PEDECIBA Informática

Instituto de Computación
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

A Formal Semantics of State Modification Primitives of Object-Oriented Systems

Andrés Vignaga

Trabajo de tesis para la obtención del grado de Magíster en Informática de la Universidad de la República en el programa de Maestría del área Informática del Pedeciba

Orientadores: Dr. Gustavo Betarte
Dr. Alvaro Tasistro

Presentación: 13 de febrero de 2004

A Formal Semantics of State Modification Primitives of Object-Oriented Systems
Andrés Vignaga

ISSN 0797-6410
Tesis de Maestría en Informática
Reporte Técnico RT04-01
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República

Montevideo, Uruguay, Febrero de 2004

A Formal Semantics of State Modification Primitives of Object-Oriented Systems

Andrés Vignaga

Instituto de Computación
Facultad de Ingeniería
Universidad de la República
avignaga@fing.edu.uy

February 9, 2004

Abstract

Sets of primitive operations for state modification of object-oriented systems have been proposed elsewhere for expressing system behavior at a conceptual level. Primitives are operations that perform basic changes on states, and they ultimately are the basic constructs from whom higher level operations can be built. The main problem addressed in this work is the statement of a formal semantics, with a high level of abstraction, of a set of five system state modification primitives. The modifications which are the subject of our study concern creation and destruction of objects, creation and destruction of links, and update of attribute values. Although in some cases informal descriptions of the primitives may suffice, a formal specification allows removing subtle ambiguities and leave an open door for the application of formal methods to early phases of object-oriented systems development. Programs expressed as sequences of state changes are shown to be powerful enough to represent non-trivial system level operations. A framework for rigorously reasoning about these programs is proposed, where proofs of program correctness can be constructed. A simple environment for specifying and verifying system behavior is prototyped using a proof assistant, where users are assisted in the task of proof construction, and where proofs are mechanically checked. Automatic proof construction within the environment is also explored. The framework is applied to a realistic information system within a known problem already developed in the bibliography. This case study demonstrates the feasibility and power of our approach.

This work is devoted to specifically study system state modification primitives and applications with mathematical rigor, and contributes to fill the existing gap in the area between related formalizations and works which informally describe concepts widely used in practice.

Acknowledgements

Reached the end of this long and hard journey, I would like to express my gratitude to my study supervisor and co-thesis advisor Gustavo Betarte for his vision in proposing such an interesting problem and for his valuable long-distance support. I am also grateful to my other co-thesis advisor Alvaro Tasistro for his experience and support, and for doing just a great job. He thoroughly reviewed this report from its early beginnings and made many interesting suggestions. I learned much from both of them. I would like to specially thank my fellow team member Daniel Perovich for countless fruitful and helpful discussions. Furthermore, I thank Cristina Cornes, Cecilia Bastarrica from Universidad de Chile, and Raúl Ruggia for kindly accept to serve on my thesis committee and for their valuable feedback. Alberto Pardo and Nora Szasz, as well as many other people at Instituto de Computación, specially my office mates Leonardo Rodríguez, Daniel Calegari, Alvaro Rodríguez, Dieter Spangenberg, Jorge Corral and Diego Rivero, directly or indirectly contributed to this work. Eduardo Giménez from Trusted Logic, France, also provided valuable comments. I am thankful to Laura Bermúdez, Mabel Seroubian and Joseline Cortazzo from Pedeciba, for their efficiency and help. Last, but not least, I thank my family for their support, specially my mother and my late father. My special thanks go to Clara for her incredible patience and continuous encouragement.

Contents

1 Introduction **1**

2 Background and Motivation **7**

2.1 Object-Oriented Systems 7
2.1.1 The Object-Oriented Approach 8
2.1.2 Static and Dynamic Views of Systems 9
2.1.3 State Well-formedness 10
2.2 System State Modification Primitives 11
2.2.1 Motivation 12
2.2.2 Informal Semantics of Primitives 12
2.2.3 A Simple Example 14
2.3 Program Specification and Correctness 15
2.3.1 The Axiomatic Approach 16
2.3.2 Application to Object-Orientation 17
2.4 Proof Systems and Proof Assistants 18
2.4.1 Proof Systems and Formal Proofs 18
2.4.2 Proof Assistants 19
2.4.3 The Role of Coq 19

3 Specification of System State Modification Primitives **21**

3.1 Types and Specification Approach 22
3.1.1 Structural Concepts 22
3.1.2 Abstract Data Types Specification 23
3.2 Specification of Basic Types 24
3.3 Specification of type System 26
3.3.1 Functions 26
3.3.2 Preconditions 27
3.3.3 Axioms 29
3.3.4 Example 31
3.4 Specification of type State 32
3.4.1 Functions 33
3.4.2 Preconditions 34
3.4.3 Relations between States 35
3.4.4 Axioms 37

3.4.5 Example 38
3.5 Discussion 39

4 Reasoning about the Use of Primitives 43
4.1 Our Approach to Program Verification 44
4.1.1 Motivation 44
4.1.2 Inference Systems 45
4.2 The Framework 45
4.2.1 Formulae 45
4.2.2 Axioms 47
4.2.3 Inference Rules 48
4.2.4 Using the Framework 53
4.2.5 Analysis 56
4.3 An Extension 56
4.3.1 Assertions and Inference Rules 56
4.3.2 Additional Rules 60
4.3.3 Example 63
4.4 Reasoning about State Well-formedness 64
4.4.1 System Invariants 64
4.4.2 Multiplicities 66

5 Formalization of the Framework in Coq 73
5.1 Formalization of Abstract Data Types 73
5.1.1 Types 74
5.1.2 Functions 74
5.1.3 Axioms 74
5.1.4 Preconditions 75
5.1.5 Other Definitions 75
5.2 Formalization of the Framework 76
5.2.1 The Core 76
5.2.2 The Extension 77
5.3 Experiments on Automatic Proof Construction 79
5.3.1 The Coq Approach to Automatizing 80
5.3.2 Application to the Framework 80

6 A Case Study 83
6.1 The Point-of-Sale System 83
6.1.1 The Process Sale Use Case 84
6.1.2 Point-of-Sale System Structure 85
6.1.3 Software Contracts 86
6.2 The Framework in Practice 87
6.2.1 Correctness Assertions for Process Sale Operations 87
6.2.2 An Invariant of the POS System 89

7 Conclusions and Further Work **91**
7.1 Summary and Conclusions 91
7.2 Further Work 93

A Other Semantics **95**
A.1 A Set Theory based Specification 95
A.2 An Object-Oriented based Specification 97

B Properties on Well-formedness **101**
B.1 Basic Properties 101
B.2 Modification of Well-formed States 103
B.3 Modification of Ill-formed States 106

Bibliography **111**

Index **116**

Chapter 1

Introduction

Systems that model computational behavior typically include a notion of state that stores information. As this information, and therefore the state, usually changes over the time, these changes determine the evolution and life cycle of a system. A good grasp of possible state changes is essential for either defining or understanding the overall behavior of a system. Operations or programs that have effect on system states may differ in their purpose and level of abstraction, but they are all ultimately expressed in terms of *system state modification primitives*, which are operations that perform basic atomic changes on system states.

