize the asynchronous transitions.

We name the specification which uses asynchronous rules plus the control structure Sp_C and the specification of the example `PEC`.

Control Algebra

Let us introduce the control mechanism used to steer the applications of transition rules. The control mechanism consists of a control algebra and an enrichment of the rules by so-called control terms. The control algebra provides us with the ability to compose transitions and to establish some sort of control flow.

Let us first introduce the control algebra. To gain control over the application of transition rules, we introduce control terms, which are part of each (controlled) transition rule. These control terms allow to determine the transition rule as well as its instantiation. As control terms we use the labels of the transition rules and enrich them by parameters. A pragmatic solution is to take at least the object identifier of the object, which is part of the transition rule, as parameter in the control term.

The operators of our control algebra are inspired by the operators of the π-calculus [Mil89, MPW92, FMQ96].

```
module CONTROL {
 import {
   protecting (ACZ-CONFIGURATION)
   protecting (RWL)
 }

 signature {
   [Ctrl < ACZ-Configuration]

   op _ ; _    : Ctrl Ctrl -> Ctrl
   op _ + _    : Ctrl Ctrl -> Ctrl
   op _ | _    : Ctrl Ctrl -> Ctrl
   op loop _   : Ctrl -> Ctrl

   op _ ; _    : Message Message -> Message
```

```
   op _ + _    : Message Message -> Message
   op _ | _    : Message Message -> Message
 }

  axioms {
    vars  l1 l2 l   : Ctrl
    var   L         : Ctrl
    vars  c d h c1 c2 d1 d2  : ACZ-Configuration

  crl [Seq]:      (l1 ; l2) c => d
                  if (l1 c ==> h) and (l2 h ==> d) .

  crl [Choice1]: (l1 + l2) c => d
                  if (l1 c ==> d) .

  crl [Choice2]: (l1 + l2) c => d
                  if (l2 c ==> d) .

  crl [Par]:      (l1 | l2) c1 c2 => d1 d2
                  if (l1 c1 ==> d1) and (l2 c2 ==> d2) .

  crl [Loop]:     (loop l) c => (loop l) d
                  if (l c ==> d) .

  crl [Loop1]:    (loop l) c => d
                  if (l ; (loop l)) c ==> d .
  }
}
```

We call a (composed) label, which is the label of a transition controlled by a control term l, a *run* of l. Typically, a control term has a set of runs.

Specification of Control Terms

The control algebra provides us with the syntactic constructs to specify control terms necessary to steer the objects involved in the refinement. We specify a number of control terms, one for each object. We compose them to a single control term for the refined part of the configuration via the parallel composition operator of specification `CONTROL`.

The enrichment of the signature and the enrichment by control terms induce a Galois connection (α, γ) and, provided the design of the refinement is well done, an (α, γ)-simulation relation between the abstract specification Sp_A and the controlled specification Sp_C. Note that we establish the relation between the initial state of the abstract state and an initial state of the concrete specification and the final state of the abstract and the final state of the concrete specification. Hereby we have to assume that the computation governed by the control term terminates. Typically this can be ensured by assuming reasonable fairness conditions.

Example. (Continued) The controlled asynchronous specification `PEC` contains control terms in each rule. Apart from this, it is identical to the concrete specification `CP`. The full specification can be found in App. C.2. To distinguish a control term from a label, we use `[[l]]` as control term for a rule with label `l`. Here, we give only the control terms and one example for an enriched transition rule:

```
module PEC {

 ...

  signature {
    class Sender {
       rec      : ObjectId
       sendq    : List
       sendbuff : Elem
       sendcnt  : Nat
      }

    class Receiver {
      sender   : ObjectId
      incoming : List
      reccnt   : Nat
    }

  op [[initialize sender _]]     : ObjectId -> Ctrl
  op [[produce _]]               : ObjectId -> Ctrl
  op [[send _]]                  : ObjectId -> Ctrl
  op [[rec-ack _]]               : ObjectId -> Ctrl
  op [[rec-ack not suitable _]]  : ObjectId -> Ctrl
  op [[initialize receiver _]]   : ObjectId -> Ctrl
  op [[receive successfully _]]  : ObjectId -> Ctrl
  op [[receive - wrong cnt _]]   : ObjectId -> Ctrl
  op [[loose message]]           : -> Ctrl
  op [[duplicate message]]       : -> Ctrl
  }

  axioms {

 ...
```

```
rl [initialize-sender]:
   [[initialize sender S]]
   (initialize sender S with R)
   < S : Sender | sendq = ML, ATTS >
   => < S : Sender | rec = R, sendq = ML,
                     sendbuff = empty, sendcnt = 0 > .

  ...
  }
}
```

We specify the control terms that steer the behavior of sender, receiver and medium in the specification.

```
module CONTROL-TERMS {
  import {
    protecting (PEC)
  }

  signature {
  op Ctrl-Sender        : ObjectId -> Ctrl
  op Ctrl-Receiver      : ObjectId -> Ctrl
  op Ctrl-LossDuplicate : -> Ctrl
  op ControlTerm        : ObjectId ObjectId -> Ctrl
  }

  axioms {
    vars S R : ObjectId


  eq Ctrl-Sender(S) =
      [[initialize sender S]] ;
       (loop ([[produce S]] ;
          (loop (( [[send S]]
                 + [[rec-ack not suitable S]])
                 + [[rec-ack S]])))) .

  eq Ctrl-Receiver(R) =
      [[initialize receiver R]] ;
       (loop (  [[receive successfully R]]
              + [[receive - wrong cnt R]])) .
```

```
  eq Ctrl-LossDuplicate =
      (loop [[loose message]])
    | (loop [[duplicate message]]) .


  eq ControlTerm(S,R) =
   ((   Ctrl-Sender(S)
      | Ctrl-Receiver(R))
      | Ctrl-LossDuplicate ) .
  }
}
```

A sender has to be initialized for the communication. Then, the sender stores the first information package (`produce`) and performs either a `send`, a `suitable-receive` or a `non-suitable-receive`, until it receives a suitable package. After receiving a suitable package, the transition rule `produce` models the autonomous state change of the sender, which begins with sending the next package.

Like a sender, a receiver is initialized first. Then, either the rule `receive-successfully` or the rule `receiver-wrong-cnt` is applied forever.

The medium may duplicate and lose messages in parallel. Thus, we control the medium by a parallel composition of two control terms: one controlling the loss, and one the duplication. Note that loss and duplication can occur in parallel. The control term for a configuration consisting of a sender, a receiver and a medium is the parallel composition of the control terms of the individual control terms.

The next step is to establish the mapping between abstract and controlled specification in form of a Galois connection. Let $\alpha : \wp(A_{\mathrm{AP}}) \rightarrow \wp(A_{\mathrm{PEC}})$ distribute over union of sets and $\alpha(\{c\ d\}) = \alpha(\{c\}) \uplus \alpha(\{d\})$, and $\alpha(\{\mathtt{t(p)}\}) = \mathtt{t'}(\alpha(\mathtt{p}))$

Specifically, for our specifications, we obtain:

$$\begin{aligned}
&\alpha(\mathtt{(send\ from\ S\ to\ R)}) = \\
&\qquad\quad \mathtt{Ctrl\text{-}Sender(S)} \\
&\qquad\ \mid \mathtt{Ctrl\text{-}Receiver(R)} \\
&\qquad\ \mid \mathtt{Ctrl\text{-}LossDuplicate(S,R)} \\
&\qquad \mathtt{(initialize\ sender\ S)(initialize\ receiver\ R)})
\end{aligned}$$

$$\begin{aligned}
&\alpha(\mathtt{<\ o\ :\ \ Sender\ |\ sendq\ :\ \ L\ >}) \\
&\quad = \{\mathtt{<\ o\ :\ \ Sender\ |\ rec\ =\ R,\ tosend\ =\ L,\ sendbuff\ =\ E,\ sendcnt\ =\ N\ >} \\
&\qquad \mid \mathtt{R}, \mathtt{E} \in \mathtt{ObjectId}, \mathtt{N} \in \mathtt{Nat}\}
\end{aligned}$$

$$\begin{aligned}
&\alpha(\mathtt{<\ o\ :\ \ Receiver\ |\ incoming\ =\ L\ >}) \\
&\quad = \{\mathtt{<\ o\ :\ \ Receiver\ |\ sender\ =\ S,\ incoming\ =\ L,\ reccnt\ =\ N\ >}) \\
&\qquad \mid \mathtt{S} \in \mathtt{ObjectId}, \mathtt{N} \in \mathtt{Nat}\}
\end{aligned}$$

$$\alpha(\mathtt{(send\ from\ S\ to\ R)}) = \{\mathtt{m} \mid \mathtt{m} \text{ is a run of } \mathtt{ControlTerm(S,R)}\}$$

Thus, finally, we obtain a simulation relation, $I(\mathtt{AP}) \precsim_{(\alpha,\gamma)} I(\mathtt{PEC})$, between the abstract specification AP and the controlled specification PEC.

◇

Note that, since we specify explicitly, which sequences of transitions we allow, it is sufficient to establish "only" a simulation relation here.

The price for introducing control objects is an increased danger of deadlock and a reduction of the performance at run time. Thus, despite of the fact that similar control structures are used in other languages, e.g., in the form of transaction objects [KLMW94, Kri97], we propose another refinement to a specification with asynchronous transition rules only.

5.2.3 Second Step: Behavioral Equivalence

Up to now, we have related one abstract transition to a fixed, controlled complex transition and we have considered as transitions equivalence classes modulo the equations on abstract data types. When abstracting from the control structures, the number of possible transitions is likely to increase. There may be new possible transitions that are unwanted because they change the set of final states and reduce the bisimulation relation between the now abstract and the now concrete transition system to a simulation relation. But there are also new transitions possible that are not harmful; these new transitions are behaviorally equivalent to transitions of the controlled specification.

More precisely, the best way to obtain a bisimulation relation between the (now) abstract controlled asynchronous and the (now) concrete asynchronous specification is to mirror the rewriting calculus in the axiomatizations of control terms considered to be behaviorally equivalent. The rewriting calculus is associative; thus, we need associativity (both of choice and of sequential composition). The rewriting calculus cannot ensure the order in which the rewriting rules are applied; thus, we need commutativity of sequential composition as well.

The axiomatizations of bisimulation and commutativity to define sets of equivalent transitions of controlled asynchronous transitions.

Our axiomatization is borrowed from the π-calculus [MPW92, FMQ96].

```
module CONTROL-ALGEBRA {
  import {
    extending (CONTROL)
  }
  axioms {
    vars  m n n1 n2 : Ctrl

  eq m + m = m .
  eq m + n = n + m .
```

```
  eq m + (n1 + n2) = (m + n1) + n2 .
  eq m | n = n | m .
  }
}
```

The rules given in the `CONTROL-ALGEBRA` identify bisimilar control structures. Thus, we get as an implementation of an abstract transition not a single, composed, controlled transition but the equivalence class of it.

Asynchronous message passing mechanisms cannot guarantee that the order in which messages are sent is the order in which they are accepted by objects. We would like to consider control terms to be behaviorally equivalent modulo commutativity, provided that the "new" sequences of transitions that are possible due to commutativity end only in final states, in which the "old" sequences of transitions end.

Definition 5.4 (Commutativity) [Len82, LH82] Let $Sp = (\Sigma, E, T)$ be a specification and let `CONTROL`, the control algebra as defined above, be a part of Sp.

Let $[[l_1]]$ m_1 c_1 `=>` d_1 and $[[l_2]]$ m_2 c_2 `=>` d_2 be two transition rules and m_1 and m_2 their labels.

Two actions m_1 and m_2 are *commutative* in a transition system (A, R) if $C \xrightarrow{m_1;m_2} D$, if and only if $C \xrightarrow{m_2;m_1} D$ for all configurations C, D in A.

Two transition rules $[[l_1]]$ and $[[l_2]]$ are *commutative* if they are commutative for all applications of rules $[[l_1]]$ and $[[l_2]]$.

We generalize the commutativity of simple actions to the commutativity of composed actions.

Definition 5.5 (Full commutativity) [Len82, LH82] Let $Sp = (\Sigma, E, T)$ be a specification and let `CONTROL`, the control algebra as defined above, be a part of Sp.

Let $M = m_1; m_2 \ldots m_m$ and $N = n_1; n_2 \ldots n_n$ be two composed messages and let L_1 M c_1 `=>` d_1 and L_2 N c_2 `=>` d_2 be two composed rules and M_1 and M_2 their labels.

If, for all $1 \leq i \leq m$ and $1 \leq i \leq n$, m_i and n_j are commutative then M and N are *fully commutative.*

If, for all applications σ_1 of L_1 and σ_2 of L_2, M and N are fully commutative then L_1 and L_2 are called *fully commutative.*

Full commutativity implies commutativity. Thus, full commutativity is an important criterion for finding commutative composed messages or transition rules.

This allows us to enrich the control algebra with a rule that allows to exploit the commutativity of rules:

```
eq a ; b = b ; a .
```

if `a` and `b` are fully commutative.

	[[initialize sender S]]	[[produce S]]	[[send S]]	[[rec-ack]]	[[rec-ack not suitable S]]	[[initialize receiver R]]	[[receive successfully R]]	[[receive - wrong cnt R]]	[[loose message]]	[[duplicate message]]
[[initialize sender S]]	-									
[[produce S]]	-	c								
[[send S]]	-	-	c							
[[rec-ack S]]	-	-	-	-						
[[rec-ack not suitable S]]	-	-	-	-	c					
[[initialize receiver R]]	c	c	c	c	c	-				
[[receive successfully R]]	c	c	c	c	c	-	-			
[[receive - wrong cnt R]]	c	c	c	c	c	-	-	c		
[[loose message]]	-	c	-	-	-	-	-	-	c	
[[duplicate message]]	-	c	-	-	-	-	-	-	-	c

Figure 5.2: Commutativity in `PEC`

Example. (Continued) Commutative pairs of transition rules of specification `PEC` are marked by an entry "c" in Fig. 5.2.

◊

We could end the refinement process at this level of specification: a transition rule employing the expensive synchronous communication pattern is refined to a transition rule employing the more cheaply and easily implementable asynchronous synchronization pattern. The inter-object synchronization that is necessary is partly ensured by additional information in the objects and messages and partly delegated to the control terms. The transaction concept of autonomous objects [Kri97] provides sophisticated techniques for correct and efficient inter-object synchronization.

Specifications with control or transaction objects are both abstract with respect to the underlying synchronization mechanisms and efficient. But, to make specification more efficient on the Rewrite Rule Machine [LMOMR94] and more realistic with respect to hardware, we would like to have specifications without control structures. Thus, we continue with our refinement process in the subsequent sections.

5.2.4 Third step: Changing the View

Up to now, we have mapped a single atomic transition to a set of of equivalent composed but still atomic transitions. In this step of our refinement, we change our view such that we observe the individual transitions in a composed transition and not the composed transition as a whole. Thus, we change the execution model of our specification. Technically, this is done by replacing the specification of the control algebra with a control algebra, which specifies a controlled transition relation in which the single transitions are observable.

```
module SCONTROL {
  import {
    extending (CONTROL)
  }

  axioms {
   vars l1 l2 l           : Ctrl
   vars c d h c1 c2 d1 d2 : ACZ-Configuration
   vars m m1 m2           : Message

  ceq (((l1 ; l2) c m1 m2 ==> l2 m2 h) and
        (l2 m2 h ==> d))  = true
        if (l1 c m1 ==> m2 h) and (l2 h m2 ==> d) .

  crl (l1 + l2) c => d
        if (l1 c ==> d) .
```

```
  crl (l1 + l2) c => d
        if (l2 c ==> d) .

  crl (l1 | l2) c1 c2 => d1 d2
        if  (l1 c1 ==> d1) and (l2 c2 ==> d2) .
  }
}
```

Let us compare the two specifications of the control algebra, `CONTROL` and `SCONTROL`, with respect to sequential composition. In specification `CONTROL`, only one transition with a composed label is observable, since we combine two transitions to a single transition. In specification `SCONTROL`, the control is still present, i.e., the order in which the transitions happen is fixed, but the two transitions are observable individually, i.e., there are two transitions with an "ordinary" message as label. E.g., in a specification containing `CONTROL`, we observe $[l1; l2]c \xrightarrow{a;b} d$ whereas, in a specification, in which `SCONTROL` implements the transitions, the relation is $[l1; l2]c \xrightarrow{a} h$ and $[l2]h \xrightarrow{b} d$.

Part of the additional synchronization between objects is specified with the enrichment of the signature, part of the synchronization is deferred to the control structures. The synchronization that is performed with the additional information stored in the enrichment of the signature remains when we abstract from the control structure. But we have to ensure the abstraction from the control structure does not contain transitions that are not equivalent to controlled transitions.

We relate specification $Sp_{\mathrm{Control}'}$, which we obtain from specification Sp_{Control}, by replacing the control algebra `CONTROL` by `SCONTROL`, with the concrete specification $\mathtt{Sp}_C$. As relation between the (now) abstract and the (now) concrete specification, we use an (α, γ)-bisimulation, parameterized by a Galois connection. Hereby, α just abstracts from the control term. If this already yields a bisimulation, we are finished, but more likely we have to strengthen the specification by rely conditions.

5.2.5 Fourth Step: Strengthening by Rely Conditions

Problems arise from inconsistent states: states in which only parts of the composed transitions are possible, or states that contain objects and messages that should appear together in a configuration. This yields a simulation, not a bisimulation relation between the controlled specification and the concrete specification. To exclude these inconsistent states and the transitions from and to such states, we introduce rely conditions and restrict the transition system to states which fulfill the rely conditions. We restrict ourselves to propositions on states as rely conditions.

Definition 5.6 (Rely condition) Let Sp be a specification and ϕ a predicate on states. ϕ is a rely condition for Sp if $Sp \models c \xrightarrow{l} d$ implies $Sp \models \phi(c) \Rightarrow \phi(d)$.

Thus, we require rely conditions to be compatible with outgoing transitions. But we do not require rely conditions to be compatible with multiset union of configurations, since we would like to use propositions that require the absence as well as the presence of messages and objects.

Example. (Continued) To enforce the bisimulation relation between the abstract and the concrete transition system, we need four rely conditions:

1. Sender and receiver have mutual knowledge of their object identifiers, i.e., there is either a matching pair of sender and receiver, or an appropriate initialization message around.

$$
\begin{array}{lll}
\phi_1(\mathtt{R},\mathtt{S})(c) =_{\mathtt{def}} & & (\texttt{< S : Sender | receiver = R >} \in c \\
& \Rightarrow & (\texttt{< R : Receiver | sender = S >} \in c \\
& \vee & (\texttt{initialize receiver R with S}) \in c)) \\
\vee & & (\texttt{< R : Receiver | sender = S >} \in c \\
& \Rightarrow & (\texttt{< S : Sender | receiver = R >} \in c \\
& \vee & (\texttt{initialize sender S with R}) \in c)) \\
\vee & & ((\texttt{initialize sender S with R}) \in c \\
& \wedge & (\texttt{initialize receiver R with S})) \in c
\end{array}
$$

2. There is only one matching pair of `initialize sender` and `initialize receiver` messages in the configuration and no `initialize sender` and `initialize receiver` messages are generated for senders or receivers involved in the communication:

$$
\begin{array}{l}
\phi_2(\mathtt{R},\mathtt{S})(c) =_{\mathtt{def}} \\
\quad ((\texttt{initialize sender S with R}) \in c \\
\vee \ (\texttt{initialize receiver R with S}) \in c) \\
\Rightarrow (\forall\ \texttt{S'},\texttt{R'} \in \texttt{ObjectId}, \texttt{S'} \neq \texttt{S}, \texttt{R'} \neq \texttt{R} : \\
\quad (\texttt{initialize sender S with R'}) \notin c \\
\quad \wedge (\texttt{initialize receiver R with S'}) \notin c)))
\end{array}
$$

3. There are neither send nor acknowledgment packages present in the configuration when the communication is initiated:

$$
\begin{array}{l}
\phi_3(\mathtt{R},\mathtt{S})(c) =_{\mathtt{def}} (\forall \texttt{E} \in \texttt{Elem}, \texttt{N} \in \texttt{Nat} : \\
\quad ((\texttt{initialize sender S with R}) \in c \Rightarrow (\texttt{to R: (E,N) from S}) \notin c) \\
\quad ((\texttt{initialize receiver R with S}) \in c \Rightarrow (\texttt{to S ack N from R}) \notin c))
\end{array}
$$

4. There is either a initialization message or a message used in the protocol present in the configuration.

$$
\begin{array}{l}
\phi_4(\mathtt{R},\mathtt{S})(c) =_{\mathtt{def}} \\
\quad ((\mathtt{initialize\ sender\ S\ with\ R}) \in c \\
\quad\ \Rightarrow (\forall \mathtt{N},\mathtt{R} : (\mathtt{to\ S\ ack\ N\ from\ R}) \notin c)) \\
\ \wedge ((\mathtt{initialize\ receiver\ R\ with\ S}) \in c \\
\quad\ \Rightarrow (\forall \mathtt{N},\mathtt{S},\mathtt{E} : (\mathtt{to\ R\ (E,N)\ from\ S}) \notin c)) \\
\ \wedge (\exists \mathtt{R} : (\mathtt{to\ S\ ack\ N\ from\ R}) \in c \\
\quad\ \Rightarrow ((\mathtt{initialize\ sender\ S\ with\ R}) \notin c) \\
\ \wedge (\exists \mathtt{N},\mathtt{E} : (\mathtt{to\ R\ (E,N)\ from\ S}) \in c \\
\quad\ \Rightarrow (\mathtt{initialize\ receiver\ R\ with\ S}) \notin c))
\end{array}
$$

♢

Let us discuss the role of these rely-conditions, e.g. of condition ϕ_2. Our specification has no mechanisms to ensure that sender or receiver are not initialized, while the transmission of the data is in progress. Thus we have to exclude that (1) more than one pair of initialization messages is present in the configuration (see ϕ_2) and (2) that an initialization message and a message used in the protocol are not present in the configuration.

The second relation of the refinement relation in our example turns out to be a bisimulation and, thus, we have succeeded in the refinement. What would have gone wrong if the refinement relation between the abstract and the concrete specification turned out to be "only" a simulation relation? Most likely, the synchronization in the concrete specification would have been too weak. If the specification is not well-designed, the rely conditions become too strong, excluding unrealistically many states from the concrete specification. In this case, the design of the refinement has to be redone.

5.2.6 Refinement of Large Specifications

The previous sections deal with a method of refining a synchronous communication to several asynchronous communications. But, in fact, we have only considered the refinement of a single transition rule to a (small) number of transition rules. What happens if this single abstract transition rule is part of a possibly complex specification? We would like such a refinement of a single communication to be as local as possible in the specification code and definitely not to affect large parts of a complex specification.

Recall that we think of the specification process as consisting of two parts. The first part is to set up the basic data types and the class hierarchy and to specify the behavior of the classes by transition rules. The second part is to model a system with the classes and the rules specified in the first part. The desired property of the refinement, "locality", applies to both parts of the design process.

In the first phase of the design process, the refinement of a transition rule could necessitate to adapt the rules in which messages and classes being used in the refined transition occur. In the second phase, it is important that the semantics of the system is not changed by the refinement.

The next sections present several suggestions of how to keep the changes induced by a refinement as "local" as possible. In Sect. 5.2.6, we use a combination of equations and states as classes in the refinement of a single transition rule. These techniques are generalized in Sect. 5.2.6 by using also multiple inheritance to deal with multiple applications of the same refinement in one specification. We demonstrate the refinement with the protocol example.

Embedding the Refinement of a Rule into a Specification

When refining a single transition rule employing a synchronous communication pattern to asynchronous transition rules, new messages have to be introduced. In particular, this might include the message(s) triggering the communication and the messages produced by the refined transitions. We use equations for the transformations of the messages and, thus, for the whole configurations.

To tie the refined concrete specification to a large abstract specification, we need in our example only one equation, namely,

```
(send from S to R) =  (initialize sender S with R)
                      (initialize receiver R with S)
```

Using equations has the advantage that the transformation from abstract to concrete configurations can be done with a strong bisimulation relation (see also the discussion in Sect. 2.3.6). A precondition for the use of such equations is that the message names introduced in the refinement do not exist already in the abstract large specification.

It remains to settle the difference between the classes as they are defined and used in the abstract and the concrete specification. To distinguish the abstract and concrete classes here, we name the abstract classes `ASender` and `AReceiver` and the concrete classes `CSender` and `CReceiver`.

Basically, we have two different implementations of classes in the large, refined specification and we have to make the switch between these implementations as smooth as possible. To that end, we use the states-as-classes approach introduced in Sect. 2.2.1. An object belongs to the "abstract" class and changes to the "concrete" class for the duration of the refinement.