From an implementation point of view, state modification primitives may be regarded as basic programming language constructs (such as Java's `new` operator or C++'s `delete` operator). In most programming languages some object-oriented concepts (commonly *associations* and related notions) are not supported. Therefore, idioms need to be specifically defined for the programming language in order to implement those concepts. For this reason, the implementation of conceptually simple state modifications may imply some complex logic involving pointer or reference handling. From a specification perspective, state modification primitives describe basic state changes at a conceptual level using the full object oriented terminology, instead of a programming language-constrained and implementation-driven one. In this way, we believe that primitives are a useful tool for thinking about more complex state modifications and describing them conceptually in early phases of software development, and thus avoiding the (increasing) complexity associated to late phases of the development process.

System state modification primitives have been used in many other works. However, a definition of their semantics with a formal background has been only partially addressed. A fully formalized semantics for them would naturally remove any subtle ambiguities, and can be useful for rigorously reasoning about the use of the primitives, in particular about system behavior.

The main problem addressed in this work is the statement of a formal semantics for a set of system state modification primitives. State-of-the-art object oriented concepts have grown fast in their complexity in the past few years, especially due to the joint effort of the object community in the definition of the Unified Modeling Language (UML) [OMG03b]. Not all the concepts from this huge domain are used in practice with the same frequency. For that reason, in this work we shall cover only a subset of those concepts. Particularly, we will be focusing on those most frequently used, which allow us to define simple but yet realistic system states and handle useful modifications on them.

Related Work

The UML Reference Manual [RJB98], first introduced the idea of primitive operations for state modification. An informal description for them was embedded in the description of basic concepts, such as *object* and *link*. The idea of using modification primitives for describing system behavior was taken from [Lar98]. There, a set of primitives were suggested, albeit not defined, for specifying the behavior of system operations.
In object-oriented software engineering, many object-oriented concepts have been formalized in different works, but to the extent of the author's knowledge this is the first time a semantics for system state modification primitives is formally addressed.
Animation techniques have been applied in [Oli99, OK99] for validation purposes. There, a system specified by a UML model is animated by showing the sequence of states resulting from the application of operations specified by OCL [OMG03a, WK98] constraints. Each operation is realized by translating its specification into a sequence of modification primitives (called *operations on snapshots*). The USE tool [Ric01], implements an environment for performing animation in a similar way. The work that originated the tool [Ric02] focuses in validating the OCL constraints that specify class operations, rather than in validating the model that specifies the system. In USE, the execution of an operation is simulated by interactively performing the application of system state modification primitives (called *system state manipulation commands*) to a given state.
A response to the Action Semantics for the UML RfP [OMG98] became part of the current adopted version of UML (version 1.5), introducing in its metamodel the Actions package. The elements in this package conform an action language for modeling executable programs, typically methods. Among others, these constructs include different kinds of actions, as well as control structures. Some metaclasses in the Read Write Actions subpackage [OMG03c, p. 2-252] specify actions with side effects on states, corresponding to modification primitives. A subset of UML together with this action language is named eXecutable UML (xUML) [KC03]. The xUML process is closely related to Model Driven Architecture (MDA) [OMG01], and involves the definition of platform-independent executable models which can be then realized into platform-specific code. MDA

provides a separation between conceptual modeling from technology specific modeling, and xUML applies the action language defined within UML for modeling and thinking about behavior in that first conceptual stage. Although an important effort is still carried out in definition of a semantics of an action language (we refer to [ASC01] for further information), action semantics is ultimately specified in natural language, as it is defined within the UML metamodel specification.

In [AL98], a logic for an imperative object-oriented language was first introduced. That language is an object based variant of the ς-calculi presented in [AC96]. A semantics of some object manipulations such as object creation and attribute update is defined, having in fact much in common with parts of the semantics defined in this work. However, issues concerning associations, which are vital in object-oriented software engineering areas are not addressed. Similarly, [Mül02] presents a logic for a programming language similar to sequential Java.

In conclusion, many authors identified a number of system state modification primitives. Some of them are based on very basic concepts and were defined formally, and others not. Different levels of rigor was applied to the specification of their semantics, from naming-descriptive approaches, through brief textual descriptions, to metamodeling. Although in some cases such informal descriptions may suffice, a formal specification would avoid ambiguities and leave an open door for the application of formal methods.

Contributions of this Work

The primary goal of this thesis is to define a formal semantics for a set of basic system state modification primitives. The benefits of the formalization is demonstrated by the definition of a framework which allows to reason about the use of primitives on states. The framework will permit the construction of reliable proofs of properties on states modified by primitives. By treating sequences of state modifications as programs and pairs of properties on states as specifications for programs, the framework will allow the proof of correctness of such programs with respect to their specification.

In summary, this work makes the following contributions:

- A formal semantics for a reduced but intuitively powerful enough set of system state modification primitives is specified. This provides a solid clarification of known object-oriented concepts, and tends to bridge the gap between existing formalizations and those concepts used in practice.
- A framework for reasoning about the use of primitives is defined. Based on the defined semantics, the framework is a tool for constructing programs as sequences of primitives, and proving their correctness with respect to

specifications in the form of pre- and postconditions, as well as properties on final states.

- A specification of the framework for a proof assistant is provided. The use of a proof assistant facilitates the application of the framework, since part of the work in building proofs is delegated to its proof engine. Moreover, the proof assistant mechanically checks the proofs built, making them completely reliable.

- A starting point is provided for the generation of an environment where the specification and verification of modifications on system states can be reliably performed. Tools for analysis and verification could use the results presented in this report.

This approach can be useful particularly for data-intensive systems, where some functionalities are usually implemented as operations that affect the state of the system. This kind of operations are normally called *system operations*, and can be realized by modification primitives. System operations are used in development processes such as the Unified Process (UP) [JBR99] to implement the steps of system use cases. [Lar02] is an example of this methodology. Use cases are a widely used tool for specifying system behavior [Coc01]. Therefore the benefits of our approach can be propagated to higher levels of abstraction.

Finally, we should note that this approach could be used to specify system level operations, but not exclusively so. In fact, it can be used to specify and reason about operations of modules with different levels of granularity. The only existing restriction is that the module, while exhibiting a public interface (i.e. a set of public operations), needs to be object oriented in its internal structure. Examples of mid-grained modules are subsystems and components. Particularly, components are software units with usually manageable size and complexity, being a feasible target for our framework. Work focusing on software components specification has been done in [CD01]. Component interfaces with component operations are a very hot spot in Component-based Development (CBD), since their specification is a key for the component to fulfill one of its most important purposes: to be identified as a compatible building block for an application for its later use or replacement.

Approach Followed

Before ending this introduction, we overview the approach followed for producing the results. The results are produced in two major steps:

1. A precise but not completely formalized specification of the semantics of the primitives, related notions and infrastructure of the framework is first introduced.

2. A formalization of the specification is then generated.

In the first step we present a mathematical specification of the semantics of system state modification primitives, the framework, and other necessary concepts of our theory. It is an intermediate step towards the formal specification. In this step we emphasize understandability, through standard and well known specification techniques, over extreme rigor. We use abstract data types (ADTs) as a tool for specifying notions present in object-oriented systems (e.g. the state of a system). The primitives are then functions associated to a type State, and their semantics is defined axiomatically. In turn, the framework is specified as an inference system, in which rules are defined in the usual way.
In the second step, the formalization process is carried out using the system Coq [INR02]. The specification introduced in the first step is translated into the Calculus of Inductive Constructions, which is a variant of Constructive Type Theory [Wer94, PM97]. Finally, the code generated for the system Coq embodies the complete formalization of both the semantics of the system state modification primitives and the framework.

Structure of the Thesis

This thesis is structured as follows. Chapter 2 includes background information on object oriented systems, program specification and verification, and proof systems. The set of system state modification primitives to be tackled and the framework for reasoning about their use are further motivated. Chapter 3 introduces an axiomatic semantics for the modification primitives based on the specification of abstract data types. Modification primitives and other operations used to define the framework are defined as functions associated to those types. In Chapter 4, an inference system that realizes our framework for reasoning about the use of primitives is presented. The steps followed for producing the formalization in Coq are reported in Chapter 5. A case study is presented in Chapter 6, where we demonstrate the use of the framework. Chapter 7 concludes the thesis with a short summary and an outline of future work.

Chapter 2

Background and Motivation

In this chapter we present background information on the object-oriented paradigm, the general context in which the subject of our study is placed. Herein, system state modification primitives are motivated and informally described. Our exploration of the application of the formal semantics of the primitives to software specification and correctness is motivated too. Finally, proof systems and proof assistants are reviewed as they contribute to the formalization of our specification.

This chapter is structured as follows. In section 2.1 we review object-oriented systems, from the essence of object-oriented approach to the concrete structure of object-oriented systems. Section 2.2 motivates and describes the set of system state modification primitives that will be treated in this work. Section 2.3 gives an overview of program specification and program correctness and its applications to object-oriented development. An application of the primitives to program correctness is also presented. In section 2.4 we overview proof systems for formalization and proof verification, proof assistants for mechanic formal proof handling, and discuss the role played by the system Coq in our work.

2.1 Object-Oriented Systems

In this section we discuss concepts of object orientation that will be used throughout this work. Starting with a brief description of the object-oriented approach, we then review current trends in object-oriented development. We also explore the key ideas in widely used methods, and the positioning of modification primitives among them. Then we analyze in more detail the structure of object-oriented systems from a runtime and static points of view as a foundation for the introduction of the primitives and the specification addressed in the next chapter. Finally, state well-formedness is addressed and its connection with the primitives is discussed.

2.1.1 The Object-Oriented Approach

The *object-oriented* approach to software development is a specific way of thinking of a software product that goes beyond a particular style of programming. It is strongly based on establishing an analogy between a software system and the physical system being represented. Entities in software, called *objects*, are abstractions of real world concepts, and emulate how things work in the physical system. The tight correspondence between entities in both worlds makes a system developed according to the object-oriented approach intuitive and understandable.
In the last decade a number of methods and techniques for object-oriented development appeared, remarkably OMT [RBP+91], OOSE [JCJÖ92], and Booch [Boo94]. A unification of their notation (the Unified Modeling Language) was quickly adopted as a standard. Although a unification of their methods, the Unified Process (UP) [JBR99], did not reach that status yet, the UP and its commercial refinement, the Rational Unified Process (RUP) [Kru00], emerged as popular software development processes for building object-oriented software. UP combines commonly accepted best practices in object-oriented development in a well documented description, but does not recommend concrete activities for object-oriented analysis and design.
This kind of activities, in the context of UP, are treated in detail in the popular text book by Larman [Lar02]. In general terms, object-oriented analysis activities involve abstracting significant concepts from the problem domain, as an intuitive first draft for further refinement of the classes of objects that will participate in the actual solution, and capturing requirements as use cases. Use cases can be roughly described as complete stories of using a system to meet goals. They can be expressed in a narrative form, but being completely informal they are meant for human interpretation only. These stories are then expressed in a more precise way as interactions between the system as a black box and external agents. An interaction is typically a sequence of calls to system operations. Some of them just query the system state, and others modify it. In turn, object-oriented design activities involve the definition of actual software entities that will participate in the execution of system operations inside the system, and the way they will collaborate to resolve them. These collaborations (objects and operations) are said to realize the story told in the use case. Design models usually limit their repertoire of object-oriented constructs to those supported by the programming language or technology adopted in order to simplify the code generation phase. Finally, object-oriented implementation activities involve the translation of the design artifacts into object-oriented code. Code generation for business or domain objects (i.e. objects which mainly hold information about domain entities) is highly automated by CASE tools such as Rational Rose [IBM03] and Together [Bor03].
The set of all use cases describes all the possible ways to use a system, and therefore completely specifies its behavior. The semantics of system operations then plays a vital role in the development process, since objects will be designed for fulfilling the system operation's intended behavior. As we shall see

in this work, this semantics can be defined and specified using system state modification primitives.

2.1.2 Static and Dynamic Views of Systems

A system is object-oriented when it is organized at runtime as a collection of connected objects that incorporate data structure and behavior [RBP+91]. Many definitions of object-oriented systems are available from different authors, each emphasizing different aspects of the system according to their particular intentions. From this wide range of definitions we chose the above definition for focusing on structure. It will be our basis for both defining the notion of system, and for motivating and further defining the modification primitives. This definition considers *objects* as more a primitive notion than *classes*, as in [AC96]. In fact, this definition emphasizes a dynamic (runtime) view, rather than a static one. Let us take a closer look at the dynamic view first and analyze the given definition. First, object-oriented systems are organized as a *collection of objects*. Objects encapsulate state and behavior [RJB98] and are structured as: a set of named slots holding typed values, and a set of implemented operations. Since we are not interested in the individual behavior of objects we ignore operations, and thus we omit them from objects structure. Second, objects *are connected* to other objects. Connections between objects are achieved by means of links. A link is a tuple (with at least two elements) of objects, and an object in a tuple is said to be connected to the rest of the objects in the tuple.
Now looking at systems from a static point of view, identifying and defining individual objects and relationships between them is not practical [EKW92]. Therefore, common properties of objects such as slots and even operations are abstracted away in classes. A class is an abstraction of a set of objects that share the same set of properties. In a class we find the description of these common properties. Particularly, a class has typed attributes that describe slots in objects. Analogously, an association abstracts away the common properties of a set of tuples of connected objects (e.g. the type of objects they refer to). Thus an association is a relationship between classes, where the same collection of classes may participate in a relationship (but under a different name) more than once. An association also includes constraints on the number of tuples that may refer to the same object. These constraints are called multiplicities of the association. One (at most) of the participating classes in an association can be designated to be composite. This means that an instance of that class is actually composed by the instances connected to it by a link through this association. The semantic impact of such construct is that, in general, actions over the composite objects (e.g. object destruction) are propagated to its parts. The composition relationship is transitive and no cycles are allowed. This means that an object may not be ultimately part of itself.
Another important concept is generalization. A generalization is a reflexive, antisymmetric and transitive relation between classes denoted by symbol $<:$. When a class c is generalized by a class c', class c incorporates properties of class c', particularly, attributes (provided that clashes are avoided) and partic-

ipation in associations. The intuition behind this idea is that instances of class c can be subsumed under instances of class c', and therefore also exhibit their properties. The mechanism by which a class is able to incorporate properties of other classes is called inheritance.
In conclusion, the structure of objects and their connections (system structure) constitutes the static view and is described by means of classes and associations. A model of a sample system structure is shown in Figure 2.1. On the other hand, the runtime structure of a system (system state) is structured as a set of connected objects. A pair of possible states corresponding to the system structure of Figure 2.1 are shown in Figure 2.2.