To make this class change consistent, the class hierarchy has to be extended: a superclass containing both an abstract and a concrete class is introduced. In our example, this new superclass is `SuperSender`, replacing `ASender` in the class hierarchy. `CSender` and `ASender` are declared to be subclasses of `SuperSender`; analogously for the receiver. This change of the class hierarchy is depicted in Fig. 5.3.

Some of the transition rules of the refinement have to be adapted to the new class hierarchy. We have to have rules that make the transition between the abstract and the concrete classes. Typically, rules that specify transitions "into" and "out of" the refined transition system have to be changed.

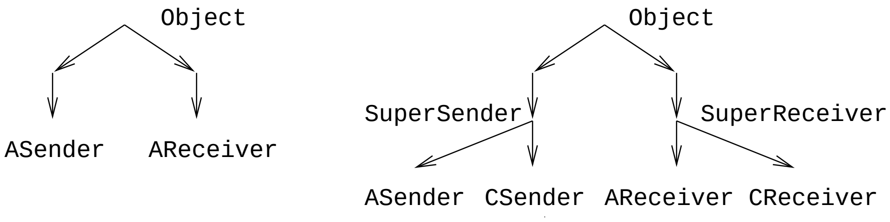

Figure 5.3: Refinement by states as classes

In our example, two rules of specification CP have to be changed:

```
[initialize sender]
rl (initialize sender S)
   < S : ASender | sendq = ML >
   =>  < S : CSender | rec = R, sendq = ML,
                       sendbuff = empty, sendcnt = N + 1 > .

[initialize receiver]
rl  (initialize receiver R)
   < R : AReceiver >
  =>  < R : CReceiver | sender = S, incoming = empty, reccnt = 0 > .
```

Transition rules that reverse this class change at the end of the communication have to be introduced. We refrain from giving them here.

Note that the states-as-classes concept and the class hierarchy we have introduced here also provide a good shield against the unrefined part of the specification from interfering with the refined part of the specification, since all other abstract rules deal only with the abstract classes, not with the concrete classes. Only rules inherited from superclasses of SuperSender might cause problems in a refinement, since they might interfere with the refinement and, thus, we have to establish independence or non-interference results.

There are several advantages of this way of embedding a refinement into a large specification: (1) the unrefined transition rules of the abstract specification remain unchanged, (2) several refinements of transition rules can coexist and can be embedded into the same abstract specification independently, (3) the refinement process can be iterated and (4) the states-as-classes approach is not only a very smooth way of modeling, but also a very efficient way of implementing state changes of objects.

Several Refinements with the Same Concrete Specification

Let us assume another, a little bit different setup in the refinement. Assume we have several transition rules in the abstract large specification to which we want to apply the same refinement. We could apply the method developed in the previous section and refine each transition rule according to the schema. This solution has one drawback. For each of the refinements, the refining specification would have to be part of the resulting specification. This would inflate considerably the number of transition rules of the resulting specification.

Thus, we modify the approach of the previous section and use multiple inheritance. Let us explain how we proceed with the example of the communication protocol. The new class hierarchy is depicted in Fig. 5.4. The dotted lines indicate inheritance relations that are newly introduced.

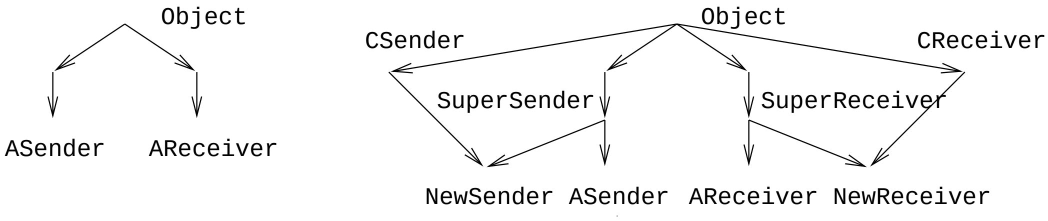

Figure 5.4: Refinement by states as classes and multiple inheritance

Let us explain how we extend the abstract specification with the example of the sender and receiver from the protocol specification. This construction has to be done for all classes for which the communication is refined.

The classes `CSender` and `CReceiver` are added to the abstract specification. The abstract specification contains the classes `ASender` and `AReceiver`. We replace them in the class hierarchy by `SuperSender` and `SuperReceiver`. `ASender` and `AReceiver` are declared to be subclasses of `SuperSender` and `SuperReceiver`. New classes `NewSender` and `NewReceiver` are added. They inherit from `SuperSender` and `CSender`, respectively, from `SuperReceiver` and `CReceiver`. In this approach, the attributes declared for class `ASender` and `AReceiver` are now declared in `SuperSender` and `SuperReceiver`, such that both of their subclasses inherit them.

The figure above looks complicated and, in particular, more complicated than Fig. 5.3, in which the changes necessary for a refinement of a single rule are depicted. The advantages of this modification of a class hierarchy become apparent only after a look at a diagram in which one specification is used to refine several transition rules, i.e., where a refinement like the one described and depicted above is done several times. In Fig. 5.5 the inheritance relations between the refining specification and the classes inheriting from the refined specification are depicted by dashed lines.

Again, the transition rules implementing transitions into the refinement and out of the refinement have to be changed. This time, the information to be exchanged has to be switched from the attributes in which it is stored in class `ASender` to attribute `sendq`, in which class `Sender` expects it to be. None of the transition rules of the abstract specification have to be altered in order to perform the refinements.

This approach precludes that the attribute in which all senders store the information to be sent and the attribute in which all receivers of different classes expect the information have the same names as the attributes in the abstract specification of the protocol.

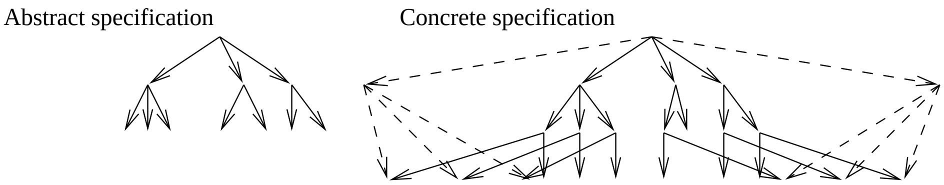

Figure 5.5: Multiple refinement with the same refining specification

We "reuse" a refinement several times to refine a complex abstract specification in a very modular way. An application of such a refinement method would be a library of hardware implementations like, e.g., skeletons [Col89, LGH97, Ski94] to refine a single abstract specification to several different implementations according to the efficiency needs of different hardware platforms.

5.3 Behavioral Refinement of Specifications

In the previous two sections of this chapter, we have dealt with refinement of the implementation of a class, and the refinement of a communication protocol. Both applications of refinement are based on bisimulation, more precisely, on (α, γ)-bisimulation. This refinement is very fine and local, i.e., the behavior of a system is not changed by the refinement of the implementation of a single class or the refinement of the communication protocol. The requirements on a refinement relation for a specification differ from the requirements on a refinement relation for classes or small parts of specifications. The properties that have to be preserved in the refinement determine the refinement relation. Abstraction is important, in order to allow some freedom in the implementation and to cope with the complexity of such systems.

In this section, we develop several notions of refinement which are tailored to Maude and, in particular, to the properties that Maude specifications express.

Let us describe our setting more closely. An object-oriented abstract Maude specification is refined stepwise to a concrete Maude specification. Such a specification comprises the sort and class hierarchy, the equations and the transition rules. In a refinement step, parts of the class hierarchy have to be redesigned and the transition rules and equations have to be adapted accordingly. Moreover, we allow action refinement, i.e., a single transition is refined to a number of concrete transitions. In the process of refinement, an abstract Maude specification becomes more and more concrete, i.e., it ends up employing only features which are present in object-oriented programming languages.

In Chap. 3, we have dealt with the correspondence between algebraic and coalgebraic specifications and between the two specification languages, the μ-calculus and Maude. Thus, we refrain in this section from using different specification languages and use as both abstract and concrete specification language Maude, because it makes reasoning about the states and behavior simpler.

5.3.1 Refinement Relation

The property that a transition is possible in a certain state is the property that is expressed by transition rules. Thus, we would like this property to be preserved by the refinement relation.

Since we do not care about the state transitions and how they are computed, we use in this section an unlabeled transition system and allow a transitivity rule to deduce transitions, i.e., $c \to d$ and $d \to e$ implies $c \to e$.

Our formal basis is the loose approach to refinement, i.e., the semantics of a specification is the class of all computation structures in which the equations and transition rules hold. Refinement is characterized by implication: all equations and state transitions derivable in the abstract specification have to be derivable in the concrete specification as well.

Let us give our definition of refinement.

Definition 5.7 (Refinement) A specification Sp is *refined* by a specification Sp', written $Sp \rightsquigarrow Sp'$, iff

$$\begin{array}{l} \quad \Sigma \subseteq \Sigma' \\ \wedge\ (\forall t_1, t_2 \in T(\Sigma) : Sp \models t_1 = t_2 \ \Rightarrow\ Sp' \models t_1 = t_2) \\ \wedge\ (\forall c, c' \in T_{\mathrm{Cf}}(\Sigma) : R \models c \to c' \ \Rightarrow\ R' \models c \to c') \end{array}$$

The step from synchrony to asynchrony involves a refinement of the object model, the communication and the order in which the objects reach their common final configuration. Since we use the loose approach, we are allowed to refine an abstract state transition, specified by a single application of some transition rule, to a concrete state transition, specified by multiple applications of transition rules—as long as every abstract state transition is represented by a (sequence of) concrete state transitions.

In the process of refinement, we have to allow a certain degree of abstraction. Typically, one is only interested in some values stored in the states of objects that are part of a global configuration. We determine which objects, which parts of their states, and which messages we would like to observe. We call our observation the observable part. It comprises the primitive sorts, classes and messages.

Definition 5.8 (Observable part) The *observable part Obs* of a specification Sp with $\Sigma = (S, C, \leq, F, M)$ is a triple (S', C', M') such that $S' \subseteq S$, $M' \subseteq M$ and $(\texttt{c'|atts'}) \in C' \ \Rightarrow\ (\exists\, \texttt{atts} : ((\texttt{c'|atts}) \in C \ \wedge\ \texttt{atts'} \subseteq \texttt{atts}))$.

We presume that all sorts of observable attributes and all class identifiers are observable and that the domain of functions with observable range is observable as well.

Let `c` $\{$ `a1 : s1` $\ldots$ `an : sn` $\} \in C'$ and `m : s1` $\ldots$ `sn -> Message` $\in M'$. Then we write `< o : c | a1 = v1,` $\ldots$ `,an = vn >` $\in C'$, where `o` is an object of class `c` and we write $(\texttt{m(p1, ... ,pn)}) \in M'$, where `m` is a message with message name `m` and `m : s1 ... sn -> Message` $\in M'$.

Note that subclasses do not inherit observability properties from their superclasses: the attributes that we observe in a class are not necessarily observed in all subclasses or all superclasses.

Note that the configurations consists of both observable and unobservable messages and objects. In order to be able to observe part of a configuration, we define an observation function obs_R on the configurations of specification R that projects the observable part $Obs = (S', C', M')$. We call the images of the observation function *observations*. Let us state the definition and then explain it.

Definition 5.9 (Observation function) The observation function obs_R of a specification R for an observable part Obs is defined by:

$$\mathrm{obs}_R : \texttt{Configuration} \rightarrow \texttt{Set of Configuration}$$

$$\mathrm{obs}_R(\epsilon) = \{\epsilon\}$$

$$\mathrm{obs}_R(c_1\ c_2) = \mathrm{obs}_R(c_1) \uplus \mathrm{obs}_R(c_2)$$

$$\mathrm{obs}_R(\texttt{< o : c | atts >}) = \begin{cases} \{\texttt{< o : c' | atts' >} \mid \\ \qquad \texttt{< o : c' | atts' >} \in C' \\ \qquad \wedge\ \texttt{c} \leq \texttt{c'}\} \\ \quad \text{if } (\exists\ \texttt{c'} \in C' : \texttt{c} \leq \texttt{c'}) \\ \{\epsilon\} \quad \text{otherwise} \end{cases}$$

$$\mathrm{obs}_R(\texttt{(m(p))}) = \begin{cases} \{\texttt{(m(p))}\} & \text{if } \texttt{m} \in M' \\ \{\epsilon\} & \text{if } \texttt{m} \notin M' \end{cases}$$

Each observation of a configuration is a set of observable parts. Each object may be observable in a number of superclasses, and its observations may differ depending on which superclass observes it. Thus, the range of the observation function is a set of configurations. Remember, that a configuration is itself a multiset of messages and objects.

The observations in the abstract specification form a subset of the observations in the concrete specification, because a refinement into new subclasses may generate new subtype dependences between observable classes. This may lead to added observations (see our example in Sect. 5.3.4).

Our concept of an observable part refines the concept of an observable part defined for algebraic specifications in [BH93, BHW95, ONS93, Wir90] and the concept of a visible action in process algebras in [Mil89]. Our extension covers the addition of attributes and messages and the refinement of the class hierarchy and of communication protocols.

5.3.2 Behavioral Transition Refinement

In the following, we propose two concepts of refinement for relating the behavior of an abstract specification to a concrete specification: behavioral transition refinement and behavioral simulation refinement. The refinement of the static part of the language is the same for both relations. See Sect. 5.3.4 for a discussion and a comparison of the two refinement relations.

Definition 5.10 (Behavioral transition refinement) A specification Sp' is a *behavioral transition refinement* of a specification Sp with respect to an observable part Obs, written $Sp \rightsquigarrow^{\mathrm{tr}}_{Obs} Sp'$, iff

$$
\begin{aligned}
& \Sigma \subseteq \Sigma' && (1)\\
\wedge\ & (\forall s \in Obs,\ t_1, t_2 \in T_s(\Sigma) : Sp \vdash t_1 = t_2 \Rightarrow\ Sp' \vdash t_1 = t_2) && (2)\\
\wedge\ & (\forall\ c, c' \in T_{\mathrm{Cf}}(\Sigma) : && (3)\\
& \quad (\exists\ d, d' \in T_{\mathrm{Cf}}(\Sigma') : \mathrm{obs}_{Sp}(c) \subseteq \mathrm{obs}_{Sp'}(d)\ \wedge\ \mathrm{obs}_{Sp}(c') \subseteq \mathrm{obs}_{Sp'}(d')\ \wedge \\
& \quad\ (Sp \vdash c \rightarrow c'\ \Rightarrow\ Sp' \vdash d \rightarrow d')))
\end{aligned}
$$

We call this relation transition refinement because it relates a transition in the abstract specification to a transition in the concrete specification. This relation takes neither traces, i.e., sequences of transitions nor choices into account. Thus, one configuration can be refined into several configurations that are not connected appropriately by transitions: each concrete configuration may have fewer successor configurations than the abstract one, or transitions between the concrete configurations may be missing. This relation is, in some sense, the weakest possible relation between two specifications. In process theory, a relation that ensures that the behavior of the concrete specification has at least the same traces and the same choices is called a simulation. We have a similar refinement relation here.

5.3.3 Behavioral Simulation Refinement

The notion of a behavioral transition refinement is quite coarse because it does not require that sequences of transitions or choices are preserved in the refinement. We adapt the notion of simulation relations to a behavioral simulation relation, which preserves sequences of transitions and choices.

Definition 5.11 (Behavioral simulation refinement) A specification Sp' is a *behavioral simulation refinement* of a specification Sp with respect to an observable part Obs, written $Sp \rightsquigarrow^{\precsim}_{Obs} Sp'$, iff (1) and (2) of Def. 5.10 hold and, in addition,

$$
\begin{aligned}
& (\exists\ \precsim\ \subseteq T_{\mathrm{Cf}}(\Sigma) \times T_{\mathrm{Cf}}(\Sigma') : (c \precsim d\ \Rightarrow && (3)\\
& \quad (\forall\ c' \in T_{\mathrm{Cf}}(\Sigma) : Sp \vdash c \rightarrow c' \Rightarrow (\exists\ d' \in T_{\mathrm{Cf}}(\Sigma') : Sp' \vdash d \rightarrow d' \wedge c' \precsim d'))) \\
& \quad \wedge (\forall\ c \in T_{\mathrm{Cf}}(\Sigma) : (\exists\ d \in T_{\mathrm{Cf}}(\Sigma') : c \precsim d)) \\
& \quad \wedge (c \precsim d\ \Rightarrow\ \mathrm{obs}_{Sp}(c) \subseteq \mathrm{obs}_{Sp'}(d)))
\end{aligned}
$$

The behavioral transition relation allows to abstract from parts of the configuration since, for the equality of states, only the observable part of the configuration is taken into account. This behavioral transition relation allows to alter the implementation of the objects and allows action refinement, since each abstract transition can be matched by a sequence of transitions. The property that is preserved by this behavioral transition relation is that a transition between observably equal states is possible.

5.3.4 Behavioral Transition vs. Behavioral Simulation Refinement

Let us illustrate that both notions of behavioral refinement presented in the previous sections—transition and simulation refinement—have reasonable applications. We expect the concrete specification to be at least as powerful as the abstract specification, i.e., to offer all choices and all transitions that exist at the abstract level. From this point of view, behavioral simulation refinement is the better choice.

Here is an example for a behavioral transition refinement, which demonstrates that simulation refinement can be too strong and that behavioral transition refinement can be a good choice as well. Assume that, in the process of refinement, we would like to follow the guidelines developed in Sect. 2.4 and introduce additional classes to reduce the complexity of the state of objects and increase the safety by introducing new states.

We omit the Maude code; we illustrate the state transitions graphically in Fig. 5.6 instead.

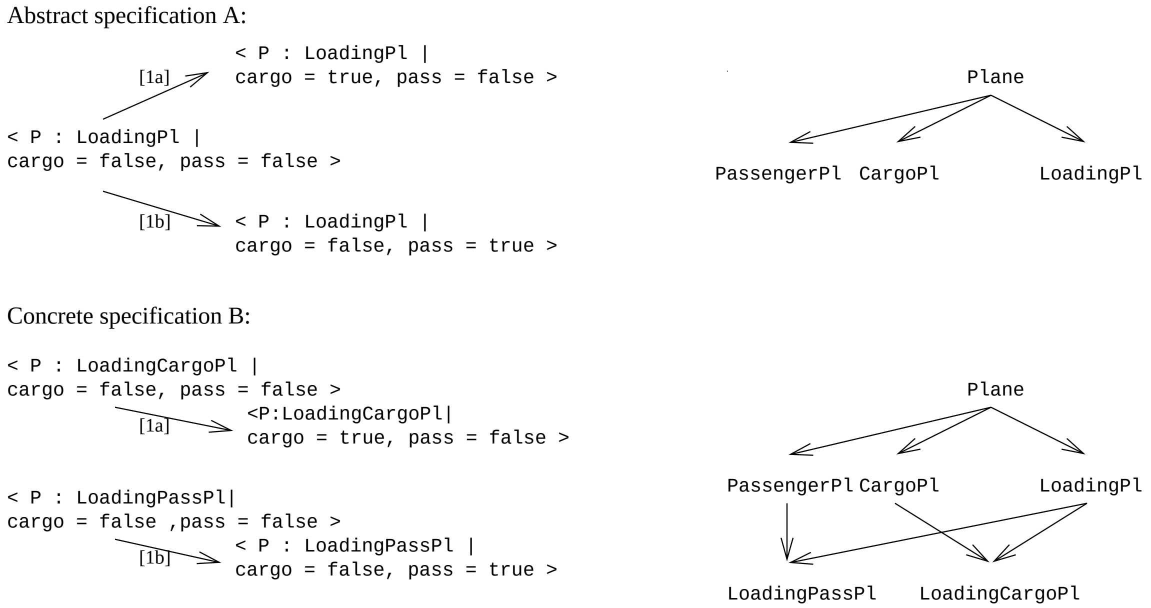

Figure 5.6: State transitions and class hierarchy of the abstract and the concrete specification

Let us consider the state Loading. This state differs for passenger and cargo planes. A passenger plane loads passengers, a cargo plane cargo. All loading planes have the additional attributes `cargo` and `pass`. An attribute is `true` if the corresponding entity has been loaded and `false` otherwise. The process of loading is modeled by two different messages, `(to P load cargo)` and `(to P load passengers)`. These messages are sent from the ground control to the plane to be loaded.

In the abstract specification, class `LoadingPl`, represents a plane that is ready to load cargo or passengers. Both cargo and passenger planes accept both requests, although one of them contradicts the value of attribute `PlaneType`.

Let us refine this specification by introducing two subclasses, `LoadingCargoPl` and `LoadingPassPl`. Both subclasses accept only the message compatible with the plane's type. This refinement increases safety because the plane can reject messages of the ground control that are not compatible with its type.

Of our specification, we would like to observe class `LoadingPl` with its two attributes and the two messages. We would also like to make some observations of class `Plane` and classes `PassengerPl` and `CargoPl`. The observational part of the specifications consists of:

```
S' = "all primitive sorts"
C' = { class Plane { flightNo : Nat, destination : ObjectId }
       class PassengerPl { #Passengers : Nat }
       class CargoPl { weight : Nat }
       class LoadingPl { cargo : Bool, pass : Bool }
     }
M' = { to _ load cargo : ObjectId -> Message
       to _ load passengers : ObjectId -> Message
     }
```

Let us consider the following configuration D (the variable `ATTS` subsumes attributes of `P` not needed in this example):

```
D = < P : LoadingPl | cargo = false, pass = false,
                      planetype = Passenger, destination = D,
                      flightNo = N, #passengers = 100, ATTS >
    (to P load passengers)
```

In the abstract specification, the plane is observable in two classes, `LoadingPl` and `Plane`. Our observation is a set of two configurations that are different projections of D:

```
obs_A(D) =
 {< P : LoadingPl | cargo = false, pass = false >
  (to P load passengers),
  < P : Plane | flightNo = N, destination = D >
  (to P load passengers)}
```

In the concrete specification, the smallest sort of plane `P` is `LoadingPassPl` (where the attribute `cargo` is `false`). Configuration D', the refinement of configuration D which we observe in the concrete specification, differs from configuration D in one point: the class the plane belongs to is `LoadingPassPl`, not `LoadingPl`. According to Def. 5.9, the plane is observable in one more class, `PassengerPl`. Thus, our observation of configuration D' is:

$\mathrm{obs}_B(D') =$
 $\mathrm{obs}_A(D) \cup$
 $\{$ `< P : PassengerPl | #passengers = 100 >`
  `(to P load passengers)` $\}$

The refinement is a behavioral transition refinement. Every state transition in the abstract specification is implemented by a corresponding state transition in the concrete specification. In Fig. 5.6, corresponding state transitions are labeled identically.

The refinement is not a behavioral simulation refinement. In the abstract specification, a plane in state Loading, with both attributes `false` (as in configuration D), can accept two different messages. In the concrete specification, there are two different states in each of which only one message can be accepted.

5.4 Related Work

For algebraic specifications and concurrent systems, different refinement notions have been developed. Since our approach to refinement comprises notions and techniques from algebraic specifications, as well as concurrent specifications we briefly review related work in object orientation but also in algebraic specifications, concurrent languages and object-oriented languages.

Algebraic specifications In our work, we use the loose approach to refinement [Wir90] and implication as refinement relation. Behavioral refinement allows to abstract from aspects of the specification. Hereby, the observations and the degrees of abstraction vary. In [Hen89, ST85], the context "observes" the elements and elements are behavioral equal if they cannot be distinguished by the contexts. This concept is generalized in [BHW95, HS95, HS96]. [ONS93] gives an excellent survey of the properties of initial and loose approaches to behavioral refinement.

Refinement in rewriting logic Several papers of Meseguer et al. [Mes90b, Mes93a, MW92] point out that synchronous and asynchronous specifications represent different levels of abstraction. In [LMOMR94], an asynchronous implementation of a synchronous specification of a spreadsheet is presented and the related issues of the transformation from synchrony to asynchrony, and of deadlock and fairness are discussed. Meseguer refers to a categorical semantics of Maude and relates the rewriting logic to other logical frameworks

[MMO95]. In [Mes90b], a rewrite relation similar to our configuration transition relation is defined and a mechanism for an implementation relation between rewrite theories expressed in terms of their associated (2-)Lawvere categories is presented.

Refinement of concurrent systems Action refinement is one kind of refinement which is particular to concurrent specifications. In action refinement, an abstract, atomic action is replaced by number of concrete actions. The representation of a concurrent system may be a process algebra, an event structure or a Petri net [vG90b]. The effects on the properties preserved by such a refinement in various abstract equivalences, e.g., trace, readiness, failure and bisimulation, are studied in [vG90a]. Their results suggest not to use action refinement when reasoning about property-preserving refinements of object-oriented systems.

The simulation relation is the basis for a refinement preorder for processes represented by modal transition systems in [LT91, Lar93]. Two different transition systems, one using the box, the other the diamond operator, specify a concurrent system. The relations between the transition systems preserve the properties expressed by one or both specifications.