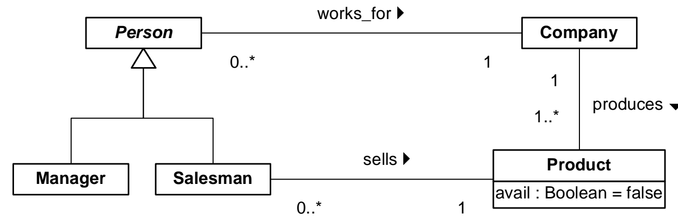

Figure 2.1: UML diagram modeling a system structure

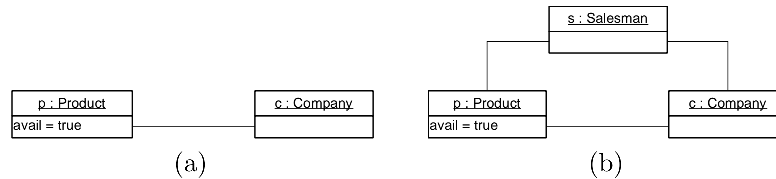

Figure 2.2: Possible system states

Throughout this work we use the example above to illustrate the concepts introduced. The example is discussed in more detail in section 2.2.3 and is developed in the following chapters.

2.1.3 State Well-formedness

In a running system, not every possible state configuration may be desirable. The notion of well-formedness allows distinguishing the desired configurations from the undesired. In general terms, well-formedness involves both syntactic and semantic aspects. In UML, a state is said to be well-formed if it satisfies all *predefined* and *model-specified* rules or constraints [RJB98]. Predefined rules and constraints are those imposed by the UML metamodel (e.g. an attribute may not realize a node, or an operation may not be an instance of an actor). States violating predefined rules and constraints are unacceptable. In turn, model-specified rules and constraints include system structure and invariants.
As discussed in the previous section, actual state configurations are described

statically by the system structure. This structure specifies what classes of objects are admissible along with their attributes, and how they are related. States not conforming to the system structure are also unacceptable. Static structure includes notion of multiplicity. A multiplicity is an expression that defines a subset of the nonnegative integers different from $\{0\}$, which is attached to an association end. It constrains the number of instances of the class at the specified end that can be linked to the same set of instances of the other participating classes. For example, given an association between classes c_1, c_2 and c_3, the number of instances of c_3 which may be linked to a particular pair of instances of classes c_1 and c_2 must be included in the set defined by the multiplicity at the end corresponding to c_3. Stable states should satisfy all the multiplicities, but it is often necessary for reaching a stable well-formed state that some intermediate ill-formed states are momentarily traversed. For this reason, these kind of semantic constraints cannot be structurally enforced, rather they are conditions that can be checked at any point in time.

A similar situation occurs with invariants, in fact, multiplicities can be understood as a particular case of these. Invariants are general conditions that must be satisfied at all times, and quoting [RJB98], "or, at least, when no operation is incomplete". This means that, as with loop invariants, system invariants are allowed to be momentarily broken during the execution of an operation, but they must be reestablished at its completion.

In conclusion, state well-formedness involves both syntactic and semantic aspects, and system state modification primitives affect both of them. As some primitives introduce new elements or modify existing ones in a state, it could be possible that a state resulting from state change violates syntactic or semantic constraints. For example, an instance of a nonexisting class could be erroneously created, or a state change may break an invariant or multiplicity. Well-formedness concerning syntactic aspects is usually enforced by construction and is intended to be permanent. Well-formedness concerning semantics aspects are just checked at some specific points at runtime, usually at the completion of higher level operations. An example in the next section will illustrate this point in more detail.

2.2 System State Modification Primitives

System state modification primitives are primitive operations that perform basic modifications to a system state at a conceptual level. Operations that have effect on the state of an object-oriented module at any level of granularity (system, subsystem, component, etc.) can be represented as a sequence of primitive changes on states. In this section we motivate a set of five primitives already identified in the bibliography, and informally describe their semantics. A final example illustrates the use of the primitives and some issues concerning state well-formedness.

2.2.1 Motivation

After the brief structure-oriented review of systems in the previous section, we motivate the set of primitives to be addressed in this work. We proceed by investigating what can and what cannot be changed in a system runtime configuration (state). A possible change in a state would suggest the existence of a system state modification primitive:

- New objects can be added to the state.
- Existing objects can be removed from the state.
- New tuples representing links can be added to the state.
- Existing tuples representing links can be removed from the state.
- The structure of a tuple cannot be changed (i.e. associations cannot be changed), that is, the number and type of objects it refers to are fixed.
- The structure of an object cannot be changed (classes cannot be changed), that is, the number and type of slots is fixed [1].
- The values held in slots can be replaced by new values of the proper type.

Adding a new object is done by a primitive named create, and removing an existing one is done by destroy. Links can be added and removed with link and unlink respectively. The value of an attribute can be changed with set. These five primitives match those identified in [Oli99] and [Ric02], and they form the set of primitives that will be the subject of our formalization. Next, we informally describe each one of them.

2.2.2 Informal Semantics of Primitives

This section introduces an informal description of the primitives listed above as a foundation for the specification in the next chapter. The main references are the UML Action Semantics Specification [OMG03c, p. 2-199], the UML Reference Manual [RJB98], and research in [Vig03].

Object Creation

The create primitive allows the instantiation of a class (i.e. creation of a new object), and adds the resulting object to the system state. It needs a unique name for the object to be created, as well as the name of the class to be instantiated. Thus, the chosen name for the object may not already be in use for an object already in the system state, and the class from whom that instance will be created must be an existing class in the system structure, and it may not be abstract. After the primitive is performed, the system state owns a new

[1] UML supports the concepts of multiple classification and dynamic classification, but they are not widely used due to a general lack of support in programming languages

instance of the specified class. The UML Action Semantics [OMG03c, p. 2-271] specifies that after the execution of an instance of the CreateObjectAction class no attribute values are set for the created instance (the default values for its attributes), that is, no constructor executes. The UML Reference Manual [RJB98, p. 307] is consistent with this approach, but distinguishes two stages in the actual creation of an object; first, the allocation of the new instance in the environment of the system (instantiation), and second, the initialization of its attributes (initialization), which occurs immediately after the instantiation. It also warns that an object instantiated but not initialized (called raw instance) might be inconsistent and "is not available to the rest of the system until it has been initialized". We expect an instance to be available after creation, so for practical reasons, we decided to assume that the create primitive both instantiates a class producing an instance, and initializes it with default values.

Object Destruction

With the destroy primitive an object can be removed from the system state. This primitive only needs the name of the object to be removed, and requires that an object with that name exists in the system. After the primitive is executed, the specified object is no longer available in the system. This implies that any link previously involving the object is also removed from the system state. Moreover, any composing object should be also recursively removed.

Link Creation

The link primitive allows establishing a connection between a tuple of objects. The ability to connect the tuple of objects must have been declared in the system structure, thus there must be an association between the classes of objects (or any of its ancestors in order $<:$) we wish to connect. For the execution of this primitive it is required that the objects to connect exist in the system state, and since the extent of an association (i.e. the collection of connections between instances of the associated classes) is a set, the objects to be connected must not be already connected through this association. After the primitive is completed, there is a new link between the tuple of specified objects through the specified association.
Finally, the case in which the association is an association class would require the creation of a special kind of object, which is also a link. Since it is a link, the semantics shown right above still holds, and since it is an object it is also needed to initialize its attributes as discussed for create.

Link Destruction

Using the unlink primitive a link between objects through a specific association can be removed. This primitive requires that the specified association exists in the system structure and associates the classes of the specified objects (or any of its ancestors). It is also required that the objects to disconnect exist in

the system state and are already linked through the specified association. After completion, the link between the specified tuple of objects through the specified association is removed.
The case of an association class would require removing an element that is both a link and an object. The semantics above still holds for the "link" part of the element. For the "object" part, object destruction (i.e. eventual link and components destruction) apply as in destroy.

Attribute Value Replacement

With the set primitive the value of an attribute of an object can be changed. It takes as input the target object, the attribute, and the new value for it. It is required that the target object exists in the system state, and the specified attribute is defined in the target object class (or any of its ancestors). Also, the type of the new value must conform to the type of the attribute (i.e. either both types match or the type of the value is a subtype of the type of the attribute). After the primitive is completed, the object holds the new value for the attribute.

2.2.3 A Simple Example

In this section we use a very simple example to illustrate the use of system state modification primitives and state well-formedness issues. In this example we use the system structure shown in Figure 2.1. We have classes Company, Product, Person, Manager and Salesman. Class Company is associated to class Person. That means that persons work for companies. Class Manager and class Salesman are related by generalization with class Person, meaning that every manager and salesmen are persons, and that they inherit every property of persons (i.e. the participation in the works_for association). Then managers and salesmen work for companies. We also have class Product. This class is associated to Company and to Salesman (by associations produces and sells respectively). This models the fact that companies produce products, and salesmen sell them. Class Product has a boolean attribute avail which represents whether a product is available or not. By default products are not available. From multiplicities we can know that persons work for exactly one company, and that companies may have any number of employees. Also, companies produce at least one product, and a product is produced by exactly one company. Finally, a product can be sold by any number of salesmen, and salesmen only sell exactly one product. We do not define invariants for this system at this moment.
We now consider the system states of Figure 2.2. First, we use primitives to reach state (a) from a state where no object exist as follows:

1. create the company with create(c,Company)
2. create the product with create(p,Product)
3. connect the company with the product with link(c,p,produces)
4. make the product available with set(p,avail,true)

Note that the order assigned to each invocation is incidental. The only restrictions are that link needs both objects, and set needs the product. Any other ordering satisfying the above restrictions is also acceptable.
We now consider the transition from state (a) to state (b). The modification depicted is not atomic, and is carried out in a number of steps. It involves the creation of a new salesman s, and connecting it to the existing company and product. This can be done as:

1. create(s,Salesman)
2. link(c,s,works_for)
3. link(s,p,sells)

In the resulting state of this program (i.e. resulting from step 3), there is a new salesman s linked to company c through the works_for association, and linked to product p through sells, as it was expected. Multiplicities are also satisfied. However, after step 1 and even after step 2 this was not the case.
In the intermediate state resulting from step 1, the multiplicities of both associations works_for and sells, at the end of Company and Product respectively, were not satisfied. In fact, both constraints require that the created salesman must be connected to exactly one company, and also to exactly one product. As salesman s is not connected to a company or a product, those multiplicities are not satisfied, and therefore the state is ill-formed. However, it is easy to see that it is not possible to avoid that situation. A salesman must be created before it can be connected to a company and a product, thus there will always be an intermediate state where the salesman exists but it is not yet connected. A similar situation occurs with the state resulting from step 2, where despite being connected to a company, the salesman is not connected to a product. For this reason, as discussed in the previous section, it is unacceptably restrictive to demand well-formedness on every intermediate state. Resulting states of system operations or sequences of meaningful state changes are expected to be well-formed, however it is admissible for some cases that intermediate states may not satisfy invariants or multiplicities. The responsibility of ensuring well-formedness on final states, as well as handling ill-formed intermediate ones, relies on the programmer. How system state modification primitives affect state well-formedness, particularly multiplicities and invariants, is explored in Chapter 4.

2.3 Program Specification and Correctness

In this section we review the technique on which most of the approaches for object-oriented program specification are based. We also explore the notion of program correctness which is closely related to that technique. An application of the formal semantics of system state modification primitives, which will be addressed in Chapter 4, is here introduced.

2.3.1 The Axiomatic Approach

According to UML terminology, the term *behavior* means semantics of an operation [RJB98]. In object-oriented development the most popular approach to specifying behavior on a rigorous basis is a technique called *Software Contracts*. Contracts are directly derived from the axiomatic approach introduced by C.A.R. Hoare [Hoa69] for proving properties of programs, but also for specifying the semantics of imperative programming languages [Win93]. According to that approach, a program is specified using a formula of the form $\{ P \} S \{ Q \}$, where S is the statement (or program) being specified, and P and Q are assertions on the state. There, P is called the *precondition* and Q is called the *postcondition*. The operational interpretation of such formula is [NN92]:

> if P holds in the initial state, and
> if the execution of S terminates when started in that state,
> then Q will hold in the state in which S halts.

This kind of formulae is called a *partial correctness assertion* because it need not to ensure that program S terminates.

Precondition P and postcondition Q alone are called a *specification* for S. Based on the interpretation given above, they express the effect of S on the state. A specification can be used by the programmer for describing his intentions about the program, but also by a user for making sure that the program fits his needs. Whether a specification is useful or not for those purposes highly depends on the way in which assertions are formulated. Assertions expressed informally or based on a poorly defined notion of state are not of much use. Instead, precise assertions about a well defined state both fulfill the goal and are the basis for more advanced techniques such as proving program correctness.
Given a specification and an implementation of a program, it could be possible to prove its *correctness*. Particularly, correctness of the implementation with respect to the specification, again in the sense of the interpretation above. If it is possible to build a proof for a correctness formula $\{P\}S\{Q\}$, then the implementation of S satisfies the specification. Such a proof can be constructed by applying rules specifically defined for the programming language used to implement S. Rules are axiomatizations of the meaning of basic constructs of the programming language; they can be understood as proved correctness assertions for programs each one consisting of just a simple construct. Then, the rules can be combined to build a proof tree for composite and more complex programs. Again, a well defined set of rules is necessary for a proof to be formally constructed.

2.3.2 Application to Object-Orientation

Several proposals applied the ideas discussed in the previous section to object-oriented development, with variable degree of formalism and different purposes. They were first applied to object-oriented development by Bertrand Meyer who introduced a technique called *Design-by-Contract* [Mey92]. This technique suggests a way of thinking about object interactions with native support in the Eiffel [Mey91] programming language. Eiffel allows the programmer to annotate class level operations with assertions (pre- and postconditions), which are evaluated by the runtime environment before and after the execution of the operation respectively. A similar approach for the Java programming language [JSGB00] is being developed in the Java Modeling Language (JML) [LC03] project.
Contracts are also applied in [Lar98] as a tool for specifying system level operations. There, assertions are written in natural language, and in particular, postconditions are expressed in terms of the five primitives discussed earlier in this chapter. Another application of contracts can be found in [DW99, CD01]. In this case they are used for specifying component level operations, and assertions are expressed using OCL [OMG03a, WK98].
Table 2.1 summarizes the applications of correctness assertions reviewed before. They are classified by the level of the operation they are applied to, the purpose of the application (proof of correctness, execution of runtime tests and specification), the language used for expressing assertions, and the programming language used for implementing the operation.