Refinement in object-oriented specifications In [DE95, Den96a, EDS93, FM94, SJE92], object-oriented specifications of TROLL are refined by an action refinement: a single abstract action is refined to an event structure, which models the dependence of the actions at the lower level of abstractions. The goal of this refinement is to preserve the dependencies between actions. It preserves also formulas in a weak temporal logic, developed for reasoning about object-oriented systems.

A refinement with an informal specification at the most abstract level is given in [WK96]. The object-oriented analysis and design method of Jacobson [JCJÖ92] is enriched by formal assertions, such that a Maude specification can be generated automatically from the abstract specification. This Maude specification is refined stepwise to a Java program. The refinement relation used for the refinement is a bisimulation based on an abstraction function of states.

In [Jon93a] refinement techniques for $\pi o \beta \lambda$ specifications are presented. The goal of these techniques is to infer more parallelism in the process of refinement. A sequential abstract object-oriented specification is refined to a concrete parallel program. This refinement changes neither the final state of the objects nor the results that the computations yield. Several rules determine when this refinement can be applied. The correctness proofs for this kind of refinement are based on bisimulation relations and confluence criteria [NS96], similarly to our refinement of a communication protocol. However, since the criteria under which the refinement can be applied are rather strict and since the amount of $\pi o \beta \lambda$ concurrency is limited, the scope of the refinement relation and the amount of concurrency it can exploit is rather restricted.

5.5 Remarks and Discussion

In this chapter, we have presented refinement techniques appropriate for concurrent object-oriented languages. We distinguish refinement of a single class, refinement of a communication protocol, and refinement of collections of objects, all of which we have demonstrated with the airport example. For these kinds of refinements, we have developed different techniques. Verification and refinement are dual and, thus, the refinement notions "reuse" the formal framework used for verification in Chap. 4. The refinement of a class uses bisimulation as the refinement relation to make sure that the behavior of a collection of objects is not changed by the refinement relation. Our refinement of the communication protocol uses a pair of bisimulation relations. Only in the refinement of a collection of objects we allow more freedom and provide a coarse and a fine refinement relation.

Generally, the methods of refinement are specific to the problem domain. Although we present different typical situations in refinement, we cannot claim to provide a refinement method. Maude is a general-purpose language and, thus, a single refinement technique and method cannot cover the whole spectrum of Maude specifications.

Important for the refinement is that the effects of the refinement of a single class and the refinement of a communication protocol remain local. We present suggestions as to how the refinement of a small subset of classes in a class hierarchy can be done with minimal changes in the class hierarchy. Here, we use the programming method and the reuse concepts developed in Sect. 2.2. Naturally, concepts for integration a local refinement into a specification are not needed for the refinement of a class and the refinement of a specification.

It is crucial that the refinement steps can be iterated. In a refinement from an abstract Maude specification to a concrete notation, several phases are necessary according to the principle of stepwise refinement. In our opinion, in each of these phases, all three kinds of refinement are necessary.

The formal background of the refinement relation is the loose approach to refinement, which has been developed for algebraic specifications. Since not only the refinement of properties and verification, but also the duality of algebraic and coalgebraic specifications as well as the different inheritance relations are based on the formal background of bisimulation relations, we can ensure the orthogonality and compositionality of our approach.

From a methodological point of view, object orientation has advantages in formal refinement. The implementation and refinement of objects is rather independent from the implementation and refinement of the communication protocols. Thus, different customized techniques can be applied in the refinement of classes and protocols. Moreover, the classes themselves can be refined independently from each other. This helps to make the refinement of complex systems feasible. The fact that these different techniques of refinement fit together is due to our uniform formal framework, and the fact that they can be applied independently is due to the paradigm of object orientation.

Chapter 6

Conclusions

In this thesis, we have presented a new approach to object-oriented specification of distributed systems. Our work comprises the kernel of an object-oriented concurrent programming paradigm: a specification language, verification and refinement techniques, and concepts for design and reuse. The language Maude [Mes93a, MW92], which has been developed by J. Meseguer, has been seminal for our work. Maude provides constructs for specifying the behavior of objects—in particular, communication and synchronization—at a very abstract level. We have designed a framework for verification and refinement techniques, concepts for design and reuse method for the specification language Maude.

Crucial to our approach was the way in which we have merged the object-oriented and the concurrent programming paradigm. We have decided to employ serial objects as entities in a concurrent setting. Objects determine autonomously whether they will accept a message or not. Thus, we have obtained two different levels of specification: the sequential intra-object and the concurrent inter-object level. At each of these two levels, we have different properties that we specify and for which we have developed different formal techniques of verification and refinement.

Particular to our approach of specification and our techniques of refinement and verification is that both levels, the specification of data at the intra-object level and the specification of behavior with the communication and synchronization at the inter-object level, are equally flexible and important. The abstractness provided by Maude sets our approach apart, since we did not have to deal with typical object-oriented constructs that have proven to be cumbersome in other, more traditional approaches and with details in the way synchronization and communication is organized that have proven to be error-prone.

Let us describe the formal background and our formal techniques more closely. The *formal background* of our work consists of traditional concepts and techniques from algebraic specification as, e.g., stepwise refinement. Thus, we classify our approach as an algebraic approach to object-oriented specification of distributed systems. However, we made a detour into the world of concurrency. From there, we employed the μ-calculus [Koz83] as a property-oriented specification language to describe the behavior and the technique of abstract interpretation [CC78, LGS$^+$95]. We employed the basic concepts of simulation, bisimulation and the properties that these relations preserve for implementing the principle of encapsulation, for the refinement at the intra- and inter-object level, and abstraction and verification. These concepts match well with Maude, as our results on the duality of Maude specifications and properties and the results on inheritability of classes of properties have shown. Despite the different levels of specification, we succeeded in designing a uniform and homogeneous formal background for the specification language and our formal techniques.

In our *verification* techniques we have focused on the inter-object level. Particular to Maude are, at the inter-object level, the flexible synchronization and communication patterns. However, the specification style is operational and this induces the necessity of verification of properties of Maude specifications. The novelty of Maude's concepts of communication and synchronization and the complexity of verification at the inter-object level made it necessary to use techniques for abstraction and verification which are new in object orientation. We have demonstrated the verification techniques, at the intra-object level, on the example of properties of a bounded buffer. At the inter-object level, we have used the verification of a mutual exclusion property of a specification of an airport as our example.

The *refinement* from an object-oriented specification to a program is also different for the two levels. At the intra-object level, the implementation of the objects is refined from a specification to a program such that the behavior observable from the outside remains unchanged. At the inter-object level, we have introduced techniques for refining the communication and synchronization mechanisms. In particular, we have refined the abstract implicit synchronous communication to concrete explicit asynchronous communication. We have introduced several different relations between specifications (and programs) at various levels of abstraction; the kind of relation and the degree of abstraction determine the properties that are preserved in the refinement. At the intra-object level, we have refined the specification of a bounded buffer. At the inter-object level, we have refined the implementation of a communication protocol and we have demonstrated the refinement relations which allow abstraction on the specification of an airport.

We have covered in our work two aspects of software engineering: concepts for design and reuse. We have developed a *set of reuse constructs* for reuse of Maude specifications. This set of reuse constructs is powerful enough to circumvent the inheritance anomaly. We have demonstrated the power of our reuse concepts with the classical example of the bounded buffer and its extensions [MY93]. We have used the verification and refinement techniques with their relations between object-oriented specifications, at various levels of abstraction, to characterize the classes of properties that they inherit. This duality of reuse constructs and inheritance of classes of properties is unique, and it proves that Maude, the properties that Maude specifications describe and the reuse constructs match well.

Based on Maude, our reuse constructs and our background of formal techniques, we have sketched a *design concepts* for Maude specifications, and have illustrated how the reuse constructs can be used to design an object-oriented concurrent specification stepwise with incremental modifications and extensions, as it is typical in object orientation. For the demonstration of the design concepts, we have developed a specification of an airport.

Comparing our approach to other object-oriented concurrent languages shows that the language itself is smaller than other object-oriented concurrent languages. On the other hand, our framework of reuse constructs, formal techniques and reuse and design concepts is much richer than other object-oriented concurrent frameworks. Designing such a framework was possible because Maude and the formal techniques are based on few but powerful basic concepts, and because Maude abstracts from many typical object-oriented constructs which have proven to be cumbersome in other approaches.

Above all, the *role of Maude* and our framework is to provide a formal basis for the specification and refinement of concurrent systems in the object-oriented paradigm. The target programs we have in mind are programs written in an object-oriented concurrent language with constructs more concrete but similar to the ones of Maude. For the relation between a Maude specification and a program we have developed refinement and verification techniques. However, Maude specifications can also play another role in the design process. Maude is an executable specification language, and so a Maude specification is a first, executable prototype, which can be validated.

It is important in our work that Maude is a convenient specification language, i.e., that specifications in Maude with the reuse concepts and the programming method are abstract, brief, modular and comprehensible. The examples and, in particular, the airport specification illustrate the main advantage of object orientation, namely its support in modeling a system following intuition. Hereby, not only objects and classes but also the reuse principles reflect basic and natural ways of structuring and reusing specifications, as we have demonstrated in the example of the airport. The reuse concepts and the programming method we developed are essential in making Maude as specification language adequate for the object-oriented programming paradigm. However, since Maude is executable, the basic principles of the language, the reuse constructs, are so concrete that they could be features of a programming language.

We have restricted ourselves in this work to the formal world and have not dealt with several relevant issues. Our work is not isolated. For implementations of rewrite systems, we refer to [BKK+96b, Eke96, FD97, Sch97] and to the Rewrite Rule Machine [AGL+92, LMOMR94]. For the connection of our formal techniques of specifications to the semi-formal, graphical object-oriented analysis and design methods, we refer to [Nic94, NW93, WK96]. The examples of specifications and applications of formal methods, the bounded buffer, a communication protocol and the specification of an airport, as well as other examples of specifications in [Mes96], demonstrate that Maude, in combination with our set of reuse constructs, is a general and very expressive language and that our formal methods are powerful enough to cope with the typical applications of object-oriented technology: complex heterogeneous systems.

In our work, we have designed a framework for object-oriented specifications and concurrent systems. Important is that Maude is a specification language, a concurrent language and an object-oriented language. Maude is a specification language, since it is abstract and specifies characteristic properties of object-oriented systems and since we have developed the techniques of verification and refinement typical for specification frameworks. Maude is a concurrent language, since we have inter-object concurrency with very flexible communication and synchronization mechanisms and and a very fine-grained concurrency.

Finally, Maude is object-oriented, since we provide all the characteristics of object orientation: objects and classes, encapsulation, concepts for design and reuse. However, we have abandoned established concepts in order to succeed in the design of an object-oriented concurrent programming paradigm.

Bibliography

[ACS96] R. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. In U. Montanari and V. Sassone, editors, *7th Int. Conf. on Concurrency Theory (CONCUR'96)*, Lecture Notes in Computer Science 1119, pages 147–162. Springer-Verlag, 1996.

[AdBKR86] P. America, J. de Bakker, J.N. Kok, and J. Rutten. Operational semantics of a parallel object-oriented language. In *Proc. 13th Int. Symp. on Principles of Programming Languages (POPL'86)*, pages 194–208. ACM Press, 1986.

[AFK+93] G. Agha, S. Frølund, W.Y. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 3–21. MIT Press, 1993.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* Artificial Intelligence. MIT Press, 1986.

[AGL+92] H. Aida, J.A. Goguen, S. Leinwand, P. Lincoln, J. Meseguer, B. Taheri, and T. Winkler. Simulation and performance estimation for the rewrite rule machine. In *Proc. FRONTIERS '92*, pages 336–344. IEEE Computer Society Press, 1992.

[AM95] M. Abadi and S. Merz. An abstract account of composition. In J. Wiedermann and P. Hajek, editors, *Proc. 20th Int. Symp. on Mathematical Foundations of Computer Science (MFCS'95)*, Lecture Notes in Computer Science 969, pages 499–508, Berlin, 1995. Springer-Verlag.

[Ame87] P. America. POOL-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[Ame90] P. America. Designing an object-oriented programming language with behavioural subtyping. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Proc. REX/FOOLS Workshop*, Lecture Notes in Computer Science 489, pages 60–90. Springer-Verlag, 1990.

[Ame92] P. America. Formal techniques for parallel object-oriented languages. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proc. ECOOP'91 Workshop on Object-Based Concurrent Computing*, Lecture Notes in Computer Science 612, pages 119–140. Springer-Verlag, 1992.

[AW89] E. Astesiano and M. Wirsing. Bisimulation in algebraic specifications. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, Algebraic Techniques, Vol. 1, pages 1–32. Academic Press, London, 1989.

[Bau85] F.L. Bauer. *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L*. Lecture Notes in Computer Science 183. Springer-Verlag, 1985.

[BDMN73] G. Birtwistle, O.J. Dahl, B. Myhrtag, and K. Nygaard. *Simula Begin*. Auerbach Press, 1973.

[BE93] A. Birrer and T. Eggenschwiler. Frameworks in the financial engineering domain: An experience report. In O. Nierstrasz, editor, *European Conf. on Object-Oriented Programming (ECOOP'93)*, Lecture Notes in Computer Science 707, pages 21–35. Springer-Verlag, 1993.

[Ben90] J.K. Bennett. Experience with distributed Smalltalk. *Software – Practice and Experience*, 20(2):157–180, 1990.

[BFG+93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Natareth, F. Regensburger, O. Slotosch, and K. Stolen. The requirement and design specification language SPECTRUM: An informal introduction, Version 1.0, Part I. Technical Report TUM-I9311, TU München, 1993.

[BH93] M. Bidoit and R. Hennicker. A general framework for modular implementations of modular system specifications. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT '93: Theory and Practice of Software Development*, Lecture Notes in Computer Science 668, pages 199–214. Springer-Verlag, 1993.

[BHW95] M. Bidoit, R. Hennicker, and M. Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming*, 2-3(25):146–186, 1995.

[BKK96a] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. *Electronic Notes in Theoretical Computer Science*, 4:168–188, 1996. Proc. First International Workshop on Rewriting Logic and its Applications.

[BKK+96b] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. *Electronic*

Notes in Theoretical Computer Science, 4:35–50, 1996. Proc. First International Workshop on Rewriting Logic and its Applications.

[BMS96] J.P. Bahsoun, S. Merz, and C. Servières. Modular description and verification of concurrent objects. In *First Workshop on Distributed and Concurrent Objects*, Lecture Notes in Computer Science 1107, pages 168–186. Springer-Verlag, 1996.

[Boo91] G. Booch. *Object Oriented Analysis and Design with Applications.* Benjamin Cummings, 1991.

[BR95] G. Booch and J. Rumbaugh. *Unified Method for Object-Oriented Development Version 0.8.* Rational Software Corporation, 1995.

[Bra92] J.C. Bradfield. *Verifying Temporal Properties of Systems.* Birkhäuser, 1992.

[Bre91] R. Breu. *Algebraic Specification Techniques in Object-Oriented Programming Environments.* Lecture Notes in Computer Science 562. Springer-Verlag, 1991.

[Bru93] G. Bruns. A practical technique for process abstraction. In E. Best, editor, *4th Int. Conf. on Concurrency Theory (CONCUR'93)*, Lecture Notes in Computer Science 715, pages 37–49. Springer-Verlag, 1993.

[Bur85] A. Burns. *Concurrent programming in Ada.* The Ada companion series. Cambridge University Press, 1985.

[Car93] D. Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993. B. Meyer (ed.), Concurrent Object-Oriented Programming.

[CC78] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *Proc. 2nd IFIP TC-2 Working Conf. on Formal Description of Programming Concepts*, pages 237–277. North-Holland, August 1978.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, 1986.

[Cha95] C. Chambers. The Cecil Language: Specification and rationale, Version 2.1, 1995.
Available at: `http://www.cs.washington.edu/research/projects/cecil/`.

[CK93] K.M. Chandy and C. Kesselmann. CC++: A declarative concurrent object-oriented programming notation. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 282–313. MIT Press, 1993.

[CM93] M. Cerioli and J. Meseguer. May I borrow your logic? In A.M. Borzyszkowski and S. Sokolowski, editors, *Proc. 18th Int. Symp. on Mathematical Foundations of Computer Science (MFCS'93)*, Lecture Notes in Computer Science 711. Springer-Verlag, 1993.

[CM96] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. *Electronic Notes in Theoretical Computer Science*, 4:309–320, 1996. Proc. First International Workshop on Rewriting Logic and its Applications.

[CMMS91] L. Cardelli, S. Martini, J.C. Mitchell, and A. Scedrov. An extension of system F with subtyping. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software (TACS '91)*, Lecture Notes in Computer Science 526, pages 750–770. Springer-Verlag, 1991.

[Col89] M.I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[Cus91] E. Cusack. Refinement, Conformance and Inheritance. *Formal Aspects of Computing*, 3(2), April 1991.

[CY91a] P. Coad and E. Yourdon. *Object Oriented Analysis (2nd Edition).* Yourdon Press, 1991.

[CY91b] P. Coad and E. Yourdon. *Object Oriented Design.* Yourdon Press, 1991.

[Dam93] M. Dam. Model checking mobile processes. In E. Best, editor, *4th Int. Conf. on Concurrency Theory (CONCUR'93)*, Lecture Notes in Computer Science 715, pages 22–36. Springer-Verlag, 1993. Full version: SICS Report RR94:1, 1994.

[DE95] G. Denker and H.-D. Ehrich. Action Reification In Object Oriented Specifications. In R.J. Wieringa and R.B. Feenstra, editors, *Proc. ISCORE Workshop Amsterdam, Sep. 94*, pages 103–118. World Scientific, 1995.

[Den96a] G. Denker. Semantic refinement of concurrent object systems based on serializability. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object Orientation with Parallelism and Persistence*, pages 105–126. Kluwer, 1996.

[Den96b] G. Denker. *Verfeinerung in objektorientierten Spezifikationen: Von Aktionen zu Transaktionen*, volume 6 of *DISDBIS*. infix-Verlag, 1996.

[DG93] G. Denker and M. Gogolla. Translating TROLL light concepts to Maude. In H. Ehrig and F. Orejas, editors, *Proc. 9th Workshop on Abstract Data Types – 4th Compass Workshop (ADT'92)*, Lecture Notes in Computer Science 785, pages 173–187. Springer-Verlag, 1993.

[DGG94] D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: Abstractions preserving $\forall$CTL*, $\exists$CTL* and CTL*. In U. Montanari and E.-R. Olderog, editors, *Proc. Procomet'94 (IFIP TC2 Working Conference on Programming Concepts, Methods and Calculi)*. North-Holland, 1994.

[DS90] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[EDS93] H.-D. Ehrich, G. Denker, and A. Sernadas. Constructing Systems as Object Communities. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. Theory and Practice of Software Development (TAPSOFT'93)*, Lecture Notes in Computer Science 668, pages 453–467. Springer-Verlag, 1993.

[Eke96] S. Eker. Fast matching in combination of regular equational theories. *Electronic Notes in Theoretical Computer Science*, 4:90–109, 1996. Proc. First International Workshop on Rewriting Logic and its Applications.

[EW96] D. Eichelberg and B. Wagner. Met++: an object-oriented multimedia application framework. Notes of the ECOOP'96 tutorial, 1996.

[FD97] K. Futatsugi and R. Diaconescu. CafeOBJ report, 1997. Available at: `http://ldl-www.jaist.ac.jp:8080/cafeobj/`.

[FM94] J.L. Fiadeiro and T. Maibaum. Sometimes "tomorrow" is "sometime" – action refinement in a temporal logic of objects. In D.M. Gabbay and H.J. Ohlbach, editors, *Proc. First Int. Conf. on Temporal Logic (ICTL'94)*, Lecture Notes in Computer Science 827, pages 48–66. Springer-Verlag, 1994.

[FM96] J.L. Fiadeiro and T. Maibaum. Modal object calculi. In J.-P. Bahsoun, J. Fiadeiro, D. Galmiche, and A. Yonezawa, editors, *Proc. of the ECOOP'96 workshop on Proof Theory of Concurrent Object-Oriented Programming*, 1996.

[FMQ96] G. Ferrari, U. Montanari, and P. Quaglia. A π-calculus with explicit substitutions. *Theoretical Computer Science*, 168(1):53–103, 1996.

[FN96] K. Futatsugi and A. Nakagawa. An overview of Cafe project. In *First CafeOBJ workshop, Yokohama, Japan*, 1996. Available at: `http://ldl-www.jaist.ac.jp:8080/cafeobj/abstracts/ocp.html`.

[Frø92] S. Frølund. Inheritance of synchronisation constraints in concurrent object-oriented programming languages. In O. Lehrmann Madsen, editor, *European Conf. on Object-Oriented Programming (ECOOP'92)*, Lecture Notes in Computer Science 615, pages 185–196. Springer-Verlag, 1992.

[GFG96] B. Garbinato, P. Felber, and R. Guerraoui. Protocol classes for designing reliable distributed environments. In P. Cointe, editor, *European Conf. on*

Object-Oriented Programming (ECOOP'96), Lecture Notes in Computer Science 1098, pages 316–343. Springer-Verlag, 1996.

[GHW85] J.V. Guttag, J.J. Horning, and J.M. Wing. LARCH in five easy pieces. Technical report, Digital Systems Research Center, 1985.

[GKK+88] J.A. Goguen, C. Kirchner, H. Kirchner, A. Megrélis, J. Meseguer, and T. Winkler. An introduction to OBJ3. In J.-P. Jouannaud and S. Kaplan, editors, *Proc. First Int. Workshop on Conditional Term Rewriting Systems*, Lecture Notes in Computer Science 308, pages 258–263. Springer-Verlag, 1988.

[GM92] J.A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[Gol83] A. Goldberg. *Smalltalk 80: the Interactive Programming Environment.* Addison-Wesley, 1983.

[GR83] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation.* Addison-Wesley, 1983.

[Har95] M. Harant. Objekt-orientierte Spezifikation von Graphalgorithmen. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1995.

[Hen89] R. Hennicker. Observational implementations. In B. Monien and R. Cori, editors, *Proc. 6th Ann. Symp. on Theoretical Aspects of Computer Science (STACS'89)*, Lecture Notes in Computer Science 349, pages 59–71. Springer-Verlag, 1989.

[HJ96] S. Hodges and C.B. Jones. Non interference properties of a concurrent object-based language: Proofs based on an operational semantics. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object Orientation with Parallelism and Persistence*, pages 1–22. Kluwer, 1996.

[HL87] C.-H. Huang and C. Lengauer. The derivation of systolic implementations of programs. *Acta Informatica*, 24(6):595–632, November 1987.

[HN96] A.E. Haxthausen and F. Nickl. Pushouts of order-sorted algebraic specifications. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST 96)*, Lecture Notes in Computer Science 1101, pages 132–147. Springer-Verlag, 1996.

[Hoa85] C.A.R. Hoare. *Communicating sequential processes.* Prentice/Hall, Int., 1985.

[HP95] M. Hofmann and B. Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995.

[HR95] U. Hensel and H. Reichel. Defining equations in terminal coalgebras. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *10th Workshop on Specification of Abstract Data Types Joint with the 5th COMPASS Workshop, Selected papers*, Lecture Notes in Computer Science 906, pages 307–318. Springer-Verlag, 1995.

[HS95] M. Hofmann and D. Sannella. On behavioral abstraction and behavioral satisfaction in higher-order logic. In *Proc. Theory and Practice of Software Development (TAPSOFT'95)*, Lecture Notes in Computer Science 915, pages 247–261. Springer-Verlag, 1995.

[HS96] R. Hennicker and C. Schmitz. Object-oriented implementation of abstract data type specifications. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST 96)*, Lecture Notes in Computer Science 1101, pages 163–179. Springer-Verlag, 1996.

[HSJ+94] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kusch. Revised version of the modelling language TROLL (Version 2.0). Technical Report Informatik-Bericht 94–03, TU Braunschweig, 1994.

[Jac96a] B. Jacobs. Coalgebraic specifications and models of deterministic hybrid systems. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST 96)*, Lecture Notes in Computer Science 1101, pages 520–535. Springer-Verlag, 1996.

[Jac96b] B. Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *European Conf. on Object-Oriented Programming (ECOOP'96)*, Lecture Notes in Computer Science 1098, pages 210–231. Springer-Verlag, 1996.

[Jac96c] B. Jacobs. Objects and classes, co-algebraically. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object Orientation with Parallelism and Persistence*, pages 83–103. Kluwer, 1996.

[JCJÖ92] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering – A Use Case Driven Approach.* Addison-Wesley/ACM Press, 1992.

[Jon89] H.B.M. Jonkers. An introduction to COLD-K. In J.A. Bergstra and M. Wirsing, editors, *Algebraic Methods: Theory, Tools and Applications*, volume 394 of *Lecture Notes in Computer Science*, pages 139–205. Springer-Verlag, 1989.

[Jon92] C.B. Jones. An Object-Based Design Method for Concurrent Programs. Technical Report UMCS-92-12-1, Manchester University, 1992.