Approach	Level	Main Purpose	Pre-/Post-	Lang
Hoare	Program	Spec/Correctness	Logic	Algol-like
Meyer	Class	Spec/Runtime test	Eiffel	Eiffel
JML	Class	Spec/Runtime test	JML	Java
Catalysis	Component	Specification	OCL	-
Larman	System	Specification	NL	-

Table 2.1: Applications of correctness assertions

In the last two rows the programming language is left unspecified because in those approaches contracts are written in an analysis phase for specification purposes only, and at that time the operation is not yet implemented.
In conclusion, software contracts, which were originated from correctness assertions, are the most popular technique for specifying behavior in object-oriented systems. It should be noticed that specification purposes are present in all of the approaches reviewed. However, the purpose of proving correctness is not. In that context, as pointed out before, assertions need a formal foundation, and also a set of inference rules for the programming language is required. A group in Nijmeegs Instituut voor Informatica en Informatiekunde [SoS03] is developing the LOOP tool [vdBJ01]. This tool is used for reasoning about small sequential Java programs annotated with JML specifications. It translates the program to its semantics in higher order logic, acting as a front-end to a theorem prover

(such as PVS [ORS92] and Isabelle [Pau94]) in which the actual proofs are built. Translation of JML specifications is not yet available in this on-going project.
The application of the JML approach takes place in the implementation phase of the development process. Approaches focusing on analysis and early design phases are unable to reason in a similar way for the simple fact that no implementation exists at that stage. However, this missing ingredient could be conceptually represented using system state modification primitives. Then, reasoning in an early phase of software development would become possible, providing (hopefully) valuable feedback to design and the definitive implementation phase. In this context, the semantics of system state modification primitives can be used to develop a theory in which formal proofs of correctness of programs can be built. This task is addressed in Chapter 4.

2.4 Proof Systems and Proof Assistants

Different approaches can be applied to theorem proving. Proof systems provide a framework where propositions (and all necessary notions) can be formally expressed, and unambiguous proofs for them can be constructed and verified by simple syntactic checking. Building and verifying formal proofs is a highly complex process, usually too complex to be carried out by hand. Proof assistants are computer programs which help users handling formal proofs.
In this section we overview the characteristics of proof systems, discuss the steps for building reliable proofs, and review the main features of proof assistants. The section ends up explaining how the System Coq has been used in this work. The information presented here is mostly based on a work by Bruno Barras [Bar99] and the Coq Reference Manual [CDT02].

2.4.1 Proof Systems and Formal Proofs

A reliable approach to theorem proving is the use of logic. The main goal of logic is to describe as rigorously as possible what a proof of a theorem is. A *proof system* provides a formal language and a set of rules. Propositions and proofs can be unambiguously expressed using the formal language. In turn, deductions are limited to the application of the well defined set of rules. Formal proofs are then reliable because it is possible to verify if things were done according to the rules.
There are three major steps in the construction of formal proofs. The problem statement, the construction of the proof itself, and (optionally) the verification of the proof. The first step is called *formalization*. It involves the translation of the problem into a logic formula of the logical system, that is, expressing the proposition using the provided formal language. The second step is the *proof construction*. Here, the structure of the proof is built and each step of deduction must be justified by a rule. The last step is the *proof checking*. Checking a proof involves the verification that the proof was actually constructed according to the rules, it reduces to a syntactic check of the proof structure. Provided that the

formalization was carried out properly, if the resulting formula can be proved in the logical system, then the proposition is a theorem of that system. In addition, a thorough verification increases the reliability of the proof.
In summary, a proof system provides a framework where reliable proofs can be built. However, for humans, the construction of complex proofs and their verification turns to be overwhelming and error prone. In this context, computer programs are very helpful for performing those tasks.

2.4.2 Proof Assistants

Proof assistants are computer programs which help users of a proof system in the construction and verification of a proof. Some examples of such systems are PVS, Isabelle and Coq. They are usually composed by two tools: a proof checker and a proof assistant. A *proof checker* can automatically tell the user whether a proof of a theorem is in fact correct. In turn, a *proof assistant* provides the user with commands which he or she can use to tell the system the actual structure of the proof. This hides much of the complex details of the construction, allowing the user to focus on the architecture of the proof.
An important characteristic of these systems is that they can usually separate the stages of construction and validation of a proof. In general, a program which takes as input a proof and which answers if it is correct is much like a parser, and thus it is relatively simple to write and validate. This program is the essential component of what is called the *core* of the system. Around this core, more or less complex functions allow the user to conceive and build a proof. The key point is that the reliability lies on the core, and not on these functions.

2.4.3 The Role of Coq

The system Coq is a proof assistant based on Type Theory. More precisely, its logical system is the Calculus of Inductive Constructions [Wer94, PM97], which is a λ-calculus with a rich type system. All logical judgments in Coq are typing judgments. The core of the Coq system is the type-checking algorithm (proof checker) that mechanically checks the correctness of proofs. Coq also provides an interactive front-end (proof assistant) for building proofs using specific programs called *tactics*.
In this work the Coq system is used for the following purposes:

1. **Formalization.** Every basic concept as well as every proposition is encoded using Coq's formal language (Calculus of Inductive Constructions). This provides the ultimate meaning for every element in our theory which is semi-formally introduced in Chapters 3 and 4.

2. **Proof construction.** Coq's front-end is used as an environment which assists the construction of proofs for every proposition.

3. **Proof verification.** Every generated proof is verified mechanically by the Coq's type-checking algorithm.

Summary

In this chapter an overview of object-oriented systems and system state modification primitives was given. Also, a natural application of the primitives was presented, and the role of a proof system in this work was explained.

System state modification primitives describe at a conceptual level basic changes that can be applied to the state of an object-oriented system. A possible implementation of higher level operations can be represented or described in terms of these primitives. A formalization of the semantics of the primitives can be then used as a mean to reason about their use. This allows the formal proof of correctness of such program descriptions with respect to a specification, as well as other properties on the resulting state. These activities can be carried out in an early phase of the development process, providing feedback to later phases.

The formalization of both semantics and proofs can be carried out using a proof assistant, particularly Coq. The system Coq provides a formal language to express our specification, a proof assistant for a computer-assisted proof construction, and most importantly a proof checker which mechanically validates the correctness of every generated proof.

In the next chapter we start our specification of system state modification primitives, as well as other required notions such as the state of a system.

Chapter 3

Specification of System State Modification Primitives

In this chapter, the semantics of the already introduced *system state modification primitives* is specified. Programs which provide methods to higher level operations, such as system operations, can be represented at early stages of the development process as sequential compositions of these basic operations. Such representations could be useful for providing some input to system design, and even for code generation in the implementation phase. Since the primitives primarily affect the state of object-oriented systems, for a specification of their semantics, a clear definition of what we understand by *system* and *state* is mandatory. We define types for system and state notions with a high level of abstraction. The primitives are specified in terms of pre- and postconditions instead of introducing particular algorithms operating on concrete data structures.

This chapter is structured as follows. Section 3.1 identifies the set of types needed to specify the semantics of system state modification primitives. It also discusses the application of abstract data types as a foundation for our specification. The specification starts in section 3.2, in which we specify all the basic types which are necessary to define types System and State. In section 3.3 the specification of type System is presented, and section 3.4 proceeds with type State. The system state modification primitives are specified as operations associated to that type. This chapter concludes with a discussion on possible implementations of the abstract types here introduced.

3.1 Types and Specification Approach

In this section we identify a set of concepts for describing the logic structure of object-oriented systems. These concepts are the types which drive our specification. We then overview the Abstract Data Types approach, and briefly describe its application to the specification of system state modification primitives.