[Jon93a] C.B. Jones. Constraining Interference in an Object-Based Design Method. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Theory and Practice of Software Development (TAPSOFT'93)*, Lecture Notes in Computer Science 668, pages 136–150. Springer-Verlag, 1993.

[Jon93b] C.B. Jones. Defining the Semantics of an Object-Based Design Notation. In E. Best, editor, *4th Int. Conf. on Concurrency Theory (CONCUR'93)*, Lecture Notes in Computer Science 715, pages 158–172. Springer-Verlag, 1993.

[Jon93c] C.B. Jones. Process-algebraic foundations for an object-based design notation. Technical Report UMCS-93-10-1, Manchester University, 1993.

[Jon93d] C.B. Jones. Reasoning about Interference in an Object-Based Design Method. In J.C.P. Woodcock and P.G. Larsen, editors, *Industrial-Strength Formal Methods (FME'93)*, Lecture Notes in Computer Science 670, pages 1–18. Springer-Verlag, 1993.

[KK93] L.V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. *ACM SIGPLAN Notices*, 28(10):91–108, 1993.

[KLMW94] A. Kemper, P.C. Lockemann, G. Moerkotte, and H.D. Walter. A natural model for complex applications. *Journal of Intelligent Information Systems (JIIS)*, 3(2):133–155, 1994.

[KMWZ91] A. Kemper, G. Moerkotte, H-D. Walter, and A. Zachmann. GOM: A strongly-typed persistent object model with polymorphism. In *Proc. of Datenbanksysteme in Büro, Technik u. Wissenschaft (BTW)*, Informatik-Fachberichte 270, pages 198–217. Springer-Verlag, 1991.

[Kon92] D. Konstantas. Hybrid update. In D. Tsichritzis, editor, *Object Frameworks*, pages 109–118. Centre Universitaire d'Informatique University of Geneva, July 1992.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, December 1983.

[Kri97] N. Krivokapiè. Synchronization in a Distributed Object System. In K. R. Dittrich and A. Geppert, editors, *Proc. of Datenbanksysteme in Büro, Technik u. Wissenschaft*, Informatik aktuell, pages 332–341. Springer-Verlag, 1997.

[Krö87] F. Kröger. *Temporal Logic of Programs*. European Association for Theoretical Computer Science: EATCS monographs on theoretical computer science. Springer-Verlag, 1987.

[KW95] P. Kosiuczenko and M. Wirsing. Timed rewriting logic for the specification of time-sensitive systems. In H. Schwichtenberg, editor, *Proc. the International Summer School on Proof and Computation*, NATO-ASI Series. Springer-Verlag, 1995.

[Lan96] C. Landauer. Discrete event systems in rewriting logic. *Electronic Notes in Theoretical Computer Science*, 4:309–320, 1996. Proc. First International Workshop on Rewriting Logic and its Applications.

[Lar93] L.G. Larsen. The expressive power of implicit specifications. *Theoretical Computer Science*, 114:119–147, 1993.

[Lec92] U. Lechner. Eine strukturierte operationelle Semantik für CCS, CSP und occam 2: Eine Anwendung des SMoLCS-Ansatzes. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1992.

[Len82] C. Lengauer. A methodology for programming with concurrency: The formalism. *Science of Computer Programming*, 2(1):19–52, October 1982.

[LGH97] C. Lengauer, S. Gorlatch, and C. Herrmann. The static parallelization of loops and recursions. In *Proc. 11th Ann. Int. Symp. on High Performance Computing Systems (HPCS'97)*, 1997. To appear.

[LGS+95] C. Loiseaux, A. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstraction for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–45, 1995.

[LH82] C. Lengauer and E.C.R. Hehner. A methodology for programming with concurrency: An informal presentation. *Science of Computer Programming*, 2(1):1–18, October 1982.

[LH86] C. Lengauer and C.-H. Huang. A mechanically certified theorem about optimal concurrency of sorting networks. In *Proc. 13th Ann. ACM Symp. on Principles of Programming Languages (POPL)*, pages 307–317. ACM, 1986.

[LMOMR94] P. Lincoln, N. Martí-Oliet, J. Meseguer, and L. Ricciulli. Compiling rewriting onto SIMD and MIMD/SIMD machines. In *Proc. Sixth Int. Conf. on Parallel Architectures and Languages Europe (PARLE '94)*, Lecture Notes in Computer Science 817. Springer-Verlag, 1994.

[LT91] K.G. Larsen and B. Thomsen. Partial specifications and compositional verification. *Theoretical Computer Science*, 88:15–32, 1991.

[MCR93] D.R. Maeng, J.W. Cho, and K. Ryu. Concurrency and inheritance in actor-based object-oriented languages. *Systems Software*, 20:53–67, 1993.

[Mes90a] J. Meseguer. A logical theory of concurrent objects. *ACM SIGPLAN Notices*, 25(10):101–115, October 1990. (Proc. OOPSLA/ECOOP'90).

[Mes90b] J. Meseguer. Rewriting as a unified model of concurrency. In J.C.M. Baeten and J.W. Klop, editors, *2nd Int. Conf. on Concurrency Theory (CONCUR'90)*, Lecture Notes in Computer Science 458, pages 384–400. Springer-Verlag, 1990.

[Mes92a] J. Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes92b] J. Meseguer. Multiparadigm logic programming. In H. Kirchner and G. Levi, editors, *Proc. Third Int. Conf. on Algebraic and Logic Programming*, Lecture Notes in Computer Science 632, pages 158–200. Springer-Verlag, 1992.

[Mes93a] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[Mes93b] J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In O. Nierstrasz, editor, *European Conf. on Object-Oriented Programming (ECOOP'93)*, Lecture Notes in Computer Science 707, pages 220–246. Springer-Verlag, 1993.

[Mes96] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In U. Montanari and V. Sassone, editors, *7th Int. Conf. on Concurrency Theory (CONCUR'96)*, Lecture Notes in Computer Science 1119, pages 331–372. Springer-Verlag, 1996.

[Mey92] B. Meyer. *EIFFEL The Language.* Prentice-Hall, 1992.

[Mey93] B. Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, September 1993.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice/Hall, Int., 1989.

[MMO95] J. Meseguer and N. Martí-Oliet. From abstract data types to logical frameworks. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *10th Workshop on Specification of Abstract Data Types joint with the 5th COMPASS Workshop, Selected papers*, Lecture Notes in Computer Science 906, pages 48–80. Springer-Verlag, 1995.

[MOM94] N. Martí-Oliet and J. Meseguer. General logics and logical frameworks. In D. Gabbay, editor, *What is a logical system?* Oxford University Press, 1994.

[MOM96] N. Martí-Oliet and J. Meseguer. Action and change in rewriting logic. In R. Pareschi and B. Fronhoefer, editors, *Theoretical Approaches to Dynamic Worlds in Computer Science and Artificial Intelligence*. Kluwer, 1996.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100(1):1–40 and 41–77, 1992.

[MPW93] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 25:267–310, 1993.

[MQ93] J. Meseguer and X. Qian. A logical semantics for object-oriented databases. In *Proc. ACM SIGMOD Conference on the Management of Data*, pages 89–98. ACM, 1993.

[MW92] J. Meseguer and T. Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, Lecture Notes in Computer Science 574, pages 253–293. Springer-Verlag, 1992.

[MY93] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in concurrent object-oriented languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[Nic94] F. Nickl. Ablaufspezifikation durch Datenflußdiagramme und Axiome. In B. Wolfinger, editor, *Innovationen bei Rechen- und Kommunikationssystemen, 24. GI-Jahrestagung*, pages 10–19. Springer-Verlag, 1994.

[Nie91] O. Nierstrasz. The next 700 concurrent object-oriented languages – reflections on the future of object-based concurrency. Technical report, Centre Universitaire d'Informatique, University of Geneva, 1991.

[Nie92] O. Nierstrasz. A tour of Hybrid – a language for programming with active objects. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 167–182. Prentice-Hall, 1992.

[Nie93] O. Nierstrasz. Regular types for active objects. *ACM SIGPLAN Notices*, 28(10):1–15, October 1993. Proc. OOPSLA '93.

[NS96] U. Nestmann and M. Steffen. Correct transformational design of concurrent search structures. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object Orientation with Parallelism and Persistence*, pages 23–42. Kluwer, 1996.

[NW93] F. Nickl and M. Wirsing. A formal approach to requirements engineering. In D. Bjørner and M. Broy, editors, *Formal Methods in Programming and their Applications*, Lecture Notes in Computer Science 735, pages 312–334, 1993.

[ÖKW96] P. Ölveczki, P. Kosiuczenko, and M. Wirsing. Steamboiler specification problem: an algebraic object-oriented solution. In J.R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Application: Specifying and Programming the Steam-Boiler Control*, Lecture Notes in Computer Science 1165, pages 379–402. Springer-Verlag, 1996.

[ÖM96] P.C. Ölveczky and J. Meseguer. Specifying real-time constraints in rewriting logic. *Electronic Notes in Theoretical Computer Science*, 4:283–308, 1996. Proc. First International Workshop on Rewriting Logic and its Applications.

[ONS93] F. Orejas, M. Navarro, and A. Sanchez. Implementation and behavioural equivalence: A survey. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specifications*, Lecture Notes in Computer Science 655, pages 93–125. Springer-Verlag, 1993.

[PMO96] I. Pita and N. Marti-Oliet. A Maude specification of an object oriented database model of telecommunication networks. *Electronic Notes in Theoretical Computer Science*, 4:404–422, 1996. Proc. First International Workshop on Rewriting Logic and its Applications.

[PS94] J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems*. Wiley, 1994.

[PS96a] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[PS96b] W. Pree and H. Sikora. Application of design patterns in commercial domains. Notes of the ECOOP'96 tutorial, 1996.

[PS97] B. Pierce and M. Steffen. Higher-order subtyping. *Theoretical Computer Science*, 176(1–2):235–282, 1997.

[PT94] B.C. Pierce and D.N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, 1994.

[PT97] B.C. Pierce and D.N. Turner. Programming in the π-calculus, an experiment in concurrent language design, 1997. Tutorial Notes for PICT Version 3.5g, Available together with Pict,
Available at `http://www.cs.indiana.edu/hyplan/pierce/pierce/ftp/`.

[PW96] A. Philippou and D. Walker. On transformation of concurrent object programs. In U. Montanari and V. Sassone, editors, *7th Int. Conf. on Concurrency Theory (CONCUR'96)*, Lecture Notes in Computer Science 1119, pages 131–146. Springer-Verlag, 1996.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[Rei95] H. Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, 5:129–152, 1995.

[Rut95] J.J.M.J. Rutten. A calculus of transition systems (towards universal coalgebra). Technical Report CS-R9503, Computer Science Departement of Software Technology, 1995.

[Sal93] B. Salmansberger. Objektorientierte Spezifikation von verteilten Systemen in Maude am Beispiel eines Flughafens. Diplomarbeit, Fakultät für Mathematik und Informatik, University of Passau, 1993.

[SB88] M. Schelvis and E. Bledoeg. The implementation of a distributed Smalltalk. In S. Gjessing and K. Nygaard, editors, *European Conf. on Object-Oriented Programming (ECOOP'88)*, Lecture Notes in Computer Science 322, pages 212–232. Springer-Verlag, 1988.

[Sch96] B. Schaeffer. Advanced Smalltalk: Elegance and efficiency. Notes of the ECOOP'96 tutorial, 1996.

[Sch97] School of Information Science, Japan Advanced Institute of Science and Technology. *CafeOBJ User's Manual, Ver. 1.2.0*, 1997.
Available at: `http://ldl-www.jaist.ac.jp:8080/cafeobj/`.

[SCK+95] B. Steffen, A. Claßen, M. Klein, J. Knoop, and T. Margaria. The fixpoint analysis machine. In I. Lee and S.A. Smolka, editors, *6th Int. Conf. on Concurrency Theory (CONCUR'95)*, Lecture Notes in Computer Science 962, pages 72–87. Springer-Verlag, 1995.

[SCM95] B. Steffen, A. Claßen, and T. Margaria. The META-Frame: An environment for flexible tool management. In *TAPSOFT '95: Theory and Practice of Software Development (TAPSOFT '95), Int. Joint Conference CAAP/FASE*, Lecture Notes in Computer Science 915, pages 791–792. Springer-Verlag, 1995.

[SF91] C. Sernadas and J. Fiadeiro. Towards a Object-Oriented Formal Modeling. *Data and Knowledge Engineering*, 6(6):479–508, October 1991.

[SG92] H. Saleh and P. Gautron. A concurrency control mechanism for C++ objects. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *ECOOP'91 Workshop on Object-Based Concurrent Computing*, Lecture Notes in Computer Science 612, pages 195–210. Springer-Verlag, 1992.

[Sha94] R. Sharp. *Principles of Protocol Design.* Prentice-Hall, 1994.

[SI94] B. Steffen and A. Ingolfsdottir. Characteristic formulae for finite state processes. *Information and Computation*, 110(1), 1994.

[SJE92] G. Saake, R. Jungclaus, and H.-D. Ehrich. Object-Oriented Specification and Stepwise Refinement. In De Meer, Heymer, and Roth, editors, *Proc. of IFIP TC6 Int. Workshop on Open Distributed Processing, Berlin, October 1991 (IFIP Transactions C, Vol 1)*, pages 99–121. North-Holland, 1992.

[Ski94] D.B. Skillicorn. *Foundations of Parallel Programming*. Cambridge International Series on Parallel Computation. Cambridge University Press, 1994.

[SKV96] B. Steffen, J. Knoop, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, 1996.

[SM92] A. Shlaer and S.J. Mellor. *Object Lifecycles: Modeling the world in states*. Prentice-Hall, 1992.

[SMC96] B. Steffen, T. Margaria, and A. Claßen. Heterogeneous analysis and verification for distributed systems. *SOFTWARE: Concepts and Tools*, 17:13–25, 1996.

[Smo94] Gert Smolka. An Oz primer. DFKI Oz documentation series, Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), 1994.

[ST85] D.T. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Proc. Joint Conf. on Theory and Practice of Software Development (TAPSOFT'85)*, Lecture Notes in Computer Science 185, pages 308–322. Springer-Verlag, 1985.

[Ste93] B. Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, 21:115–139, 1993.

[Sti92] C. Stirling. Modal and temporal logics for processes. Technical Report Reports ECS-LFCS-92-221, Computer Science Department, University of Edinburgh, June 1992.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Tal96] C.L. Talcott. An actor rewriting theory. *Electronic Notes in Theoretical Computer Science*, 4:309–320, 1996. Proc. First International Workshop on Rewriting Logic and its Applications.

[Vas94] V.T. Vasconcelos. Typed concurrent objects. In M. Tokoro and R. Pareschi, editors, *European Conf. on Object Oriented Programming (ECOOP'94)*, Lecture Notes in Computer Science 821, pages 100–117. Springer-Verlag, 1994.

[vG89] R.J. van Glabbeek and U. Goltz. Equivalence notion for concurrent systems and refinement fo actions. In A. Kreczmar and G. Mirkowska, editors, *Proc. 14th Int. Symp. on Mathematical Foundations of Computer Science (MFCS'89)*, Lecture Notes in Computer Science 379, pages 237–248. Springer-Verlag, 1989.

[vG90a] R.J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions*. PhD thesis, Centrum voor Wiskunde en Informatica, Free Univ. of Amsterdam, 1990.

[vG90b] R.J. van Glabbeek and U. Goltz. Refinement of actions in causality based models. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, Lecture Notes in Computer Science 430, pages 267–300. Springer-Verlag, 1990.

[Vir96] P. Viry. Input/output for ELAN. *Electronic Notes in Theoretical Computer Science*, 4:51–64, 1996. Proc. First International Workshop on Rewriting Logic and its Applications.

[VT93] V.T. Vasconcelos and M. Tokoro. A typing system for a calculus of objects. In *Object Technologies for Advanced Software*, Lecture Notes in Computer Science 742, pages 460–474. Springer-Verlag, 1993.

[Wal91] D. Walker. π-calculus semantics of object-oriented programming languages. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software, TACS '91*, Lecture Notes in Computer Science 526, pages 532–547. Springer-Verlag, 1991.

[Wal94] D. Walker. Algebraic proofs of properties of objects. In D. Sannella, editor, *5th European Symposium on Programming*, Lecture Notes in Computer Science 788, pages 501–516. Springer-Verlag, 1994.

[WBWW90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

[Wed93] C. Wedler. Verifikation objektorientierter verteilter Systeme. Diplomarbeit, Universität Erlangen, October 1993.

[Wir87] M. Wirsing. Structured algebraic specifications: a kernel language. *Theoretical Computer Science*, 42:123–249, 1987.

[Wir90] M. Wirsing. Algebraic specification. In J.V. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 675–788. Elsevier (North-Holland), 1990.

[Wir95] M. Wirsing. Algebraic specification languages: An overview. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *10th Workshop on Specification of Abstract Data Types Joint with the 5th COMPASS Workshop, Selected papers*, Lecture Notes in Computer Science 906, pages 81–115. Springer-Verlag, 1995.

[WK96] M. Wirsing and A. Knapp. A formal approach to object-oriented software engineering. *Electronic Notes in Theoretical Computer Science*, 4:321–359, 1996. Proc. First International Workshop on Rewriting Logic and its Applications.

[WK97] M. Wirsing and P. Kosiuczenko. Timed rewriting logic. In H. Schwichtenberg, editor, *Proc. International Summer School on Proof and Computation, Marktoberdorf*, NATO-ASI Series. Springer-Verlag, 1997. To appear.

[WNL95] M. Wirsing, F. Nickl, and U. Lechner. Concurrent object-oriented specification in spectrum. In Y. Inagaki, editor, *Workshop on Algebraic and Object-Oriented Approaches to Software Science, Nagoya/Japan*, pages 39–70. Nagoya University, 1995. Also appeared as: Technical Report 9418, Institut für Informatik, Ludwig-Maximilians-Universität München, December 1994.

[WZ88] P. Wegner and S.B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *European Conf. on Object-Oriented Programming (ECOOP'88)*, Lecture Notes in Computer Science 322, pages 55–77. Springer-Verlag, 1988.

[Yon90] A. Yonezawa. *ABCL An Object-Oriented Concurrent System.* The MIT Press, 1990.

[YT87] Y. Yokote and M. Tokoro. Concurrent programming in ConcurrentSmalltalk. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. MIT Press, Cambridge, Massachusetts, 1987.

Appendix A

Object-Oriented Specifications in Maude

A.1 Basic specifications

The specification ACZ-CONFIGURATION as well as NAT and RWL, which contains the specification of the transition relation ==>, can be found in [Sch97].

```
module EXT-ACZ-CONFIGURATION {
  import {
    protecting (NAT)
    protecting (ACZ-CONFIGURATION)
  }

  signature {
   op mkoid   : ClassId Nat -> ObjectId
   op incoid  : ObjectId -> ObjectId
  }

  axioms {
    var  N : Nat
    var  A : ClassId

    eq [inc]: incoid(mkoid(A,N)) =  mkoid(A,s(N)) .
  }
}


module LIST-ENRICHED {
  import {
    protecting (NAT)
```

```
    protecting (LIST)
  }

  signature {
    op length : List -> Nat
    op first  : List -> Elem
    op removefirst : List -> List
    op iselem : Elem List -> Bool
    op select : List -> Elem
    op remove : Elem List -> List
  }

  axioms {
    var  L    : List
    vars E E' : Elem

  eq [Eq3]: length(eps) = 0 .
  eq [Eq4]: length(E L) = 1 + length(L) .
  eq [Eq5]: length(L E) = 1 + length(L) .

  eq [Eq6]: first(E L) = E .
  eq [Eq7]: removefirst(E L) = L .

  eq [Eq8]: iselem(E, eps) = false .
  eq [Eq9]: iselem(E, L E) = true .
  eq [Eq10]: iselem(E, L E') = (E == E') .

  eq [Eq11]: select(L E) = E .

  eq  [Eq12]: remove(E, L E) = L .
  ceq [Eq13]: remove(E, L E') = remove(E, L) E' if E =/= E' .
  }
}
```

A.2 Specification `AIRPORT`

We present the fragment of a specification of an airport which is sufficient to interpret the modules given in Sect. 2.4 with CafeOBJ. This specification illustrates the relation between the different modules and provides the declarations of data types that have not been given before.

```
in LIST
```

```
in LIST-ENRICHED
in EXT-ACZ-CONFIGURATION
in SUBCONFIGURATION
in MSG-ALGEBRA

module TIME {
 import {
   protecting (NAT)
 }

  signature {
   [Time]
   op t : Nat -> Time
   }
}

module PARTS-OF-AIRPORTS {
  import {
   protecting (LIST)
   protecting (ACZ-CONFIGURATION)
  }

  signature {
    [ObjectId < Elem]

    class Cargo { }
    class Runway { }
    class Tower  {
      free  : List
      nfree : List
    }
  }
}

in PLANES
in PLANE-COCKPIT
in SYNCHRONOUS-TAKEOFF
in FAULTY-PLANES

module PLANES-ALL-STATES {
  import {
    extending (PLANES)
  }
```

```
  signature {
    class StartingPl [Plane] {
      runway : ObjectId
    }
  }
}

in MESSAGE-IN-COCKPIT
in MESSAGE-IN-GLOBAL-STATE
in FLIGHT
in TAKEOFF-SYNCHRONOUS
```

Note that module TAKEOFF-LONG is used in Sect. 4.3.2.

Appendix B

Properties and Verification

B.1 Some Properties of Galois Connections

Corollary B.1 Let f be a monotonic function, then $f(X \cap Y) \subseteq f(X) \cap f(Y)$ and $f(X) \cup f(Y) \subseteq f(X \cup Y)$

Proof.

$$
\begin{array}{ll}
 & f(X \cap Y) \\
= & \quad \{\ S \cap S = S\ \} \\
 & f(X \cap Y) \cap f(X \cap Y) \\
\subseteq & \quad \{\ \text{monotonicity of } f,\ X \cap Y \subseteq X,\ X \cap Y \subseteq Y\ \ \} \\
 & f(X) \cap f(Y)
\end{array}
$$

$$
\begin{array}{ll}
 & f(X) \cup f(Y) \\
\subseteq & \quad \{\ \text{monotonicity of } f,\ X \subseteq X \cup Y,\ Y \subseteq X \cup Y\ \ \} \\
 & f(X \cup Y) \cup f(X \cup Y) \\
= & \quad \{\ X \cup X = X\ \} \\
 & f(X \cup Y)
\end{array}
$$

□

Lemma 4.8 (Properties of Galois connections) Let Q_1 and Q_2 be two sets and (α, γ) a Galois connection from from $\wp(Q_1)$ to $\wp(Q_2)$.

1. $(\tilde{\gamma}, \tilde{\alpha})$ is a Galois connection from $\wp(Q_2)$ to $\wp(Q_1)$.
2. α distributes over union of sets, i.e., $\alpha(S_1 \cup S_2) = \alpha(S_1) \cup \alpha(S_2)$.
3. γ distributes over intersection of sets, i.e., $\gamma(S_1 \cap S_2) = \gamma(S_1) \cap \gamma(S_2)$.
4. $\tilde{\alpha}$ distributes over intersection of sets, i.e., $\tilde{\alpha}(S_1 \cap S_2) = \tilde{\alpha}(S_1) \cap \tilde{\alpha}(S_2)$.

5. $\tilde{\gamma}$ distributes over union of sets, i.e., $\tilde{\gamma}(S_1 \cup S_2) = \tilde{\gamma}(S_1) \cup \tilde{\gamma}(S_2)$.

Proof.

1. We prove beforehand: If function α is monotonic then $\tilde{\alpha}$ is monotonic:

$$
\begin{array}{ll}
 & X \subseteq Y \\
\Rightarrow & \quad \{ \text{ set theory } \} \\
 & \overline{X} \supseteq \overline{Y} \\
\Rightarrow & \quad \{ \text{ monotonicity of } \alpha \ \} \\
 & \alpha(\overline{X}) \supseteq \alpha(\overline{Y}) \\
\Rightarrow & \quad \{ \text{ set theory } \} \\
 & \overline{\alpha(\overline{X})} \subseteq \overline{\alpha(\overline{Y})} \\
\Leftrightarrow & \quad \{ \text{ definition of the } \tilde{\alpha} \ \} \\
 & \tilde{\alpha}(X) \subseteq \tilde{\alpha}(Y)
\end{array}
$$

$$
\begin{array}{ll}
 & \tilde{\gamma} \circ \tilde{\alpha} \subseteq Id^{Q_1} \\
\Leftrightarrow & \quad \{ \text{ definition of the dual } \} \\
 & (\forall X : X \in \wp(Q_1) : \overline{\gamma(\overline{\overline{\alpha(\overline{X})}})} \subseteq X) \\
\Leftrightarrow & \quad \{ \text{ elimination of double negation } \} \\
 & (\forall X : X \in \wp(Q_1) : \overline{\gamma(\alpha(\overline{X}))} \subseteq X) \\
\Leftrightarrow & \quad \{ \text{ set theory } \} \\
 & (\forall X : X \in \wp(Q_1) : \gamma(\alpha(\overline{X})) \supseteq \overline{X}) \\
\Leftrightarrow & \quad \left\{ \text{ substitute } \overline{X} \text{ by } X \ \right\} \\
 & (\forall X : X \in \wp(Q_1) : \gamma(\alpha(X)) \supseteq X) \\
\Leftrightarrow & \quad \{ \text{ abstract notation } \} \\
 & Id^{Q_1} \subseteq \gamma \circ \alpha
\end{array}
$$

$$
\begin{array}{ll}
 & Id^{Q_2} \subseteq \tilde{\alpha} \circ \tilde{\gamma} \\
\Leftrightarrow & \quad \{ \text{ definition of the dual } \} \\
 & (\forall X : X \in \wp(Q_2) : X \subseteq \overline{\alpha(\overline{\overline{\gamma(\overline{X})}})}) \\
\Leftrightarrow & \quad \{ \text{ elimination of double negation } \} \\
 & (\forall X : X \in \wp(Q_2) : X \subseteq \overline{\alpha(\gamma(\overline{X}))}) \\
\Leftrightarrow & \quad \{ \text{ Set theory } \} \\
 & (\forall X : X \in \wp(Q_2) : \overline{X} \supseteq \alpha(\gamma(\overline{X}))) \\
\Leftrightarrow & \quad \left\{ \text{ substitute } \overline{X} \text{ by } X \ \right\} \\
 & (\forall X : X \in \wp(Q_2) : X \supseteq \alpha(\gamma(X))) \\
\Leftrightarrow & \quad \{ \text{ abstract notation } \} \\
 & \alpha \circ \gamma \subseteq Id^{Q_2}
\end{array}
$$

2. Proof idea:
$\alpha(X \cup Y) = \alpha(X) \cup \alpha(Y) \Leftrightarrow (\alpha(X \cup Y) \subseteq \alpha(X) \cup \alpha(Y) \wedge \alpha(X) \cup \alpha(Y) \subseteq \alpha(X \cup Y))$

$$\begin{array}{ll} & \alpha(X \cup Y) \subseteq \alpha(X) \cup \alpha(Y) \\ \Leftarrow & \{ \ Id \subseteq \gamma \circ \alpha \ \} \\ & \alpha(\gamma(\alpha(X)) \cup \gamma(\alpha(Y))) \subseteq \alpha(X) \cup \alpha(Y) \\ \Leftarrow & \{ \text{ Lemma 4.8 } \} \\ & \alpha(\gamma(\alpha(X) \cup \alpha(Y))) \subseteq \alpha(X) \cup \alpha(Y) \\ \Leftarrow & \{ \ \alpha \circ \gamma = Id \ \} \\ & \alpha(X) \cup \alpha(Y) \subseteq \alpha(X) \cup \alpha(Y) \end{array}$$

$$\begin{array}{ll} & \alpha(X) \cup \alpha(Y) \subseteq \alpha(X \cup Y) \\ \Leftarrow & \{ \text{ monotonicity of } \alpha \ \} \\ & \alpha(X \cup Y) \cup \alpha(X \cup Y) \subseteq \alpha(X \cup Y) \\ \Leftarrow & \{ \ S \cup S = S \ \} \\ & \alpha(X \cup Y) \subseteq \alpha(X \cup Y) \end{array}$$

3.
$$\begin{array}{ll} & \gamma(X \cap Y) \supseteq \gamma(X) \cap \gamma(Y) \\ \Leftarrow & \{ \ Id \supseteq \alpha \circ \gamma \ \} \\ & \gamma(\alpha(\gamma(X)) \cap \alpha(\gamma(Y))) \supseteq \gamma(X) \cap \gamma(Y) \\ \Leftarrow & \{ \text{ Lemma4.8 } \} \\ & \gamma(\alpha(\gamma(X) \cap \gamma(Y))) \supseteq \gamma(X) \cap \gamma(Y) \\ \Leftarrow & \{ \ \gamma \circ \alpha = Id \ \} \\ & \gamma(X) \cap \gamma(Y) \supseteq \gamma(X) \cap \gamma(Y) \end{array}$$

$$\begin{array}{ll} & \gamma(X) \cap \gamma(Y) \supseteq \gamma(X \cap Y) \\ \Leftarrow & \{ \ S \cap S = S \ \} \\ & \gamma(X) \cap \gamma(Y) \supseteq \gamma(X \cap Y) \cap \gamma(X \cap Y) \\ \Leftarrow & \{ \text{ monotonicity of } \gamma \ \} \\ & \gamma(X) \cap \gamma(Y) \supseteq \gamma(X) \cap \gamma(Y) \end{array}$$

4. If (α, γ) is a Galois connection, then $(\widetilde{\gamma}, \widetilde{\alpha})$ is a Galois connection (Lemma 4.8,(1)) and, thus, $\widetilde{\gamma}$ distributes over $\cup$ according to Lemma 4.8,2 (substitute $\widetilde{\gamma}$ for α).

5. Analogous to (4).

□

For the case $[L]\phi$ of the proof of Thm. 4.21, we need:

Lemma B.2 Let (α, γ) be a Galois connection from $\wp(Q_1)$ to $\wp(Q_2)$.
$\widetilde{\gamma} \circ \widetilde{pre}(R_2)(\alpha(L)) \circ \widetilde{\alpha} \subseteq \widetilde{pre}(R_1)(L) \;\Leftrightarrow\; \alpha \circ pre(R_1)(L) \circ \gamma \subseteq pre(R_2)(\alpha(L))$

Proof.

$$
\begin{array}{ll}
& \tilde{\gamma} \circ \widetilde{pre}(R_2)(\alpha(L)) \circ \tilde{\alpha} \subseteq \widetilde{pre}(R_1)(L) \\
\Leftrightarrow & \{ \text{ definition of the dual } \} \\
& (\forall X : X \in \wp(Q_2) : \overline{\gamma(\overline{\overline{pre(R_2)(\alpha(L))(\overline{\overline{\alpha(\overline{X})}})}})} \subseteq \overline{pre(R_1)(L)(\overline{X})}) \\
\Leftrightarrow & \{ \text{ elimination of double negation, twice } \} \\
& (\forall X : X \in \wp(Q_2) : \overline{\gamma(pre(R_2)(\alpha(L))(\alpha(\overline{X})))} \subseteq \overline{pre(R_1)(L)(\overline{X})}) \\
\Leftrightarrow & \{ \text{ set theory } \} \\
& (\forall X : X \in \wp(Q_2) : \gamma(pre(R_2)(\alpha(L))(\alpha(\overline{X}))) \supseteq pre(R_1)(L)(\overline{X})) \\
\Leftrightarrow & \left\{ \text{ substitute } \overline{X} \text{ by } X \right\} \\
& (\forall X : X \in \wp(Q_2) : \gamma(pre(R_2)(\alpha(L))(\alpha(X))) \supseteq pre(R_1)(L)(X)) \\
\Leftrightarrow & \{ \text{ abstract notation } \} \\
& \gamma \circ pre(R_2)(\alpha(L)) \circ \alpha \supseteq pre(R_1)(L) \\
\Leftrightarrow & \{ \text{ monotonicity of } \alpha \text{ and } \gamma \ \} \\
& \alpha \circ \gamma \circ pre(R_2)(\alpha(L)) \circ \alpha \circ \gamma \supseteq \alpha \circ pre(R_1)(L) \circ \gamma \\
\Leftrightarrow & \left\{ \ \alpha \circ \gamma \subseteq Id^{Q_2} \ \right\} \\
& pre(R_2)(\alpha(L)) \supseteq \alpha \circ pre(R_1)(L) \circ \gamma \\
\Leftrightarrow & \{ \text{ set theory } \} \\
& \alpha \circ pre(R_1)(L) \circ \gamma \subseteq pre(R_2)(\alpha(L))
\end{array}
$$

□

Lemma 4.11 (Alternative simulation relations) Let (Q_1, R_1), (Q_2, R_2), L_1, L_2, L be as in Def. 4.9 then

$$
\begin{array}{lrcll}
 & \alpha \circ pre(R_1)(L) \circ \gamma & \subseteq & pre(R_2)(\alpha(L)) & (1) \\
\text{iff} & \alpha \circ pre(R_1)(L) & \subseteq & pre(R_2)(\alpha(L)) \circ \alpha & (2) \\
\text{iff} & pre(R_1)(L) \circ \gamma & \subseteq & \gamma \circ pre(R_2)(\alpha(L)) & (3) \\
\text{iff} & pre(R_1)(L) & \subseteq & \gamma \circ pre(R_2)(\alpha(L)) \circ \alpha & (4)
\end{array}
$$

Proof. We prove: (1) $\Rightarrow$ (2), (2) $\Rightarrow$ (1), (1) $\Rightarrow$ (3), (3) $\Rightarrow$ (1), (3) $\Rightarrow$ (4), (4) $\Rightarrow$ (3).

(1) $\Rightarrow$ (2):

$$
\begin{array}{ll}
& \alpha \circ pre(R_1)(L) \circ \gamma \subseteq pre(R_2)(\alpha(L)) \\
\Rightarrow & \{ \text{ monotonicity } \} \\
& \alpha \circ pre(R_1)(L) \circ \gamma \circ \alpha \subseteq pre(R_2)(\alpha(L)) \circ \alpha \\
\Rightarrow & \{ \ Id \subseteq \gamma \circ \alpha \ \} \\
& \alpha \circ pre(R_1)(L) \subseteq pre(R_2)(\alpha(L)) \circ \alpha
\end{array}
$$

(2) $\Rightarrow$ (1):

$$
\begin{array}{l}
\quad \alpha \circ pre(R_1)(L) \subseteq pre(R_2)(\alpha(L)) \circ \alpha \\
\Rightarrow \quad \{ \text{ monotonicity } \} \\
\quad \alpha \circ pre(R_1)(L) \circ \gamma \subseteq pre(R_2)(\alpha(L)) \circ \alpha \circ \gamma \\
\Rightarrow \quad \{ \ \alpha \circ \gamma \subseteq Id \ \} \\
\quad \alpha \circ pre(R_1)(L) \circ \gamma \subseteq pre(R_2)(\alpha(L))
\end{array}
$$

(1) $\Rightarrow$ (3):

$$
\begin{array}{l}
\quad \alpha \circ pre(R_1)(L) \circ \gamma \subseteq pre(R_2)(\alpha(L)) \\
\Rightarrow \quad \{ \text{ monotonicity } \} \\
\quad \gamma \circ \alpha \circ pre(R_1)(L) \circ \gamma \subseteq \gamma \circ pre(R_2)(\alpha(L)) \\
\Rightarrow \quad \{ \ Id \subseteq \gamma \circ \alpha \ \} \\
\quad pre(R_1)(L) \circ \gamma \subseteq \gamma \circ pre(R_2)(\alpha(L))
\end{array}
$$

(3) $\Rightarrow$ (1):

$$
\begin{array}{l}
\quad pre(R_1)(L) \circ \gamma \subseteq \gamma \circ pre(R_2)(\alpha(L)) \\
\Rightarrow \quad \{ \text{ monotonicity } \} \\
\quad \alpha \circ pre(R_1)(L) \circ \gamma \subseteq \alpha \circ \gamma \circ pre(R_2)(\alpha(L)) \\
\Rightarrow \quad \{ \ \alpha \circ \gamma \subseteq Id \ \} \\
\quad \alpha \circ pre(R_1)(L) \circ \gamma \subseteq pre(R_2)(\alpha(L))
\end{array}
$$

(3) $\Rightarrow$ (4):

$$
\begin{array}{l}
\quad pre(R_1)(L) \circ \gamma \subseteq \gamma \circ pre(R_2)(\alpha(L)) \\
\Rightarrow \quad \{ \text{ monotonicity } \} \\
\quad pre(R_1)(L) \circ \gamma \circ \alpha \subseteq \gamma \circ pre(R_2)(\alpha(L)) \circ \alpha \\
\Rightarrow \quad \{ \ Id \subseteq \gamma \circ \alpha \ \} \\
\quad pre(R_1)(L) \subseteq \gamma \circ pre(R_2)(\alpha(L)) \circ \alpha
\end{array}
$$

(4) $\Rightarrow$ (3):

$$
\begin{array}{l}
\quad pre(R_1)(L) \subseteq \gamma \circ pre(R_2)(\alpha(L)) \circ \alpha \\
\Rightarrow \quad \{ \text{ monotonicity } \} \\
\quad pre(R_1)(L) \circ \gamma \subseteq \gamma \circ pre(R_2)(\alpha(L)) \circ \alpha \circ \gamma \\
\Rightarrow \quad \{ \ \alpha \circ \gamma \subseteq Id \ \} \\
\quad pre(R_1)(L) \circ \gamma \subseteq \gamma \circ pre(R_2)(\alpha(L))
\end{array}
$$

□

B.2 Proof of Thm. 4.21

Theorem 4.21 Let (Σ_1, E_1, T_1) and (Σ_2, E_2, T_2) be two specifications and (A_1, R_1) and (A_2, R_2) two of their state transition systems. Let $P_1 \subseteq P(\Sigma_1)$ and $P_2 \subseteq P(\Sigma_2)$ be two languages of basic propositions. Let $I_1 : \mathcal{L}_\mu(P_1) \to A_1$ and $I_2 : \mathcal{L}_\mu(P_2) \to A_2$ be two interpretation functions.

1. If $(A_1, R_1) \sqsubseteq_{(\alpha,\gamma)} (A_2, R_2)$ then α preserves $\langle\,\rangle\mathcal{L}_\mu^+(P_1)$ for I_1 and, if α is consistent with I_1, then α preserves $\langle\,\rangle\mathcal{L}_\mu(P_1)$.

2. If $(A_1, R_1) \sqsubseteq_{(\alpha,\gamma)} (A_2, R_2)$ then $\tilde{\gamma}$ preserves $[\,]\mathcal{L}_\mu^+(P_2)$ for I_2 and, if $\tilde{\gamma}$ is consistent with I_2, then $\tilde{\gamma}$ preserves $[\,]\mathcal{L}_\mu(P_2)$ for I_2.

3. If $(A_1, R_1) \simeq_{(\alpha,\gamma)} (A_2, R_2)$ then α preserves $\mathcal{L}_\mu^+(P_1)$ for I_1 and, if α is consistent with I_1, then α preserves $\mathcal{L}_\mu(P_1)$ for I_1.

Proof. To prove:

$$\alpha(|\,\phi\,|_{S_1,I_1}(v))) \ \subseteq |\,\phi\,|_{S_1,\alpha\circ I_1}(\alpha(v))$$

$\texttt{true}$: $\alpha(|\,\texttt{true}\,|_{S_1,I_1}(v)) = \alpha(A_1) \subseteq A_2 = |\,\texttt{true}\,|_{S_1,\alpha\circ I_1}(\alpha(v))$.
$\texttt{false}$: $\alpha(|\,\texttt{false}\,|_{S_1,I_1}(v)) = \alpha(\{\}) \subseteq \{\} = |\,\texttt{false}\,|_{S_1,\alpha\circ I_1}(\alpha(v))$.

$(\phi_1 \vee \phi_2)$:

$$\begin{array}{ll}
 & \alpha(|\,(\phi_1 \vee \phi_2)\,|_{S_1,I_1}(v)) \\
= & \quad \{ \text{ definition of the interpretation function } \} \\
 & \alpha(|\,\phi_1\,|_{S_1,I_1}(v) \cup |\,\phi_2\,|_{S_1,I_1}(v)) \\
= & \quad \{ \ \alpha \text{ distributes over } \cup \ \} \\
 & \alpha(|\,\phi_1\,|_{S_1,I_1}(v)) \cup \alpha(|\,\phi_2\,|_{S_1,I_1}(v)) \\
\subseteq & \quad \{ \text{ induction hypothesis: } \alpha(|\,\phi_i\,|_{S_1,I_1}(v)) \subseteq |\,\phi_i\,|_{S_1,\alpha\circ I_1}(\alpha(v)) \ \} \\
 & |\,\phi_1\,|_{S_1,\alpha\circ I_1}(\alpha(v)) \cup |\,\phi_2\,|_{S_1,\alpha\circ I_1}(\alpha(v)) \\
= & \quad \{ \text{ definition of the interpretation function } \} \\
 & |\,\phi_1 \vee \phi_2\,|_{S_1,\alpha\circ I_1}(\alpha(v))
\end{array}$$

$(\phi_1 \wedge \phi_2)$:

$$\begin{array}{ll}
 & \alpha(|\,(\phi_1 \wedge \phi_2)\,|_{S_1,I_1}(v)) \\
= & \quad \{ \text{ definition of the interpretation function } \} \\
 & \alpha(|\,(\phi_1)\,|_{S_1,I_1}(v) \cap |\,\phi_2\,|_{S_1,I_1}(v)) \\
\subseteq & \quad \{ \ \alpha \text{ monotonic } \} \\
 & \alpha(|\,\phi_1\,|_{S_1,I_1}(v)) \cap \alpha(|\,\phi_2\,|_{S_1,I_1}(v)) \\
\subseteq & \quad \{ \text{ induction hypothesis: } \alpha(|\,\phi_i\,|_{S_1,I_1}(v)) \subseteq |\,\phi_i\,|_{S_1,\alpha\circ I_1}(\alpha(v)) \ \} \\
 & |\,\phi_1\,|_{S_1,\alpha\circ I_1}(\alpha(v)) \cap |\,\phi_2\,|_{S_1,\alpha\circ I_1}(\alpha(v)) \\
= & \quad \{ \text{ definition of the interpretation function } \} \\
 & |\,\phi_1 \wedge \phi_2\,|_{S_1,\alpha\circ I_1}(\alpha(v))
\end{array}$$

Preservation of finite disjunction and conjunction is straightforward (by induction).

Let us consider infinite disjunction and conjunction:

We need a way to express infimum by supremum: Let $(D, \subseteq)$ be a set complete partial order and T a set. Then

$$\inf T = (\sup d \in D : (\forall t \in T : d \subseteq t))\}$$

$$
\begin{array}{ll}
& y = (\sup d \in D : (\forall t \in T : d \subseteq t)) \\
\Leftrightarrow & \{ \text{ definition of the supremum } \} \\
& (\forall d \in D, (\forall t \in T : d \subseteq t) : d \subseteq y) \wedge (\forall z \in D, (\forall d \in D, (\forall t \in T : d \subseteq t) : d \subseteq z) : y \subseteq z) \\
\Leftrightarrow & \{ \text{ set theory } \} \\
& (\forall d \in D, (\forall t \in T : d \subseteq t) : d \subseteq y) \wedge (\forall t \in T : y \subseteq t) \\
\Leftrightarrow & \{ \text{ commutativity of } \wedge \text{, renaming of bound variables } \} \\
& (\forall t \in T : y \subseteq t) \wedge (\forall z \in D, (\forall t \in T : z \subseteq t) : z \subseteq y) \\
\Leftrightarrow & \{ \text{ definition of infimum } \} \\
& y = \inf T
\end{array}
$$

In Fig. 4.1 we have defined the semantics of disjunction, conjunction and quantification as follows:

$$
\begin{array}{lcll}
(A,R),C,v & \models & (\wedge i : i \in T : \phi_i) & \text{iff } (A,R),C,v \models \phi_i \text{ for all } i \in T \\
(A,R),C,v & \models & (\vee i : i \in T : \phi_i) & \text{iff } (A,R),C,v \models \phi_i \text{ for some } i \in T \\
(A,R),C,v & \models & (\forall x \in T : \phi) & \text{iff } (A,R),C,x := t+v \models \phi \text{ for all } t \in T \\
(A,R),C,v & \models & (\exists x \in T : \phi) & \text{iff } (A,R),C,x := t+v \models \phi \text{ for some } t \in T
\end{array}
$$

We use in the proofs a different notation, which allows to manipulate sets:

$$
\begin{array}{l}
(A,R),C,v \models (\wedge i : i \in T : \phi_i) \\
\quad \text{iff } C \in (\inf u \in T : \{C' \mid (A,R),C',v \models (\wedge i : i \in T, \min(T) \le i \le u : \phi_i)\}) \\
(A,R),C,v \models (\vee i : i \in T : \phi_i) \\
\quad \text{iff } C \in (\sup u \in T : \{C' \mid (A,R),C',v \models (\vee i : i \in T, \min(T) \le i \le u : \phi_i)\}) \\
(A,R),C,v \models (\forall x \in T : \phi) \\
\quad \text{iff } C \in (\inf u \in T : \{C' \mid (A,R),C',x := t+v \models \phi \text{ for all } t, \min(T) \le t \le u\}) \\
(A,R),C,v \models (\exists x \in T : \phi) \\
\quad \text{iff } C \in (\sup u \in T : \{C' \mid (A,R),C',x := t+v \models \phi \text{ for some } t, \min(T) \le t \le u\})
\end{array}
$$

We show the equivalence of the two notions for $(\wedge i : i \in T : \phi_i)$

$$
\begin{array}{ll}
& (A,R),C,v \models (\wedge i : i \in T : \phi_i) \\
\Leftrightarrow & \{ \text{ definition of interpretation } \} \\
& (A,R),C,v \models \phi_i \text{ for all } i \in T \\
\Leftrightarrow & \{ \text{ set theory } \} \\
& C \in \mid \phi_i \mid_{(A,R),I} (v) \text{ for all } i \in T \\
\Leftrightarrow & \{ \text{ set theory } \} \\
& C \in S, S \subseteq \mid \phi_i \mid_{(A,R),I} (v) \text{ for all } i \in T \text{ for some } S \in A \\
\Leftrightarrow & \{ \text{ set theory } \} \\
& C \in (\sup S \in A, S \subseteq \mid \phi_i \mid_{(A,R),I} (v) \text{ for all } i \in T : S) \\
\Leftrightarrow & \{ \text{ set theory } \} \\
& C \in (\sup S \in A, S \subseteq \mid (\wedge i : i \in T, \min(T) \le i \le u : \phi_i) \mid_{(A,R),I} (v) \text{ for all } u \in T : S) \\
\Leftrightarrow & \{ \text{ rephrasing } \} \\
& C \in (\sup S \in A : (\forall u \in T : S \subseteq \mid \wedge i : i \in T, \min(T) \le i \le u : \phi_i \mid_{(A,R),I} (v)) \\
\Leftrightarrow & \{ \text{ definition infimum } \} \\
& C \in (\inf u \in T : \{C' \mid (A,R),C',v \models (\wedge i : i \in T, \min(T) \le i \le u : \phi_i)\})
\end{array}
$$

Let us continue:

$(\wedge i : i \in R : \phi_i)$

$$
\begin{array}{ll}
& \alpha(|(\wedge i : i \in R : \phi_i)|_{S_1,I_1}\ (v)) \\
= & \quad \{\ \text{definition of the interpretation function}\ \} \\
& \alpha(\inf u \in T : \{C' \mid (A,R), C', v \models (\wedge i : i \in T, \min(T) \leq i \leq u : \phi_i)\}) \\
= & \quad \{\ \text{definition of } I_1\ \} \\
& \alpha(\inf u \in T : |(\wedge i : i \in T, \min(T) \leq i \leq u : \phi_i)|_{(A_1,R_1),I_1}\ v) \\
= & \quad \{\ \text{definition of infimum}\ \} \\
& \alpha(\sup S \in \wp(A_1), (\forall u \in R : S \subseteq |(\wedge i : i \in T, \min(T) \leq i \leq u : \phi_i)|_{(A_1,R_1),I_1}\ v) : S) \\
= & \left\{ \begin{array}{l}
\text{For each set } Q \text{ is } (\wp(Q), \subseteq) \text{ a set complete partial order.} \\
\text{Let } (\wp(A_1), \subseteq) \text{ and } (\wp(A_2), \subseteq) \text{ be two set complete partial orders and} \\
\alpha : \wp(A_1) \to \wp(A_2) \text{ be continous, i.e., for each nonempty chain } T, \\
\alpha(\sup_{A_1} T) = \sup_{A_2} \alpha(T) =_{\text{def}} \sup_{A_2}\{f(t) \mid t \in T\} \\
(\wedge i : \min(T) \leq i \leq u : \phi_i) \text{ with } u \in T \text{, is a non-empty chain, since } T \\
\text{non-empty}
\end{array} \right\} \\
& (\sup S \in \wp(A_1), (\forall u \in T : S \subseteq |(\wedge i : i \in T, \min(T) \leq i \leq u : \phi_i|_{(A_1,R_1),I_1}\ v)) : \alpha(S)) \\
= & \quad \{\ \text{monotonicity of } \alpha\ \} \\
& (\sup S \in \wp(A_1), (\forall u \in T : \alpha(S) \subseteq \alpha(|(\wedge i : i \in T, \min(T) \leq i \leq u : \phi_i)|_{(A_1,R_1),I_1}\ v)) : \alpha(S)) \\
\subseteq & \quad \{\ \text{induction hypothesis---finite conjunction}\ \} \\
& (\sup S \in \wp(A_1), (\forall u \in T : \alpha(S) \subseteq |(\wedge i : i \in T, \min(T) \leq i \leq u : \phi_i)|_{(A_2,R_2),\alpha \circ I_1}\ \alpha(v)) : \alpha(S)) \\
= & \quad \{\ \text{definition infimum}\ \} \\
& (\inf u \in T : |(\wedge i : i \in T, \min(T) \leq i \leq u : \phi_i)|_{(A_2,R_2),\alpha \circ I_1}\ \alpha(v)) \\
= & \quad \{\ \text{definition of the interpretation function}\ \} \\
& |(\vee i : i \in T : \phi_i)|_{S_2,\alpha \circ I_1}\ (\alpha(v))
\end{array}
$$

$(\vee i : i \in T : \phi_i)$:

$$
\begin{array}{ll}
& \alpha(|(\vee i : i \in T : \phi_i)|_{S_1,I_1}\ (v)) \\
= & \quad \{\ \text{definition of the interpretation function}\ \} \\
& \alpha(\sup u \in T : \{C' \mid (A,R), C', v \models (\vee i : i \in T, \min(T) \leq i \leq u : \phi_i)\}) \\
\Leftrightarrow & \quad \{\ \text{Definition of } I_1\ \} \\
& \alpha(\sup u \in T : |(\vee i : i \in T, \min(T) \leq i \leq u : \phi_i)|_{(A_1,R_1),I_1}\ v) \\
= & \left\{ \begin{array}{l}
\text{For each set } Q \text{ is } (\wp(Q), \subseteq) \text{ a set complete partial order.} \\
(\wp(A_1), \subseteq) \text{ and } (\wp(A_2), \subseteq) \text{ be two set complete partial orders and } \alpha : \\
\wp(A_1) \to \wp(A_2) \text{ be continous, i.e., for each nonempty chain } T, \\
\alpha(\sup u \in T : |(\wedge i : i \in T, \min(T) \leq i \leq u : \phi_i)|_{(A_1,R_1),I_1}\ v\}).
\end{array} \right\} \\
& (\sup u \in T : \alpha(|\ (\vee i : i \in T, \min(T) \leq i \leq u : \phi_i)\ |_{(A_1,R_1),I_1}\ v)) \\
\subseteq & \quad \{\ \text{Induction hypothesis---finite disjunction}\ \} \\
& (\sup u \in T : |\ (\vee i : i \in T, \min(T) \leq i \leq u : \phi_i)\ |_{(A_2,R_2),\alpha \circ I_2}\ \alpha(v)) \\
= & \quad \{\ \text{definition of the interpretation function}\ \} \\
& |(\vee i : i \in T : \phi_i)|_{S_1,\alpha \circ I_1}\ (\alpha(v))
\end{array}
$$

$(\exists w : w \in R : \phi)$ and $(\forall w : w \in R : \phi)$: The proofs follow infinite conjunction and disjunction.