3.1.1 Structural Concepts

We now proceed with the identification of the types which are necessary for specifying the semantics of the system state modification primitives. In combination, all these types represent the structural concepts which are the basis of our specification.
We organize our specification around two core types, State and System. The concept of system state plays a vital role in our specification since the primitives operate directly on it. Instances of type State represent the runtime configuration of a system at a particular moment in time. The concept of system structure is also important because it defines the structure of states, and affects how the primitives manipulate them. An instance of type System represents the static structure of a system. Different states of the same system are described by the static structure of that system. Therefore, an instance of type State needs to be related to the instance of type System which describes it. This relation will be discussed later on. Table 3.1 summarizes the basic types which were selected and specifies the core type to which they are associated.

Basic type	Associated to
Association	System
Attribute	System
Class	System
Multiplicity	System
Object	State
Type	System
Value	State

Table 3.1: Basic types associated to types State and System

The selection of the concepts above is based on the concepts reviewed in section 2.1.2 and is aligned with UML terminology [OMG03c]. UML elements for modeling the static structure of a system are defined in the Core package of the UML metamodel [OMG03c, pp. 2-12]. In turn, model elements for modeling runtime structure are defined in the Common Behavior package [OMG03c, pp. 2-93]. For us, the structure of a system is defined by a set of classes and the association relationships between them. Classes have typed attributes, and multiplicities are attached to associations. We do not handle generalization relationships explicitly. A system state is defined by a set of objects and the values they hold for their attributes. Links between objects are not handled explicitly either.

A number of restrictions are applied, for reasons of clarity and in some cases for simplicity as discussed next. Particularly:

- Associations are assumed to be binary.
- Association classes are not considered.
- Compositions are not considered (an OCL-based semantics of system state modification primitives considering compositions can be found in [Vig03]).
- We assume a multiplicity to be a subrange of the set of the nonnegative integers.
- We do not support multiple classification (i.e. objects are created from exactly one class).
- Attributes hold exactly one value at a time, that is, we assume that attribute multiplicities are 1.

The restrictions listed above do not represent an important loss in expressiveness, and with the unique exception of compositions, the do not show either a great impact on the semantics of the primitives. The binary form is the most widely used variant of associations. In fact, n-ary associations are not supported in Meta Object Facility (MOF) [OMG02], which is a basic subset of UML, and is the language used for expressing the UML metamodel. Association classes were not considered as a relevant structural concept in [Ric02]. According to the UML Reference Manual [RJB98], a multiplicity is most often a single interval of natural numbers with a minimum and a maximum value. Attributes commonly hold a single value. However, multi-valued attributes can be defined by simply using collection types [Ric02]. As discussed in section 2.2.2, compositions do play an important role in the semantics of the primitives, especially for destroy. When destroying an object, every (either direct or indirect) component object should be also destroyed. This suggests two different levels of behavior. In the lower level, where no composition is defined or considered, destroy involves a simple destruction of the target object and also the destruction of all the links in which it participates. In the higher level, where compositions are applied, that primitive potentially involves recursive object destruction. In this work we address only the former form of destroy. We specify a basic form of destroy, which can be used for defining a more sophisticated one.
Next, we overview the specification of abstract data types, which will be applied in the remaining sections of this chapter.

3.1.2 Abstract Data Types Specification

The specification presented in this chapter is based on abstract data types (ADTs). An abstract data type can be understood as a algebraic model of a set of entities [AHU83]. No assumption on the actual representation of the model is made, and the entity is entirely defined by the set of associated operations. The ADT approach focuses on the essential properties of the entity

being specified, rather than on its internal structure. Varying representations of the entity are allowed, avoiding the complexities and particular details of concrete data structures. This feature is very important to us, since it enables a specification possessing a high level of abstraction, which results to be fairly simple and compact. In section 3.5 we discuss other existing specifications and the impact of their use on our specification of the semantics of the primitives.
For the core types State and System, and for type Multiplicity we define abstract data types following the style applied in [Mey97]. An ADT specification is basically structured in four sections: the first is named *Functions* which lists the operations applicable to instances of the ADT. For convenience, we organize functions into groups: *creators* are operations which produce instances of the ADT from instances of other types; *queries* are operations which yield properties of instances of the ADT, expressed in terms of instances of other types; *commands* are operations which yield instances of the ADT from existing instances of it (and possibly instances of other types); and *extensors* are operations which are usually queries, and are not primitive operations of the ADT (they are derivable from others). The second section is named *Preconditions* and consist of a number of predicates which define the domain of the operations that are partial functions. Preconditions are of the form: *func* requires *cond*, which means that an application to *func* only makes sense if *cond* is satisfied. The last section is named *Axioms* which implicitly specifies the semantics for each function, by stating properties in the form of predicates on their possible values. System state modification primitives are then specified as commands associated to type State. Some operations of an ADT are used in the definition of other operations. These are the *basic operations* of the ADT. Basic operations are properly identified in each ADT specification.
Finally, at this stage of the specification, with the unique exception of type Multiplicity, the basic types are informally specified as types whose instances are just names. Their ultimate meaning, along with a concrete semantics for partial functions denoted by a hooked arrow ($\hookrightarrow$) in ADTs specifications, is given in Chapter 5, where the formalization of the entire specification in the Calculus of Inductive Constructions is addressed.
In the rest of this chapter we introduce types in a bottom-up fashion, starting with the basic types and specifying types System and State at last.

3.2 Specification of Basic Types

In this section we introduce a first part of our specification. Here we present the basic types that are needed for the specification types System and State.
For types *Association*, *Attribute*, *Class*, *Object*, and *Type* we only assume that instances of these types are disjoint names. Types are associated with attributes, and are used for specifying the domain of admissible values held by objects. Our specification does not include operations on concrete types such as *Integer* or *String*. However, if needed, operations over types (instances of Type) could be specified and seamlessly integrated to our specification. We assume *Value* as a

family of sets indexed by *Type*, and a function `hasType` which tests whether a value corresponds to a type:

`hasType :  Value`$\times$`Type`$\rightarrow$`Boolean`

In what follows, we specify an abstract data type for type *Multiplicity*. A multiplicity is a (potentially infinite) subrange of the natural numbers, defined by its lower and upper bounds. For this ADT specification we introduce a special type *UnlimitedNatural*, whose values are natural numbers augmented with the special value '∞':

`UnlimitedNatural` $\triangleq$ `Natural +` $\{\infty\}$

This value is greater that every natural number, that is, $(\forall n : natural)(n < \infty)$. Then a multiplicity of the form (1,10) denotes natural numbers between 1 and 10, and (0,∞) denotes the whole set of natural numbers. The upper bound must be grater than zero. If the upper bound was zero, no links through the association would be allowed, since the multiplicity constrains the number of admissible links. An association without links is no use. The creator operation `newMultiplicity` is the basic operation of this ADT.