$\langle L \rangle \phi$:

$$
\begin{array}{ll}
& \alpha(|\langle L \rangle \phi|_{S_1,I_1}(v)) \\
= & \quad \{ \text{ definition of } \langle L \rangle \phi \ \} \\
& \text{for all ground valuations } v+v'\text{: } \alpha(|\langle (v+v')(L) \rangle \phi|_{S_1,I_1}(v+v')) \\
= & \quad \{ \text{ definition of } \langle L \rangle \phi \ \} \\
& \text{for all ground valuations } v+v'\text{: } \alpha(\mathit{pre}(R_1)((v+v')(L))\ |\phi|_{S_1,I_1}(v+v')) \\
\subseteq & \quad \left\{ \ Id^{A_1} \subseteq \gamma \circ \alpha \ \right\} \\
& \text{for all ground valuations } v+v'\text{: } \alpha(\mathit{pre}(R_1)((v+v')(L))(\gamma(\alpha(|\phi|_{S_1,I_1}(v+v'))))) \\
\subseteq & \quad \{ \text{ definition simulation relation: } \alpha \circ \mathit{pre}(R_1)(L) \circ \gamma \subseteq \mathit{pre}(R_2)(\alpha(L)) \ \} \\
& \text{for all ground valuations } v+v'\text{: } \mathit{pre}(R_2)(\alpha((v+v')(L)))(\alpha(|\phi|_{S_1,I_1}(v+v'))) \\
\subseteq & \quad \{ \text{ induction hypothesis: } \alpha(|\phi|_{S_1,I_1}) \subseteq |\phi|_{S_1,\alpha\circ I_1} \ \} \\
& \text{for all ground valuations } v+v'\text{: } \mathit{pre}(R_2)(\alpha((v+v')(L)))(|\phi|_{S_1,\alpha\circ I_1}(\alpha(v+v'))) \\
= & \quad \{ \text{ definition of } \langle L \rangle \phi \ \} \\
& \text{for all ground valuations } v+v'\text{: } |\langle \alpha(L) \rangle \phi|_{S_1,\alpha\circ I_1}(\alpha(v+v')) \\
= & \quad \{ \text{ efinition of } \langle L \rangle \phi \ \} \\
& |\langle \alpha(L) \rangle \phi|_{S_1,\alpha\circ I_1}(\alpha(v))
\end{array}
$$

$$
\begin{array}{ll}
& \alpha(|\neg\phi|_{S_1,I_1}(v)) \\
\Leftrightarrow & \quad \{ \text{ interpretation of negation } \} \\
& \alpha(A_1 \setminus |\phi|_{S_1,I_1}(v)) \\
= & \quad \{ \text{ set theory } \} \\
& \alpha(\{C | C \in A_1, C \notin |\phi|_{S_1,I_1}(v)\}) \\
= & \quad \{ \ \alpha \text{ distributes over } \cup \ \} \\
& \{D | D \in \alpha(C), C \in A_1, C \notin |\phi|_{S_1,I_1}(v)\} \\
= & \quad \{ \text{ consistency } \} \\
& \{D | D \in \alpha(A_1), D \in \alpha(C), C \in A_1, \alpha(C) \not\subseteq \alpha(|\phi|_{S_1,I_1}(v))\} \\
= & \quad \{ \text{ set theory } \} \\
& \{D | D \in A_2, D \in \alpha(C), C \in A_1, \alpha(C) \not\subseteq \alpha(|\phi|_{S_1,I_1}(v))\} \\
= & \quad \{ \text{ consistency of the interpretation with } \alpha \ \} \\
& \{D | D \in A_2, D \in \alpha(C), C \in A_1, \alpha(C) \not\subseteq |\phi|_{S_2,\alpha(I_1)}(\alpha(v))\} \\
= & \quad \{ \text{ set theory } \} \\
& A_2 \setminus |\phi|_{S_2,\alpha(I_1)}(\alpha(v)) \\
= & \quad \{ \text{ interpretation of negation } \} \\
& |\neg\phi|_{S_2,\alpha(I_1)}(\alpha(v))
\end{array}
$$

$(\mu X.\phi)$:

$$
\begin{array}{ll}
& \alpha(|\mu X.\phi|_{S_1,I_1}(v)) \\
= & \quad \{ \text{ definition of } \mu X.\phi \ \} \\
& \alpha(\cap P_1 : P_1 \subseteq A_1, |\phi|_{S_1,I_1}[P_1/X](v) \subseteq P_1 : P_1) \\
\subseteq & \quad \{ \ P_1 = \gamma(P_2) \ \} \\
& \alpha(\cap P_2 : P_2 \subseteq A_2, |\phi|_{S_1,I_1}[\gamma(P_2)/X](v) \subseteq \gamma(P_2) : \gamma(P_2)) \\
\subseteq & \quad \{ \ (*) \text{ see below } \}
\end{array}
$$

$$
\begin{array}{ll}
 & (\cap P_2 : P_2 \subseteq A_2, |\phi|_{S_2,\alpha\circ I_1}\,[P_2/X](\alpha(v)) \subseteq P_2 : P_2) \\
= & \quad \{ \text{ definition of } \mu X.\phi \ \} \\
 & |\,\mu X.\phi|_{S_2,\alpha\circ I_1}\,(\alpha(v))
\end{array}
$$

We prove (*):

Proof.

$$
\begin{array}{ll}
 & |\,\phi|_{S_2,\alpha\circ I_1}\,[P_2/X](\alpha(v)) \subseteq P_2 \\
\Rightarrow & \quad \{ \text{ monotonicity of } \gamma \ \} \\
 & \gamma(|\,\phi|_{S_2,\alpha\circ I_1}\,[P_2/X](\alpha(v))) \subseteq \gamma(P_2) \\
\Rightarrow & \quad \left\{ \text{ monotonicity of } \phi,\ \alpha\circ\gamma \subseteq Id^{A_2} \ \right\} \\
 & \gamma(|\,\phi|_{S_2,\alpha\circ I_1}\,[\alpha(\gamma(P_2))/X](\alpha(v))) \subseteq \gamma(P_2) \\
\Rightarrow & \quad \left\{ \begin{array}{l} \text{induction hypothesis:} \\ |\,\phi\,|_{S_1,I_1}\,[\gamma(P_2)/X](v) \subseteq \gamma(|\,\phi|_{S_1,\alpha\circ I_1}\,[\alpha(\gamma(P_2))/X](\alpha(v))) \end{array} \right\} \\
 & |\,\phi|_{S_1,I_1}\,[\gamma(P_2)/X](v) \subseteq \gamma(P_2)
\end{array}
$$

and thus

$$
\begin{array}{ll}
 & (\cap P_2 : P_2 \in A_2, |\,\phi|_{S_1,I_1}\,[\gamma(P_2)/X](v) \subseteq \gamma(P_2) : P_2) \\
\subseteq & (\cap P_2 : P_2 \in A_2, |\,\phi|_{S_2,\alpha\circ I_1}\,[P_2/X](\alpha(v)) \subseteq P_2 : P_2)
\end{array}
$$

□

$\nu X.\phi$:

$$
\begin{array}{ll}
 & \alpha(|\,\nu X.\phi|_{S_1,I_1}\,(v)) \\
= & \quad \{ \text{ definition of } \nu X.\phi \ \} \\
 & \alpha\overline{(\cap P_1 : P_1 \subseteq A_1, |\neg\phi|_{S_1,I_1}\,[P_1/\neg X](v) \subseteq P_1 : P_1)} \\
= & \quad \{ \ \neg(a \cap b) = \neg a \cup \neg b \ \} \\
 & \alpha(\cup P_1 : P_1 \subseteq A_1, |\neg\phi|_{S_1,I_1}\,[P_1/\neg X](v) \not\subseteq P_1 : P_1) \\
\subseteq & \quad \{ \ P_1 = \gamma(P_2) \ \} \\
 & \alpha(\cup P_2 : P_2 \subseteq A_2, |\neg\phi|_{S_1,I_1}\,[\gamma(P_2)/\neg X](v) \not\subseteq \gamma(P_2) : \gamma(P_2)) \\
\subseteq & \quad \{ \text{ (*) See below } \} \\
 & (\cup P_2 : P_2 \subseteq A_2, |\neg\phi|_{S_2,\alpha\circ I_1}\,[P_2/\neg X](\alpha(v)) \not\subseteq P_2 : P_2) \\
= & \quad \{ \ \neg a \cup \neg b = \neg(a \cap b) \ \} \\
 & \overline{(\cap P_2 : P_2 \subseteq A_2, |\neg\phi|_{S_2,\alpha\circ I_1}\,[P_2/\neg X](\alpha(v)) \subseteq P_2 : P_2)} \\
= & \quad \{ \text{ definition of } \nu X.\phi \ \} \\
 & |\,\nu X.\phi|_{S_2,\alpha\circ I_1}\,(\alpha(v))
\end{array}
$$

We prove (*):

Proof. $|\neg\phi|_{S_2,\alpha\circ I_1}[P_2/\neg X](\alpha(v)) \not\subseteq P_2$
$\Rightarrow$ $\{$ monotonicity of γ $\}$
$\gamma(|\neg\phi|_{S_2,\alpha\circ I_1}[P_2/\neg X](\alpha(v))) \not\subseteq \gamma(P_2)$
$\Rightarrow$ $\left\{ \text{monotonicity of } \phi,\ \alpha\circ\gamma \subseteq Id^{A_2} \right\}$
$\gamma(|\neg\phi|_{S_2,\alpha\circ I_1}[\alpha(\gamma(P_2))/\neg X](\alpha(v))) \not\subseteq \gamma(P_2)$
$\Rightarrow$ $\left\{ \begin{array}{l}\text{induction hypothesis:}\\ |\neg\phi|_{S_1,I_1}[\gamma(P_2)/\neg X](v) \not\subseteq \gamma(|\neg\phi|_{S_1,\alpha\circ I_1}[\alpha(\gamma(P_2))/\neg X](\alpha(v)))\end{array} \right\}$
$|\phi|_{S_1,I_1}[\gamma(P_2)/\neg X](v) \not\subseteq \gamma(P_2)$

and thus

$$\begin{array}{ll} & (\cup P_2 : P_2 \in A_2, |\neg\phi|_{S_1,I_1}[\gamma(P_2)/\neg X](v) \not\subseteq \gamma(P_2) : P_2) \\ \subseteq & (\cup P_2 : P_2 \in A_2, |\neg\phi|_{S_2,\alpha\circ I_1}[P_2/\neg X](\alpha(v)) \not\subseteq P_2 : P_2) \end{array}$$

□

Case 2: To prove $\widetilde{\gamma}$ preserves ϕ for I_2

$\phi_1 \wedge \phi_2$:

$$\begin{array}{ll} & \widetilde{\gamma}(|\phi_1 \wedge \phi_2|_{S_2,I_2}(v)) \\ = & \{ \text{ definition of the interpretation function } \} \\ & \widetilde{\gamma}(|\phi_1|_{S_2,I_2}(v) \cap |\phi_2|_{S_2,I_2}(v)) \\ = & \{ \ \widetilde{\gamma} \text{ distributes over } \cap \ \} \\ & \widetilde{\gamma}(|\phi_1|_{S_2,I_2}(v)) \cap \widetilde{\gamma}(|\phi_2|_{S_2,I_2}(v)) \\ = & \left\{ \text{ induction hypothesis: } \widetilde{\gamma}(|\phi_1|_{S_2,I_2}(v)) \subseteq |\phi_1|_{S_1,\widetilde{\gamma}\circ I_2}(\widetilde{\gamma}(v)) \right\} \\ & |\phi_1|_{S_1,\widetilde{\gamma}\circ I_2}(\widetilde{\gamma}(v)) \cap |\phi_2|_{S_1,\widetilde{\gamma}\circ I_2}(\widetilde{\gamma}(v)) \\ = & \{ \text{ definition of the interpretation function } \} \\ & |\phi_1 \wedge \phi_2|_{S_1,\widetilde{\gamma}\circ I_2}(\widetilde{\gamma}(v)) \end{array}$$

$[L]\phi_1$:

$$\begin{array}{ll} & \widetilde{\gamma}(|[L]\phi|_{S_2,I_2}(v)) \\ = & \{ \text{ definition interpretation } \} \\ & \text{for all ground valuations } v+v'\text{: } \widetilde{\gamma}(|[L]\phi|_{S_2,I_2}(v+v')) \\ = & \{ \text{ definition of interpretation } \} \\ & \text{for all ground valuations } v+v'\text{: } \widetilde{\gamma}(\widetilde{pre}(R_2)((v+v')(L))(|\phi|_{S_2,I_2})(v+v')) \\ \subseteq & \left\{ Id^{A_2} \subseteq \widetilde{\alpha}\circ\widetilde{\gamma} \text{ (see Lemma 4.8) } \right\} \\ & \text{for all ground valuations } v+v'\text{: } \widetilde{\gamma}(\widetilde{pre}(R_2)((v+v')(L))(\widetilde{\alpha}(\widetilde{\gamma}(|\phi|_{S_2,I_2}(v+v'))))) \\ \subseteq & \{ \text{ Lemma B.2 } \} \\ & \text{for all ground valuations } v+v'\text{: } \widetilde{pre}(R_1)((v+v')(L))(\widetilde{\gamma}(|\phi|_{S_2,I_2}(v+v'))) \\ \subseteq & \{ \text{ induction hypothesis } \} \\ & \text{for all ground valuations } v+v'\text{: } \widetilde{pre}(R_1)(L)(|\phi|_{S_1,\widetilde{\gamma}\circ I_2}(\widetilde{\gamma}(v+v')))) \\ \subseteq & \{ \text{ definition of } [L]\phi \ \} \\ & \text{for all ground valuations } v+v'\text{: } |[L]\phi|_{S_1,\widetilde{\gamma}\circ I_2}(\widetilde{\gamma}(v+v')))) \end{array}$$

$=$ { definition interpretation }
$\mid [L]\phi \mid_{S_1,\widetilde{\gamma}\circ I_2} (\widetilde{\gamma}(v))))$

Analogous to the preservation results for α are the cases $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, $(\exists v : v \in V : \phi(v))$ and $(\forall v : v \in V : \phi(v))$ ($\widetilde{\gamma}$ distributes over $\cup$ and $\widetilde{\gamma}(X \cap Y) \subseteq \widetilde{\gamma}(X) \cap \widetilde{\gamma}(Y)$ (see Lemma 4.8)).

□

B.3 Proof of *ME*

Lemma 4.26 *ME* holds for $(A_{\mathrm{Sen}}, R_{\mathrm{Sen}})$.

Proof.

$(A_{\mathrm{Sen}}, R_{\mathrm{Sen}}), C \models \mathit{ME}$ for all $C \in \mathrm{Sen}$
$\Leftrightarrow$ { definition of $\models$ }
ME holds for $(A_{\mathrm{Sen}}, R_{\mathrm{Sen}})$

We prove the mutual exclusion property for arbitrary `R`. For clarification, we depict the transitions in Fig. B.1.

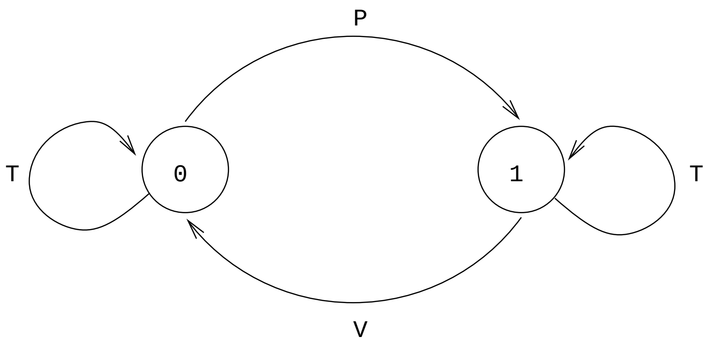

Figure B.1: State changes of a sentinel

We proceed as follows: we prove that *ME* holds by induction for $\alpha = \omega$. This is sufficient since we show in a second step that ME^n stabilizes for $n = 1$.

Proof by nested induction over the number of iterations. To prove, according to Prop. 4.1,

$$C \in \mid \mathit{ME}^n \mid_{I(\mathtt{SENTINEL}), I_{\mathrm{Sen}}}$$

for all n and all $C \in I(\mathtt{SENTINEL})$.
Induction base $n = 0$:

$$
\begin{array}{l}
\quad I(\texttt{SENTINEL}) \models ME^0 \\
\Leftarrow \quad \left\{ ME^0 = |\, \texttt{true} \,|_{I(\texttt{SENTINEL}), I_{\mathrm{Sen}}} \right\} \\
\quad C \in I(\texttt{SENTINEL})
\end{array}
$$

Induction step $n \to n+1$:
Case 1: $\texttt{< R : Sentinel | status = 0 >} \notin C \ \surd$
Case 2: $\texttt{< R : Sentinel | status = 0 >} \in C$

$$
\begin{array}{l}
\quad I(\texttt{SENTINEL}), C \models ME^{n+1} \\
\Leftarrow \quad \{ \text{ definition of } ME^{n+1} \ \} \\
\quad I(\texttt{SENTINEL}) \models \ \text{“}\texttt{< R : Sentinel | status = 0 >}\text{”} \Rightarrow \\
\qquad\qquad\qquad\qquad [(\texttt{V(R)})] \text{ false} \\
\qquad\qquad\qquad\quad \wedge \ [T_{\texttt{Message}}(\Sigma_{\mathrm{Sen}}) \backslash \{(\texttt{V(R)}), (\texttt{P(R)})\}] ME^n \\
\qquad\qquad\qquad\quad \wedge \ [(\texttt{P(R)})] \ (\nu X_2. \\
\qquad\qquad\qquad\qquad\qquad [(\texttt{P(R)})] \ \texttt{false} \\
\qquad\qquad\qquad\qquad\quad \wedge \ [T_{\texttt{Message}}(\Sigma_{\mathrm{Sen}}) \backslash \{(\texttt{V(R)}), (\texttt{P(R)})\}] \ X_2 \\
\qquad\qquad\qquad\qquad\quad \wedge \ [(\texttt{V(R)})] \ ME^n)
\end{array}
$$

$$
\Leftarrow \quad \left\{
\begin{array}{lcl}
I(\texttt{SENTINEL}), C & \models & [(\texttt{V(R)})] \ \texttt{false} \\
I(\texttt{SENTINEL}), C & \models & [(\texttt{T(R)})] \ \texttt{true} \\
I(\texttt{SENTINEL}), C & \models & [(\texttt{T(R)})] \ (\nu X_2. \\
 & & \qquad [(\texttt{P(R)})] \ \texttt{false} \\
 & & \quad \wedge \ [T_{\texttt{Message}}(\Sigma_{\mathrm{Sen}}) \backslash \{(\texttt{V(R)}), (\texttt{P(R)})\}] \ X_2 \\
 & & \quad \wedge \ [(\texttt{V(R)})] \ ME^n,) \\
(*) \text{ see below} & &
\end{array}
\right\}
$$

$$
\begin{array}{l}
\quad
\begin{array}{lcl}
post(R_{\mathrm{Sen}})(\texttt{V(R)})(C) & = & \emptyset \\
post(R_{\mathrm{Sen}})(\texttt{T(R)})(C) & = & C' \ \texttt{< R : Sentinel | status = 0 >} \\
post(R_{\mathrm{Sen}})(\texttt{P(R)})(C) & = & C \ \texttt{< R : Sentinel | status = 1 >}
\end{array} \\
\Leftarrow \quad \{ \text{ characterization of initial model } \} \\
\quad I(\texttt{SENTINEL}) \text{ is the initial model of } \texttt{SENTINEL}; \ \texttt{< R : Sentinel | status = 0 >} \in C
\end{array}
$$

To prove $(*)$ by induction on the number m of iterations.

$$I(\texttt{SENTINEL}), D \models H$$

where

$$
\begin{array}{ll}
H =_{\mathrm{def}} (\nu X_2. & \quad [(\texttt{P(R)})] \ \texttt{false} \\
 & \wedge \ [T_{\texttt{Message}}(\Sigma_{\mathrm{Sen}}) \backslash \{(\texttt{V(R)}), (\texttt{P(R)})\}] \ X_2 \\
 & \wedge \ [(\texttt{V(R)})] \ ME^n)
\end{array}
$$

for all $D \in \{C \mid \texttt{< R : Sentinel | status = 1 >} \in C\}$,
$D = \texttt{< R : Sentinel | status = 1 >} \ D'$

Proof. Induction base $m = 0$:

$$
\begin{array}{l}
\quad I(\texttt{SENTINEL}), D \models H^0 \\
\Leftarrow \quad \{ \text{ definition of } H^0 \ \} \\
\quad I(\texttt{SENTINEL}), D \models \mid \texttt{true} \mid_{I(\texttt{SENTINEL}), I_{\text{Sen}}}
\end{array}
$$

Induction base $m = 1$:

$$
\begin{array}{l}
\quad I(\texttt{SENTINEL}), D \models H^1 \\
\Leftarrow \quad \{ \text{ definition of } H^1 \ \} \\
\quad I(\texttt{SENTINEL}), D \models \quad [\texttt{(P(R))}]\ \texttt{false} \\
\qquad\qquad\qquad\qquad \wedge \ [\texttt{T(V)}] H^0 \\
\qquad\qquad\qquad\qquad \wedge \ [\texttt{(V(R))}]\ ME^n \\
\Leftarrow \quad \left\{ \begin{array}{l}
\begin{array}{lcl}
post(R_{\text{Sen}})(\texttt{P(R)})(D) & = & \emptyset \\
post(R_{\text{Sen}})(\texttt{T(R)})(D) & = & D' \texttt{ < R : Sentinel | status = 1 >} \\
post(R_{\text{Sen}})(\texttt{V(R)})(D) & = & D' \texttt{ < R : Sentinel | status = 0 >} \} \\
post(R_{\text{Sen}})(\texttt{X(Y)})(D) & = & D' \texttt{ < R : Sentinel | status = 1 >} \\
 & & \texttt{for some } D\text{''}, \text{for all } \texttt{Y} \neq \texttt{R} \text{ and } \texttt{P} \in \texttt{P}, \texttt{V}, \texttt{T}
\end{array} \\
\{ C \mid \texttt{< R : Status | status = 0 >} \in C \} \subseteq \mid \texttt{true} \mid_{I(\texttt{SENTINEL}), I_{\text{Sen}}}, \\
\text{monotonicity of } \mu\text{-calculus formulas}
\end{array} \right\} \\
\quad I(\texttt{SENTINEL}) \text{ is the initial model of } \texttt{SENTINEL}, \\
\texttt{< R : Sentinel | status = 1 >} \in D
\end{array}
$$

Induction step $m \to m + 1$:

$$
\begin{array}{l}
\quad I(\texttt{SENTINEL}), D \models H^{m+1} \\
\Leftarrow \quad \{ \text{ definition of } H^{m+1} \ \} \\
\quad I(\texttt{SENTINEL}), D \models \quad [\texttt{(P(R))}]\ \texttt{false} \\
\qquad\qquad\qquad\qquad \wedge \ [\texttt{T(V)}] H^m \\
\qquad\qquad\qquad\qquad \wedge \ [\texttt{(V(R))}]\ ME^n \\
\Leftarrow \quad \left\{ \begin{array}{l}
\begin{array}{lcl}
post(R_{\text{Sen}})(\texttt{P(R)})(D) & = & \emptyset \\
post(R_{\text{Sen}})(\texttt{T(R)})(D) & = & D' \texttt{ < R : Sentinel | status = 1 >} \\
post(R_{\text{Sen}})(\texttt{V(R)})(D) & = & D' \texttt{ < R : Sentinel | status = 0 >} \\
post(R_{\text{Sen}})(\texttt{X(Y)})(D) & = & D' \texttt{ < R : Sentinel | status = 1 >} \\
 & & \text{for all } \texttt{Y} \neq \texttt{R} \text{ and all } \texttt{X} \in \texttt{P}, \texttt{V}, \texttt{T}
\end{array} \\
D' \texttt{< R : Status | status = 0 >} \in \mid \texttt{true} \mid_{I(\texttt{SENTINEL}), I_{\text{Sen}}}, \\
\text{monotonicity of } \mu\text{-calculus formulas}, \\
\text{induction hypothesis}
\end{array} \right\} \\
\quad D \in \mid H^m \mid_{I(\texttt{SENTINEL}), I_{\text{Sen}}}
\end{array}
$$

□

We show that $I_{\text{Sen}}(ME^1) = I_{\text{Sen}}(ME^0) = A_{\text{Sen}}$

$$
\begin{array}{ll}
 & I_{\mathrm{Sen}}(ME^1) \\
= & \quad \{ \text{ definition of } ME^1 \text{ } \} \\
 & I(\forall \mathtt{R} \in T_{\mathrm{ObjectId}}(\Sigma): \\
 & \text{“}\mathtt{< R : Sentinel \mid status = 0 >}\text{”} \Rightarrow \\
 & \qquad [(\mathtt{V(R)})]\ \mathtt{false} \\
 & \quad \wedge \ \ [T_{\mathtt{Message}}(\Sigma_{\mathrm{Sen}}) \backslash \{(\mathtt{V(R)}), (\mathtt{P(R)})\}]\ ME^0 \\
 & \quad \wedge \ \ [(\mathtt{P(R)})]\ (\nu X_2. \\
 & \qquad\qquad [(\mathtt{P(R)})]\ \mathtt{false} \\
 & \qquad\quad \wedge \ \ [T_{\mathtt{Message}}(\Sigma_{\mathrm{Sen}}) \backslash \{(\mathtt{V(R)}), (\mathtt{P(R)})\}]\ X_2 \\
 & \qquad\quad \wedge \ \ [(\mathtt{V(R)})]\ ME^0)) \\
= & \left\{ \begin{array}{l} \text{definition of the interpretation function,} \\ I_{\mathrm{Sen}}(ME^0) = A_{\mathrm{Sen}}, \\ \text{we omit the suffix “for any } \mathtt{R} \in A_{\mathrm{Sen}_{\mathrm{ObjectId}}} \text{” until it becomes} \\ \text{relevant} \end{array} \right\} \\
 & \{C \mid C \in A_{\mathrm{Sen}}, \mathtt{< R : Sentinel \mid status = 0 >} \notin C\} \cup \\
 & \qquad \widetilde{pre}(R_{\mathtt{Sen}})((\mathtt{V(R)}))I_{\mathtt{Sen}}(\mathtt{false}) \\
 & \quad \cap \ \ \widetilde{pre}(R_{\mathtt{Sen}})(T_{\mathtt{Message}}(\Sigma_{\mathrm{Sen}}) \backslash \{(\mathtt{V(R)}), (\mathtt{P(R)})\})A_{\mathrm{Sen}} \\
 & \quad \cap \ \ \widetilde{pre}(R_{\mathtt{Sen}})((\mathtt{P(R)}))(I_{\mathrm{Sen}}(\nu X_2. \\
 & \qquad\qquad [(\mathtt{P(R)})]\ \mathtt{false} \\
 & \qquad\quad \wedge \ \ [T_{\mathtt{Message}}(\Sigma_{\mathrm{Sen}}) \backslash \{(\mathtt{V(R)}), (\mathtt{P(R)})\}]\ X_2 \\
 & \qquad\quad \wedge \ \ [(\mathtt{V(R)})]\ ME^0)) \\
= & \quad \{ \text{ definition of the dual } \} \\
 & \{C \mid C \in A_{\mathrm{Sen}}, \mathtt{< R : Sentinel \mid status = 0 >} \notin C\} \cup \\
 & \quad A_{\mathrm{Sen}} \setminus (\qquad pre(R_{\mathtt{Sen}})((\mathtt{V(R)}))(A_{\mathrm{Sen}}) \\
 & \qquad\qquad \cup \ \ pre(R_{\mathtt{Sen}})(T_{\mathtt{Message}}(\Sigma_{\mathrm{Sen}}) \backslash \{(\mathtt{V(R)}), (\mathtt{P(R)})\})(\emptyset) \\
 & \qquad\qquad \cup \ \ pre(R_{\mathtt{Sen}})((\mathtt{P(R)}))\ (I_{\mathrm{Sen}}(\nu X_2. \\
 & \qquad\qquad\qquad\quad [(\mathtt{P(R)})]\ \mathtt{false} \\
 & \qquad\qquad\qquad \wedge \ \ [T_{\mathtt{Message}}(\Sigma_{\mathrm{Sen}}) \backslash \{(\mathtt{V(R)}), (\mathtt{P(R)})\}]\ X_2 \\
 & \qquad\qquad\qquad \wedge \ \ [(\mathtt{V(R)})]\ ME^0)) \\
\supseteq & \left\{ \begin{array}{l} \qquad pre(R_{\mathtt{Sen}})((\mathtt{V(R)}))I_{\mathtt{Sen}}(A_{\mathrm{Sen}}) \\ \quad \subseteq \ \ \{C \mid C \in A_{\mathrm{Sen}}, \mathtt{< R : Sentinel \mid status = 1 >} \in C\} \\ pre(R_{\mathtt{Sen}})(L)(\emptyset) = \emptyset \end{array} \right\} \\
 & \{C \mid C \in A_{\mathrm{Sen}}, \mathtt{< R : Sentinel \mid status = 0 >} \notin C\} \cup \\
 & \quad A_{\mathrm{Sen}} \setminus (\qquad \{C \mid C \in A_{\mathrm{Sen}}, \mathtt{< R : Sentinel \mid status = 1 >} \in C\} \\
 & \qquad\qquad \cup \ \ \emptyset \\
 & \qquad\qquad \cup \ \ pre(R_{\mathrm{Sen}})(\mathtt{P(V)})(A_{\mathrm{Sen}}(I_{\mathrm{Sen}}(\nu X_2. \\
 & \qquad\qquad\qquad\quad [(\mathtt{P(R)})]\ \mathtt{false} \\
 & \qquad\qquad\qquad \wedge \ \ [T_{\mathtt{Message}}(\Sigma_{\mathrm{Sen}}) \backslash \{(\mathtt{V(R)}), (\mathtt{P(R)})\}]\ X_2 \\
 & \qquad\qquad\qquad \wedge \ \ [(\mathtt{V(R)})]\ ME^0)))
\end{array}
$$

$$
\begin{array}{ll}
\supseteq & \left\{ \begin{array}{ll} & \{C \mid C \in A_{\text{Sen}}, \texttt{< R : Sentinel | status = 1 >} \in C\} \\ \subseteq & I_{\text{Sen}}(\nu X_2. \\ & \quad [\texttt{(P(R))}]\ \texttt{false} \\ & \wedge\ [T_{\texttt{Message}}(\Sigma_{\text{Sen}}) \backslash \{\texttt{(V(R))}, \texttt{(P(R))}\}]\ X_2 \\ & \wedge\ [\texttt{(V(R))}]\ ME^0) \end{array} \right\} \\
& \{C \mid C \in A_{\text{Sen}}, \texttt{< R : Sentinel | status = 0 >} \notin C\} \cup \\
& \quad A_{\text{Sen}} \setminus (\quad \{C \mid C \in A_{\text{Sen}}, \texttt{< R : Sentinel | status = 1 >} \in C\} \\
& \qquad \cup\ \emptyset \\
& \qquad \cup\ \mathit{pre}(R_{\text{Sen}})\texttt{(P(V))}\overline{\{C \mid \texttt{< R : Sentinel | status = 1 >} \in C\}} \\
= & \quad \{ \text{ set theory } \} \\
& \{C \mid C \in A_{\text{Sen}}, \texttt{< R : Sentinel | status = 0 >} \notin C\} \cup \\
& \quad A_{\text{Sen}} \setminus (\quad \{C \mid C \in A_{\text{Sen}}, \texttt{< R : Sentinel | status = 1 >} \in C\} \\
& \qquad \cup\ \mathit{pre}(R_{\text{Sen}})\texttt{(P(V))}(\{C \mid \texttt{< R : Sentinel | status = 1 >} \notin C\}) \\
= & \quad \{ \text{ set theory } \} \\
& \{C \mid C \in A_{\text{Sen}}, \texttt{< R : Sentinel | status = 0 >} \notin C\} \cup \\
& \quad A_{\text{Sen}} \setminus (\quad \{C \mid C \in A_{\text{Sen}}, \texttt{< R : Sentinel | status = 1 >} \in C\} \\
& \qquad \cup\ \emptyset) \\
= & \quad \{ \text{ set theory } \} \\
& A_{\text{Sen}} \text{ for any } \texttt{R} \in A_{\text{Sen}_{\text{ObjectId}}} \\
= & \quad \{ \text{ set theory } \} \\
& A_{\text{Sen}}
\end{array}
$$

Since $I_{\text{Sen}} ME^1 \supseteq A_{\text{Sen}}$ implies already $I_{\text{Sen}} ME^1 = A_{\text{Sen}}$ stabilizes ME^n and ME^1 is the fixpoint.

$$I_{\text{Sen}}(ME^0) = I_{\text{Sen}}(ME^1) = \ldots = I_{\text{Sen}}(ME^n) = I_{\text{Sen}}(ME^\omega) = I_{\text{Sen}}(ME^{\omega+1})$$

$\square$

Appendix C

Refinement

C.1 Refinement of Classes

Demonstrandum 5.3
$Sp_M \rightsquigarrow$`BD-BUFFER-INTERN`.

Proof. Let us make two remarks beforehand:

- We consider only the initial model, which is finite-branching. The formulas are not nested fixpoint, thus the cardinality of the induction is ω.
- Specification `BD-BUFFER-INTERN` is coherent, thus the ground term algebra is (up to isomorphism) the initial model of `BD-BUFFER-INTERN`.

We define $I(\texttt{BD-BUFFER-INTERN}) = (A, R)$.

$$
\begin{array}{ll}
 & Sp_M \rightsquigarrow \texttt{BD-BUFFER-INTERN} \\
\Leftrightarrow & \{ \text{ Definition of } \rightsquigarrow, \text{(Def. 5.1) } \} \\
 & I(\texttt{BD-BUFFER-INTERN}) \models \quad \mathit{Persistence}(\texttt{B}) \\
 & \qquad\qquad\qquad\qquad\qquad \wedge \ \mathit{State}(\texttt{B}) \\
 & \qquad\qquad\qquad\qquad\qquad \wedge \ \mathit{Synchronization}(\texttt{B}) \\
 & \qquad\qquad\qquad\qquad\qquad \wedge \ \mathit{StateChange}(\texttt{B}) \\
 & \qquad\qquad\qquad\qquad\qquad \wedge \ \mathit{AnswerMessages}(\texttt{B}) \\
\Leftrightarrow & \{ \text{ semantics of } \wedge \ \} \\
 & \begin{array}{lll}
(A, R) \models \mathit{Persistence}(\texttt{B}) & & \text{for all } \texttt{B} \\
(A, R) \models \mathit{State}(\texttt{B}) & & \text{for all } \texttt{B} \\
(A, R) \models \mathit{Synchronization}(\texttt{B}) & & \text{for all } \texttt{B} \\
(A, R) \models \mathit{StateChange}(\texttt{B}) & & \text{for all } \texttt{B} \\
(A, R) \models \mathit{AnswerMessages}(\texttt{B}) & & \text{for all } \texttt{B}
\end{array}
\end{array}
$$

$(A, R) \models \mathit{Persistence}(\texttt{B})$ for all B
Induction base : $n = 0 \ \surd$

Induction step : $n \to n+1$

$$
\begin{array}{ll}
 & I(\mathit{Persistence}^{n+1}(\mathtt{B})) \\
= & \quad \{ \text{ definition of } \mathit{Persistence} \ \} \\
 & I(\text{“}\mathtt{< B : BdBuffer >}\text{”} \Rightarrow [\mathtt{-}]\text{“}\mathtt{< B : BdBuffer >}\text{”} \wedge \mathit{Persistence}^{n}(\mathtt{B})) \\
= & \quad \{ \text{ definition of the interpretation function } \} \\
 & \quad \{C \mid C \in A_{\mathrm{Cf}}, \mathtt{< B : BdBuffer >} \notin C\} \\
 & \cup \ \ \widetilde{pre}(R)(-)(\{C \mid \mathtt{< B : BdBuffer >} \in C, C \in I(\mathit{Persistence}^{n}(\mathtt{B}))\}) \\
= & \quad \{ \text{ definition of } \widetilde{pre} \ \} \\
 & \quad \{C | C \in A_{\mathrm{Cf}}, \mathtt{< B : BdBuffer >} \notin C\} \\
 & \cup \ \ A_{\mathrm{Cf}} \setminus pre(R)(-)(\{C \mid \mathtt{< B : BdBuffer >} \notin C \vee C \notin I(\mathit{Persistence}^{n}(\mathtt{B}))\}) \\
= & \quad \{ \text{ induction hypothesis } \} \\
 & \quad \{C | C \in A_{\mathrm{Cf}}, \mathtt{< B : BdBuffer >} \notin C\} \\
 & \cup \ \ A_{\mathrm{Cf}} \setminus pre(R)(-)(\{C \mid \mathtt{< B : BdBuffer >} \notin C \vee C \notin A_{\mathrm{Cf}})\}) \\
= & \quad \{ \text{ set theory } \} \\
 & \quad \{C | C \in A_{\mathrm{Cf}}, \mathtt{< B : BdBuffer >} \notin C\} \\
 & \cup \ \ A_{\mathrm{Cf}} \setminus pre(R)(-)(\{C \mid \mathtt{< B : BdBuffer >} \notin C\}) \\
= & \quad \{ \text{ characterization of the initial model, definition of } \widetilde{pre} \ \} \\
 & \quad \{C | C \in A_{\mathrm{Cf}}, \mathtt{< B : BdBuffer >} \notin C\} \\
 & \cup \ \ A_{\mathrm{Cf}} \setminus \{C \mid \mathtt{< B : BdBuffer >} \notin C\} \\
= & \quad \{ \text{ set theory } \} \\
 & A_{\mathrm{Cf}}
\end{array}
$$

i.e. $C \in I(\mathit{Persistence}^{n}(\mathtt{B})) \Rightarrow C \in I(\mathit{Persistence}^{n+1}(\mathtt{B}))$ $\surd$

$(A, R) \models \mathit{State}(\mathtt{B})$ for all B
Induction base: $n = 0$ $\surd$
Induction step: $n \to n+1$:

$$
\begin{array}{ll}
 & I(\mathit{State}^{n+1}(\mathtt{B})) \\
= & \quad \{ \text{ definition of } \mathit{State} \ \} \\
 & \ \mathtt{for\ all\ I, I', O, O'} \in \mathtt{Nat}, \mathtt{M} \in \mathtt{NzNat}, \mathtt{L, L'} \in \mathtt{List} : \\
 & \qquad (\text{“}\mathtt{< B : BdBuffer \mid in = I,\ out = O,\ max = M,\ cont = L >}\text{”} \\
 & \quad \wedge \ \ \mathtt{length(L)} = \mathtt{I} - \mathtt{O} \\
 & \quad \wedge \ \ \mathtt{0} \leq \mathtt{I} - \mathtt{O} \leq \mathtt{M}) \\
 & \ \Rightarrow [\mathtt{-}] \ (\text{“}\mathtt{< B : BdBuffer \mid in = I',\ out = O',\ max = M,\ cont = L'>}\text{”} \\
 & \qquad\quad \Rightarrow \ \mathtt{length(L')} = \mathtt{I'} - \mathtt{O'} \wedge \\
 & \qquad\qquad \mathtt{0} \leq \mathtt{I'} - \mathtt{O'} \leq \mathtt{M'} \wedge \mathit{State}^{n}) \\
= & \quad \{ \text{ definition of the interpretation function } \}
\end{array}
$$

$$
\begin{array}{l}
\quad \text{for all } \mathtt{I}, \mathtt{I'}, \mathtt{O}, \mathtt{O'} \in \mathtt{Nat}, \mathtt{M} \in \mathtt{NzNat}, \mathtt{L}, \mathtt{L'} \in \mathtt{List} : \\
\qquad A_{\mathrm{Cf}} \setminus \{ C \mid \ \mathtt{< B : BdBuffer \mid in = I,\ out = O,\ max = M,\ cont = L >} \in C, \\
\qquad\qquad\qquad \mathtt{length(L)} = \mathtt{I} - \mathtt{O} \\
\qquad\qquad\qquad 0 \leq \mathtt{I} - \mathtt{O} \leq \mathtt{M} \} \\
\quad \cup \widetilde{pre}(R)(-) \ (\quad \{ C \mid \ \mathtt{< B : BdBuffer \mid in = I',\ out = O',} \\
\qquad\qquad\qquad\qquad\qquad \mathtt{max = M,\ cont = L'>} \notin C \} \\
\qquad\qquad\quad \cup \ \{ C \mid \ \mathtt{< B : BdBuffer \mid in = I',\ out = O',} \\
\qquad\qquad\qquad\qquad\qquad \mathtt{max = M,\ cont = L'>} \in C, \\
\qquad\qquad\qquad\quad \mathtt{length(L')} = \mathtt{I'} - \mathtt{O'} \wedge \\
\qquad\qquad\qquad\quad 0 \leq \mathtt{I'} - \mathtt{O'} \leq \mathtt{M'}, C \in A_{\mathrm{Cf}} \}) \\
= \qquad \{ \text{ characteritzation initial model } \} \\
\quad \text{for all } \mathtt{I}, \mathtt{I'}, \mathtt{O}, \mathtt{O'} \in \mathtt{Nat}, \mathtt{M} \in \mathtt{NzNat}, \mathtt{L}, \mathtt{L'} \in \mathtt{List} : \\
\qquad A_{\mathrm{Cf}} \setminus \{ C \mid \ \mathtt{< B : BdBuffer \mid in = I,\ out = O,\ max = M,\ cont = L >} \in C, \\
\qquad\qquad\qquad \mathtt{length(L)} = \mathtt{I} - \mathtt{O} \\
\qquad\qquad\qquad 0 \leq \mathtt{I} - \mathtt{O} \leq \mathtt{M} \} \\
\quad \cup \ \ \widetilde{pre}(R)(-)(\{ C \mid \ \mathtt{< B : BdBuffer \mid in = I',\ out = O',} \\
\qquad\qquad\qquad\qquad\qquad \mathtt{max = M,\ cont = L'>} \notin C \}) \\
\quad \cup \ \ \widetilde{pre}(R) \ (\mathtt{(get\ B\ replyto\ U)}) \\
\qquad\qquad (\{ C \mid \ \mathtt{< B : BdBuffer \mid in = I',\ out = O',} \\
\qquad\qquad\qquad\qquad \mathtt{max = M,\ cont = L'>} \in C, \\
\qquad\qquad\quad \mathtt{length(L')} = \mathtt{I'} - \mathtt{O'} \wedge \\
\qquad\qquad\quad 0 \leq \mathtt{I'} - \mathtt{O'} \leq \mathtt{M'}, C \in A_{\mathrm{Cf}} \}) \\
\quad \cup \ \ \widetilde{pre}(R) \ (\mathtt{(put\ E\ into\ B)}) \\
\qquad\qquad (\{ C \mid \ \mathtt{< B : BdBuffer \mid in = I',\ out = O',} \\
\qquad\qquad\qquad\qquad \mathtt{max = M,\ cont = L'>} \in C, \\
\qquad\qquad\quad \mathtt{length(L')} = \mathtt{I'} - \mathtt{O'} \wedge \\
\qquad\qquad\quad 0 \leq \mathtt{I'} - \mathtt{O'} \leq \mathtt{M'}, C \in A_{\mathrm{Cf}} \}) \\
\quad \cup \ \ \widetilde{pre}(R) \ (\mathtt{L} \setminus \{ \mathtt{(get\ B\ replyto\ U)(put\ E\ into\ B)} \}) \\
\qquad\qquad (\{ C \mid \ \mathtt{< B : BdBuffer \mid in = I',\ out = O',} \\
\qquad\qquad\qquad\qquad \mathtt{max = M,\ cont = L'>} \in C, \\
\qquad\qquad\quad \mathtt{length(L')} = \mathtt{I'} - \mathtt{O'} \wedge \\
\qquad\qquad\quad 0 \leq \mathtt{I'} - \mathtt{O'} \leq \mathtt{M'}, C \in A_{\mathrm{Cf}} \}) \\
= \qquad \{ \text{ definition of } \widetilde{pre} \ \}
\end{array}
$$

$$
\begin{array}{l}
\quad \text{for all } \mathtt{I},\mathtt{I'},\mathtt{O},\mathtt{O'} \in \mathtt{Nat}, \mathtt{M} \in \mathtt{NzNat}, \mathtt{L},\mathtt{L'} \in \mathtt{List} : \\
\qquad A_{\mathrm{Cf}} \setminus \{C \mid \ \mathtt{< B : BdBuffer \mid in = I,\ out = O,\ max = M,\ cont = L >} \in C, \\
\qquad\qquad \mathtt{length(L)} = \mathtt{I} - \mathtt{O} \\
\qquad\qquad 0 \leq \mathtt{I} - \mathtt{O} \leq \mathtt{M}\} \\
\quad \cup \ \widetilde{pre}(R)(-)\{C \mid \ \mathtt{< B : BdBuffer \mid in = I',\ out = O',} \\
\qquad\qquad\qquad \mathtt{max = M,\ cont = L'>} \notin C\} \\
\quad \cup \ \ \{C \mid \ \mathtt{< B : BdBuffer \mid in = I',\ out = O'-1,} \\
\qquad\qquad\qquad \mathtt{max = M,\ cont = L'\ E >} \in C, \\
\qquad\qquad \mathtt{length(L')} = \mathtt{I'} - \mathtt{O'} \wedge \\
\qquad\qquad 0 \leq \mathtt{I'} - \mathtt{O'} \leq \mathtt{M'}, C \in A_{\mathrm{Cf}}\} \\
\quad \cup \ \ \{C \mid \ \mathtt{< B : BdBuffer \mid in = I'-1,\ out = O',\ max = M,} \\
\qquad\qquad\qquad \mathtt{cont = rmlast(L')\ >} \in C, \\
\qquad\qquad \mathtt{length(L')} = \mathtt{I'} - \mathtt{O'} \wedge \\
\qquad\qquad 0 \leq \mathtt{I'} - \mathtt{O'} \leq \mathtt{M'}, C \in A_{\mathrm{Cf}}\} \\
\quad \cup \ \ \{C \mid \ \mathtt{< B : BdBuffer \mid in = I',\ out = O',\ max = M,\ cont = L'>} \in C, \\
\qquad\qquad \mathtt{length(L')} = \mathtt{I'} - \mathtt{O'} \wedge \\
\qquad\qquad 0 \leq \mathtt{I'} - \mathtt{O'} \leq \mathtt{M'}, C \in A_{\mathrm{Cf}}\} \\
= \qquad \{ \text{ mathematics } \} \\
\quad A_{\mathrm{Cf}}
\end{array}
$$

$(A, R) \models \mathit{Synchronization}(\mathtt{B})$ for all B
Induction base: $n = 0$ $\surd$
Induction step: $n \to n + 1$:

$$
\begin{array}{l}
\quad I(\mathit{Synchronization}^{n+1}(\mathtt{B})) \\
= \qquad \{ \text{ definition of } \mathit{Synchronization}^{n+1} \ \} \\
\quad I(\forall \mathtt{I},\mathtt{O} \in \mathtt{Nat}, \mathtt{M} \in \mathtt{NzNat}, \mathtt{L} \in \mathtt{List}, \mathtt{E} \in \mathtt{Elem}, \mathtt{U} \in \mathtt{ObjectId} : \\
\qquad\qquad (\text{``}\mathtt{< B : BdBuffer \mid in = I,\ out = O,\ max = M,\ cont = L >}\text{''} \\
\qquad\quad \wedge \ \ \mathit{SI}(\mathtt{< B : BdBuffer >}) \\
\qquad\quad \wedge \ \ \mathtt{I} - \mathtt{O} < \mathtt{M} \\
\qquad\quad \Rightarrow \ \langle \mathtt{(put\ E\ into\ B)} \rangle I(\mathit{Synchronization}^{n}(\mathtt{B}))) \\
\qquad \wedge \qquad (\text{``}\mathtt{< B : BdBuffer \mid in = I,\ out = O,\ max = M,\ cont = L\ E >}\text{''} \\
\qquad\quad \wedge \ \ \mathit{SI}(\mathtt{< B : BdBuffer >}) \\
\qquad\quad \wedge \ \ \mathtt{I} - \mathtt{O} > 0 \\
\qquad\quad \Rightarrow \ \langle \mathtt{(get\ B\ replyto\ U)} \rangle I(\mathit{Synchronization}^{n}(\mathtt{B})))) \\
= \qquad \{ \text{ definition of the interpretation function } \}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{for all } \texttt{I}, \texttt{O} \in \texttt{Nat}, \texttt{M} \in \texttt{NzNat}, \texttt{L} \in \texttt{List}, \texttt{E} \in \texttt{Elem}, \texttt{U} \in \texttt{ObjectId}:} \\
& (A_{\mathrm{Cf}} \setminus \{C \mid & \texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L >} \in C, \\
& & \mathit{SI}(\texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L >}), \\
& & \texttt{I} - \texttt{O} < \texttt{M}\} \\
& \cup & \mathit{pre}(R)(\texttt{(put E into B)})(I(\mathit{Synchronization}^n(\texttt{B})))) \\
\cap & (A_{\mathrm{Cf}} \setminus \{C \mid & \texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L E >} \in C, \\
& & \mathit{SI}(\texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L E >}) \\
& & \texttt{I} - \texttt{O} > 0\} \\
& \cup & \mathit{pre}(R)(\texttt{(get B replyto U)})(I(\mathit{Synchronization}^n(\texttt{B})))) \\
= & \multicolumn{2}{l}{\{ \text{ induction hypothesis } \}} \\
\multicolumn{3}{l}{\text{for all } \texttt{I}, \texttt{O} \in \texttt{Nat}, \texttt{M} \in \texttt{NzNat}, \texttt{L} \in \texttt{List}, \texttt{E} \in \texttt{Elem}, \texttt{U} \in \texttt{ObjectId}:} \\
& A_{\mathrm{Cf}} \setminus \{C \mid & \texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L >} \in C, \\
& & \mathit{SI}(\texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L >}), \\
& & \texttt{I} - \texttt{O} < \texttt{M}\} \\
& \cup & \mathit{pre}(R)(\texttt{(put E into B)})A_{\mathrm{Cf}} \\
\cap & A_{\mathrm{Cf}} \setminus \{C \mid & \texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L E >} \in C, \\
& & \mathit{SI}(\texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L E >}) \\
& & \texttt{I} - \texttt{O} > 0\} \\
& \cup & \mathit{pre}(R)(\texttt{(get B replyto U)})A_{\mathrm{Cf}})
\end{array}
$$

$$
= \left\{
\begin{array}{lll}
\multicolumn{3}{l}{\text{definition of } \mathit{pre},} \\
& \multicolumn{2}{l}{\mathit{pre}(R)(\texttt{(put E into B)})(A_{\mathrm{Cf}})} \\
& & \texttt{(put E into B)} \in C, \\
= & \{C \mid & \texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L >} \in C, \\
& & \texttt{I} - \texttt{O} < M\} \\
& \multicolumn{2}{l}{\mathit{pre}(R)(\texttt{(get B replyto U)})(A_{\mathrm{Cf}})} \\
& & \texttt{(get B replyto U)} \in C, \\
= & \{C \mid & \texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L >} \in C, \\
& & \texttt{I} - \texttt{O} > 0\}
\end{array}
\right\}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{for all } \texttt{I}, \texttt{O} \in \texttt{Nat}, \texttt{M} \in \texttt{NzNat}, \texttt{L} \in \texttt{List}, \texttt{E} \in \texttt{Elem}, \texttt{U} \in \texttt{ObjectId}:} \\
\multicolumn{2}{l}{A_{\mathrm{Cf}} \cap A_{\mathrm{Cf}}} \\
= & \{ \text{ set theory } \} \\
\multicolumn{2}{l}{A_{\mathrm{Cf}}}
\end{array}
$$

$(A, R) \models \mathit{StateChange}(\texttt{B})$ for all B
Induction base: $n = 0$ $\surd$
Induction step: $n \to n+1$:

$$
\begin{array}{ll}
\multicolumn{2}{l}{I(\mathit{StateChange}^{n+1}(\texttt{B}))} \\
= & \{ \text{ definition of } \mathit{StateChange}^{n+1} \}
\end{array}
$$

$$
\begin{array}{ll}
& I(\forall \mathtt{I},\mathtt{O},\mathtt{M}\in\mathtt{Nat},\mathtt{M'}\in\mathtt{Nat},\mathtt{L}\in\mathtt{List},\mathtt{E},\mathtt{E'}\in\mathtt{Elem},:\\
& \qquad (\text{``}\mathtt{< B : BdBuffer \mid in = I, out = O, max = M, cont = L >}\text{''}\\
& \quad\wedge \quad \mathit{SI}(\mathtt{< B : BdBuffer >})\\
& \quad\wedge \quad \mathtt{I}-\mathtt{O}<\mathtt{M}\\
& \quad\Rightarrow \quad [\mathtt{(put\ E\ into\ B)}]\\
& \qquad\qquad \text{``}\mathtt{< B : BdBuffer \mid in = I + 1, out = O, max = M, cont = E\ L >}\text{''}\\
& \qquad\qquad \wedge I(\mathit{StateChange}^n(\mathtt{B})))\\
& \wedge \qquad (\text{``}\mathtt{< B : BdBuffer \mid in = I, out = O, max = M, cont = L\ E' >}\text{''}\\
& \quad\wedge \quad \mathit{SI}(\mathtt{< B : BdBuffer >})\\
& \quad\wedge \quad \mathtt{I}-\mathtt{O}>0\\
& \quad\Rightarrow \quad [\mathtt{(get\ B\ replyto\ U)}]\\
& \qquad\qquad (\text{``}\mathtt{< B : BdBuffer \mid in = I, out = O + 1, max = M, cont = L >}\text{''}\\
& \qquad\qquad \wedge I(\mathit{StateChange}^n(\mathtt{B})))))\\
= & \quad \{ \text{ definition of } \widetilde{pre} \ \}\\
& \mathtt{for\ all\ I,O,M}\in\mathtt{Nat},\mathtt{M'}\in\mathtt{Nat},\mathtt{L}\in\mathtt{List},\mathtt{E},\mathtt{E'}\in\mathtt{Elem},:\\
& \qquad (A_{\mathrm{Cf}} \setminus \{C \mid \mathtt{< B : BdBuffer \mid in = I, out = O, max = M,}\\
& \qquad\qquad\qquad\qquad \mathtt{cont = L >}\in C,\\
& \qquad\qquad\qquad \mathit{SI}(\mathtt{< B : BdBuffer >}),\\
& \qquad\qquad\qquad \mathtt{I}-\mathtt{O}<\mathtt{M}\}\\
& \quad\cup \quad \widetilde{pre}(R)\ (\mathtt{(put\ E\ into\ B)})(\\
& \qquad\qquad \{C \mid \ C\in A_{\mathrm{Cf}}\\
& \qquad\qquad\quad \mathtt{< B : BdBuffer \mid in = I + 1, out = O, max = M,}\\
& \qquad\qquad\qquad\qquad \mathtt{cont = E\ L >}\in C,\\
& \qquad\qquad\quad C\in I(\mathit{StateChange}^n(\mathtt{B}))\}))\\
& \cap \qquad (A_{\mathrm{Cf}} \setminus \{C \mid \mathtt{< B : BdBuffer \mid in = I, out = O, max = M,}\\
& \qquad\qquad\qquad\qquad \mathtt{cont = L\ E' >}\in C,\\
& \qquad\qquad\qquad \mathit{SI}(\mathtt{< B : BdBuffer >}),\\
& \qquad\qquad\qquad \mathtt{I}-\mathtt{O}>0\}\\
& \quad\cup \quad \widetilde{pre}(R)\ (\mathtt{(get\ B\ replyto\ U)})\\
& \qquad\qquad (\{C \mid \mathtt{< B : BdBuffer \mid in = I, out = O + 1, max = M,}\\
& \qquad\qquad\qquad\qquad \mathtt{cont = L >}\in C,\\
& \qquad\qquad\quad C\in I(\mathit{StateChange}^n(\mathtt{B}))\}))\\
= & \quad \{ \text{ characterization of the initial model, definition of } \widetilde{pre}\text{, induction hyp. } \}
\end{array}
$$

$$
\begin{array}{ll}
& \text{for all } \mathtt{I},\mathtt{O},\mathtt{M}\in\mathtt{Nat},\mathtt{M'}\in\mathtt{Nat},\mathtt{L}\in\mathtt{List},\mathtt{E},\mathtt{E'}\in\mathtt{Elem},: \\
& \quad (A_{\mathrm{Cf}} \setminus \{C \mid \mathtt{< B : BdBuffer \mid in = I, out = O,} \\
& \qquad\qquad \mathtt{max = M, cont = L >} \in C, \\
& \qquad\qquad \mathit{SI}(\mathtt{< B : BdBuffer >}), \\
& \qquad\qquad \mathtt{I} - \mathtt{O} < \mathtt{M}\} \\
& \quad \cup\ \{C \mid \mathtt{< B : BdBuffer \mid in = I, out = O,} \\
& \qquad\qquad \mathtt{max = M, cont = L >} \in C, \\
& \qquad\quad C \in A_{\mathrm{Cf}}\}) \\
& \cap \quad (A_{\mathrm{Cf}} \setminus \{C \mid \mathtt{< B : BdBuffer \mid in = I, out = O,} \\
& \qquad\qquad \mathtt{max = M, cont = L\ E' >} \in C \\
& \qquad\qquad \mathit{SI}(\mathtt{< B : BdBuffer >}), \\
& \qquad\qquad \mathtt{I} - \mathtt{O} > 0\} \\
& \quad \cup\ \{C \mid \mathtt{< B : BdBuffer \mid in = I, out = O\ , max = M,} \\
& \qquad\qquad \mathtt{cont = L\ E >} \in C, \\
& \qquad\quad C \in A_{\mathrm{Cf}}\}) \\
= & \quad \{ \text{ set theory } \} \\
& \mathtt{for\ all}\ \mathtt{I},\mathtt{O},\mathtt{M}\in\mathtt{Nat},\mathtt{M'}\in\mathtt{Nat},\mathtt{L}\in\mathtt{List},\mathtt{E},\mathtt{E'}\in\mathtt{Elem},: \\
& A_{\mathrm{Cf}} \cap A_{\mathrm{Cf}} \\
= & \quad \{ \text{ set theory } \} \\
& A_{\mathrm{Cf}}
\end{array}
$$

$(A, R) \models \mathit{AnswerMessages}(\mathtt{B})$ for all B
Induction base : $n = 0$ $\surd$
Induction step : $n \rightarrow n + 1$:

$$
\begin{array}{ll}
& I(\mathit{AnswerMessages}^{n+1}(\mathtt{B})) \\
= & \quad \{ \text{ definition of } \mathit{AnswerMessages}^{n+1} \ \} \\
& I(\forall \mathtt{I},\mathtt{O}\in\mathtt{Nat},\mathtt{M}\in\mathtt{NzNat},\mathtt{E},\mathtt{E'}\in\mathtt{Elem},\mathtt{L}\in\mathtt{List},\mathtt{U}\in\mathtt{ObjectId}: \\
& \qquad (\text{“}\mathtt{< B : BdBuffer \mid in = I, out = O, max = M, cont = L >}\text{”} \\
& \quad\ \wedge \quad \mathit{SI}(\mathtt{< B : BdBuffer >}) \\
& \quad\ \wedge \quad \mathtt{I} - \mathtt{O} < \mathtt{M} \\
& \quad\ \Rightarrow \quad [\mathtt{(put\ E\ into\ B)}]\mathit{AnswerMessages}^{n}(\mathtt{B})) \\
& \wedge \qquad (\text{“}\mathtt{< B : BdBuffer \mid in = I, out = O, max = M, cont = L\ E' >}\text{”} \\
& \quad\ \wedge \quad \mathit{SI}(\mathtt{< B : BdBuffer >}) \\
& \quad\ \wedge \quad \mathtt{I} - \mathtt{O} > 0 \\
& \quad\ \Rightarrow \quad [\mathtt{(get\ B\ replyto\ U)}](\ \text{“}\mathtt{(to\ U\ answer\ to\ get\ is\ E)}\text{”} \\
& \qquad\qquad\qquad \wedge \mathit{AnswerMessages}^{n}(\mathtt{B})))) \\
= & \quad \{ \text{ definition interpretation function } \}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\texttt{for all I,O}\in\texttt{Nat},\texttt{M}\in\texttt{NzNat},\texttt{E},\texttt{E'}\in\texttt{Elem},\texttt{L}\in\texttt{List},\texttt{U}\in\texttt{ObjectId}:} \\
 & (A_{\mathrm{Cf}} \setminus \{C \mid & \texttt{< B : BdBuffer | in = I, out = O, max = M,} \\
 & & \qquad\qquad \texttt{cont = L >} \in C, \\
 & & \mathit{SI}(\texttt{ < B : BdBuffer | in = I, out = O, max = M,} \\
 & & \qquad\qquad\qquad \texttt{cont = L >}), \\
 & & \texttt{I} - \texttt{O} < \texttt{M}\} \\
 & \cup & \widetilde{pre}(\texttt{put E into B})\, I(\mathit{AnswerMessages}^n)(\texttt{B})\}) \\
\cap & (A_{\mathrm{Cf}} \setminus \{C \mid & \texttt{< B : BdBuffer | in = I, out = O, max = M,} \\
 & & \qquad\qquad \texttt{cont = L E' >} \in C, \\
 & & \mathit{SI}(\texttt{< B : BdBuffer | in = I, out = O, max = M,} \\
 & & \qquad\qquad\qquad \texttt{cont = L E' >}), \\
 & & \texttt{I} - \texttt{O} > 0\} \\
 & \cup & \widetilde{pre}(R)((\texttt{get B replyto U}))(\{C \mid \; (\texttt{to U answer to get is E}) \in C, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad C \in I(\mathit{AnswerMessages}^n)(\texttt{B})\}))
\end{array}
$$

$= \quad$ { induction hypothesis }

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{for all } \texttt{I,O}\in\texttt{Nat},\texttt{M}\in\texttt{NzNat},\texttt{E},\texttt{E'}\in\texttt{Elem},\texttt{L}\in\texttt{List},\texttt{U}\in\texttt{ObjectId}:} \\
 & (A_{\mathrm{Cf}} \setminus \{C \mid & \texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L >} \in C, \\
 & & \mathit{SI}(\texttt{ < B : BdBuffer | in = I, out = O,} \\
 & & \qquad\qquad\qquad \texttt{max = M, cont = L >}), \\
 & & \texttt{I} - \texttt{O} < \texttt{M}\} \\
 & \cup & \widetilde{pre}(\texttt{put E into B})(A_{\mathrm{Cf}})) \\
\cap & (A_{\mathrm{Cf}} \setminus \{C \mid & \texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L E' >} \in C, \\
 & & \mathit{SI}(\texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L E' >}), \\
 & & \texttt{I} - \texttt{O} > 0\} \\
 & \cup & \widetilde{pre}(R)((\texttt{get B replyto U}))(\{C \mid \; (\texttt{to U answer to get is E}) \in C, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad C \in A_{\mathrm{Cf}}\}))
\end{array}
$$

$= \quad$ { set theory }

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{for all } \texttt{I,O}\in\texttt{Nat},\texttt{M}\in\texttt{NzNat},\texttt{E},\texttt{E'}\in\texttt{Elem},\texttt{L}\in\texttt{List},\texttt{U}\in\texttt{ObjectId}:} \\
 & A_{\mathrm{Cf}} \setminus \{C \mid & \texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L >} \in C, \\
 & & \mathit{SI}(\texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L >}), \\
 & & \texttt{I} - \texttt{O} < \texttt{M}\} \\
 & \cup & A_{\mathrm{Cf}} \\
\cap & A_{\mathrm{Cf}} \setminus \{C \mid & \texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L E' >} \in C, \\
 & & \mathit{SI}(\texttt{< B : BdBuffer | in = I, out = O, max = M, cont = L E' >}), \\
 & & \texttt{I} - \texttt{O} > 0\} \\
 & \cup & \widetilde{pre}(R)((\texttt{get B replyto U}))\{C \mid \; (\texttt{to U answer to get is E}) \in C, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad C \in A_{\mathrm{Cf}}\}
\end{array}
$$

$$
= \left\{ \begin{array}{l}
\text{set theory, definition of } \widetilde{pre}\text{:} \\
\quad \widetilde{pre}(R)(\texttt{(get B replyto U)})(\{C \mid \begin{array}[t]{l}\texttt{(to U answer to get is E)} \in C, \\ C \in A_{\mathrm{Cf}}\})\end{array} \\
\quad = \overline{pre(R)(\texttt{(get B replyto U)})(\overline{\{C \mid \begin{array}[t]{l}\texttt{(to U answer to get is E)} \in C, \\ \overline{C \in A_{\mathrm{Cf}}\}})\end{array}}} \\
\quad = \overline{pre(R)(\texttt{(get B replyto U)})(\{C \mid \begin{array}[t]{l}\texttt{(to U answer to get is E)} \notin C, \\ \overline{\overline{C \in A_{\mathrm{Cf}}\}}})\end{array}} \\
\quad = \overline{\emptyset} \\
\quad = A_{\mathrm{Cf}}
\end{array} \right\}
$$

for all $\texttt{I}, \texttt{O} \in \texttt{Nat}, \texttt{M} \in \texttt{NzNat}, \texttt{E}, \texttt{E'} \in \texttt{Elem}, \texttt{L} \in \texttt{List}, \texttt{U} \in \texttt{ObjectId}$:

$A_{\mathrm{Cf}} \cap A_{\mathrm{Cf}}$

$= \quad \{ \text{ set theory } \}$

A_{Cf}

□

C.2 Specification PEC

```
module PEC {
  import {
   protecting (NAT)
   protecting (LIST)
   protecting (MESSAGELIST)
   extending  (CONTROL)
  }

  signature {
  class Sender {
    rec      : ObjectId
    sendq    : List
    sendbuff : Elem
    sendcnt  : Nat
  }

  class Receiver {
    sender   : ObjectId
    incoming : List
    reccnt   : Nat
  }

  op empty                          : -> Elem
```

```
  op initialize sender _ with _    : ObjectId ObjectId          -> Message
  op initialize receiver _ with _  : ObjectId ObjectId          -> Message
  op to _ ack _  from _            : ObjectId Nat ObjectId      -> Message
  op to _ (_,_) from _             : ObjectId Elem Nat ObjectId -> Message

  op [[initialize sender _]]     : ObjectId -> Ctrl
  op [[produce _]]               : ObjectId -> Ctrl
  op [[send _]]                  : ObjectId -> Ctrl
  op [[rec-ack _]]               : ObjectId -> Ctrl
  op [[rec-ack not suitable _]]  : ObjectId -> Ctrl
  op [[initialize receiver _]]   : ObjectId -> Ctrl
  op [[receive successfully _]]  : ObjectId -> Ctrl
  op [[receive - wrong cnt _]]   : ObjectId -> Ctrl
  op [[loose message]]           : -> Ctrl
  op [[duplicate message]]       : -> Ctrl
}

  axioms {
    vars S S' R R' : ObjectId
    vars M N N'    : Nat
    var  MSG       : Message
    vars L ML L'   : List
    vars E E'      : Elem
    var  ATTS      : Attributes

rl [initialize-sender]:
   [[initialize sender S]]
   (initialize sender S with R)
   < S : Sender | rec = R', sendq = ML,
                  sendbuff = E', sendcnt = N', ATTS >
   =>  < S : Sender | rec = R, sendq = ML,
                      sendbuff = empty, sendcnt = 0, ATTS > .

rl [produce]:
   [[produce S]]
   < S : Sender | sendq = E L, sendbuff = empty, sendcnt = N, ATTS >
   => < S : Sender | sendq = L, sendbuff = E, sendcnt = N + 1, ATTS > .

crl [send]:
    [[send S]]
    < S : Sender | rec = R, sendbuff = E, sendcnt = N, ATTS >
    => < S : Sender | rec = R, sendbuff = E, sendcnt = N, ATTS >
```

```
       (to R (E,N) from S)
   if E =/= empty:Elem .

rl [rec-ack]:
   [[rec-ack S]]
   (to S ack N from R)
    < S : Sender | rec = R,  sendbuff = E, sendcnt = N, ATTS >
    => < S : Sender | rec = R, sendbuff = empty, sendcnt = N, ATTS > .

crl [rec-ack-not-suitable]:
    [[rec-ack not suitable S]]
    (to S ack N from R)
    < S : Sender | rec = R, sendcnt = M, ATTS >
    => < S : Sender | rec = R, sendcnt = M, ATTS >
    if N =/= M .

rl [initialize-receiver]:
   [[initialize receiver R]]
   (initialize receiver R with S)
   < R : Receiver | sender = S', incoming = L', reccnt = N', ATTS >
    => < R : Receiver | sender = S, incoming = empty, reccnt = 0, ATTS > .

crl [receive-successfully]:
    [[receive successfully R]]
    (to R (E,N) from S)
    < R : Receiver | sender = S, incoming = L, reccnt = M, ATTS >
    => < R : Receiver | sender = S, incoming = L E, reccnt = M + 1, ATTS >
       (to S ack N from R)
    if N == M + 1 .

crl [receive-wrong-cnt]:
     [[receive - wrong cnt R]]
    (to R (E,N) from S)
    < R : Receiver | sender = S, incoming = L, reccnt = M, ATTS >
    => < R : Receiver | sender = S, incoming = L, reccnt = M, ATTS >
       (to S ack M from R)
    if N =/= M + 1 .

crl [loose-message]:
     [[loose message]]
    MSG
    => acz-empty
    if elemoflist(MSG, conc((to S ack N from R),
```

```
                  conc((to R  (E,N) from S), empty))) .

crl [duplicate-message]:
    [[duplicate message]]
    MSG => MSG MSG
    if elemoflist(MSG, conc((to S ack N from R),
                       conc((to R (E,N) from S), empty))) .
  }
}
```

C.3 Addendum to the Specification of the Protocol

We present the basic data types used in the specifications of the protocol and give a script, which demonstrates how the different specification modules are related.

```
in LIST
in MSG-ALGEBRA
in EXT-ACZ-CONFIGURATION
in BD-BUFFER
in BD-BUFFER-INTERN

module EXPLICITLIST {
  signature {
     [ElemofExplicitList ExplicitList]
  op empty      : -> ExplicitList
  op conc       : ElemofExplicitList ExplicitList -> ExplicitList
  op elemoflist : ElemofExplicitList ExplicitList -> Bool
  }

  axioms {
    vars E E' : ElemofExplicitList
    var  L    : ExplicitList

  eq elemoflist(E,empty)      = false .
  eq elemoflist(E,conc(E,L))  = true .
  eq elemoflist(E,conc(E',L)) = elemoflist(E,L)   .
  }
}

module MESSAGELIST {
  import {
    protecting (ACZ-CONFIGURATION)
```

```
    protecting (EXPLICITLIST)
   }

   signature {
     [Message < ElemofExplicitList]
   }
}

in AP
in CP
in CONTROL
in PEC
in CONTROL-TERMS
in CONTROL-ALGEBRA
in SCONTROL
```

Index