Functions

Creators
`newMultiplicity :  Natural`$\times$`UnlimitedNatural`$\hookrightarrow$`Multiplicity`
-- creates a new multiplicity
Queries
`min :  Multiplicity`$\rightarrow$`Natural`
-- returns the lower bound of the multiplicity
`max :  Multiplicity`$\rightarrow$`UnlimitedNatural`
-- returns the upper bound of the multiplicity
Extensors
`inRange :  Natural`$\times$`Multiplicity`$\rightarrow$`Boolean`
-- tests if a value is in the range of the multiplicity

Preconditions

PreMul1 The minimum may not be greater than the maximum, which in turn must be greater than zero:
$newMultiplicity(m, M)$ requires $m \leq M \wedge M > 0$

Axioms

AxMul1 The lower bound of a multiplicity equals the value of the first argument used for its creation:

$min(newMultiplicity(m, M)) = m$

AxMul2 The upper bound of a multiplicity equals the value of the second argument used for its creation:
$max(newMultiplicity(m, M)) = M$

AxMul3 A value is in range if it is greater or equal to the lower bound, and it is less or equal to the upper bound:
$inRange(n, m) \Leftrightarrow (min(m) \leq n \wedge n \leq max(m))$

3.3 Specification of type System

In this section we introduce an ADT specification for type System. A system has the following properties:

- A collection of classes
- A collection of typed attributes for each class
- A collection of associations with multiplicities
- A generalization hierarchy over classes

In this specification the hierarchical organization of concepts concerning the system structure as discussed in section 2.1.2 is flattened. We specify all those concepts as direct properties of System. For example, an attribute is a property of a system and is associated to classes in it, instead of being a property of a class.

3.3.1 Functions

Properties of a system are exposed through the following set of operations. Since a system is meant to be static, a property may not be removed, and thus we only provide operations for adding properties to a system. A class may be specified as abstract. An abstract class may not be instantiated, and is included in a system for other classes to inherit its associated attributes and association relationships. A default value is specified for every attribute. The notion of ancestor of a class is specified as the set of its related classes (transitive closure) over the generalization hierarchy, therefore representing relation $<:$.

Creators

`newSystem :  System`
 - - creates an empty system

Queries

`existsClass : Class×System→Boolean`
  - - tells if certain class exists in a given system
`isAbstract : Class×System↪Boolean`
  - - tells if a class is abstract
`isSubclass : Class×Class×System↪Boolean`
  - - tells if one class is subclass of another in the system
`isAttribute : Attribute×Class×System↪Boolean`
  - - tells if an attribute is a feature of a class in the system
`getAttributeType : Attribute×Class×System↪Type`
  - - returns the type of the attribute of the class in the system
`defVal : Attribute×Class×System↪Value`
  - - returns the default value of an attribute of a class
`existsAssociation : Association×System→Boolean`
  - - tells if an association exists in the system
`associates : Association×Class×Class×System↪Boolean`
  - - tells if an association associates a pair of classes in the system
`multiplicities :`
  `Association×Multiplicity×Multiplicity×System↪Boolean`
  - - tells if the association has the specified multiplicities in the system

Commands

`addClass : Class×Boolean×System↪System`
  - - returns a new system containing the new class (abstract or concrete
  - - as specified by the boolean argument)
`addAssociation : Association×Class×Class×`
  `Multiplicity×Multiplicity×System↪System`
  - - returns a new system containing the new association between the
  - - specified classes with the specified multiplicities
`addGeneralization : Class×Class×System↪System`
  - - returns a new system containing a generalization relationship
  - - between the specified classes
`addAttribute : Attribute×Type×Value×Class×System↪System`
  - - returns a new system in which the specified class has a new
  - - attribute of the specified type with a default value

Extensors

`isAncestor : Class×Class×System↪Boolean`
  - - tells if a class is an ancestor of another in the system

3.3.2 Preconditions

We now specify preconditions for every partial function in the previous section. Classes and associations may not be duplicated. An attribute may not

be associated more than once with the same class, but it can be associated to different classes as long as they are not related by generalization. This prevents the occurrence of clashes caused by a class inheriting repeated attributes. Since generalization is an antisymmetric relation, cycles may not be formed.

PreSys1 A class must exist for being abstract:
$isAbstract(c, S)$ requires $existsClass(c, S)$

PreSys2 Two classes must exist for one to be subclass of the other:
$isSubclass(c_1, c_2, S)$ requires $existsClass(c_1, S) \wedge existsClass(c_2, S)$

PreSys3 A class must exist for owning an attribute:
$isAttribute(a, c, S)$ requires $existsClass(c, S)$

PreSys4 A class must exist and an attribute must be owned by the class for retrieving its type:
$getAttributeType(a, c, S)$ requires $existsClass(c, S) \wedge isAttribute(a, c, S)$

PreSys5 A class must exist and an attribute must be owned by the class for retrieving its default value:
$defVal(a, c, S)$ requires $existsClass(c, S) \wedge isAttribute(a, c, S)$

PreSys6 The association in question and two classes must exist for these to be associated:
$associates(a, c_1, c_2, S)$ requires
$existsAssociation(a, S) \wedge existsClass(c_1, S) \wedge existsClass(c_2, S)$

PreSys7 An association must exist for having multiplicities:
$multiplicities(a, m_1, m_2, S)$ requires $existsAssociation(a, S)$

PreSys8 A class may not be added twice to a system:
$addClass(c, b, S)$ requires $\neg existsClass(c, S)$

PreSys9 An association may not be added twice to a system:
$addAssociation(a, c_1, c_2, m_1, m_2, S)$ requires
$\neg existsAssociation(a, S) \wedge existsClass(c_1, S) \wedge existsClass(c_2)$

PreSys10 Two classes must exist for establishing a generalization between them. They may not already be in a generalization, a cycle may not be formed, and attribute clashes may not occur:
$addGeneralization(c_1, c_2, S)$ requires
$existsClass(c_1) \wedge existsClass(c_2) \wedge \neg isSubclass(c_1, c_2, S) \wedge$
$\neg isAncestor(c_1, c_2, S) \wedge$
$(\forall a : Attribute \mid isAttribute(a, c_1, S); \forall c : Class \mid isAncestor(c, c_1, S))$
$(\neg isAttribute(a, c, S))$

PreSys11 A class must exist for adding an attribute to it. The attribute may not be already owned by the class nor by any ancestor of it. The default value must be an instance of the specified type:

$addAttribute(a, t, v, c, S)$ requires
$existsClass(c, S) \wedge hasType(v, t) \wedge$
$(\nexists c' : Class \mid isAncestor(c', c, S) \wedge isAttribute(a, c', S))$

PreSys12 Two classes must exist in a system for one to be ancestor of the other:
$isAncestor(c1, c_2, S)$ requires $existsClass(c_1, S) \wedge existsClass(c_2, S)$

3.3.3 Axioms

In this section, functions declared in section 3.3.1 are specified. The basic operations in this ADT specification are the command operations: `addClass`, `addAttribute`, `addAssociation` and `addGeneralization`.

AxSys1 In a new system no classes exist:
$(\forall c : Class)(\neg existsClass(c, newSystem())$

AxSys2 A class exists in a system iff it was previously added:
$existsClass(c, S) \Leftrightarrow$
$(\exists S' : System \mid S = addClass(c, b, S')) \vee$
$(\exists S' : System \mid existsClass(c, S') \wedge$
$(S = addClass(c', b, S') \vee$
$S = addAssociation(a, c_1, c_2, m_1, m_2, S') \vee$
$S = addGeneralization(c_1, c_2, S') \vee$
$S = addAttribute(a, t, v, c', S'))$

AxSys3 A class is abstract in a system iff it was previously added using a *true* argument:
$isAbstract(c, S) \Leftrightarrow$
$(\exists S' : System \mid S = addClass(c, \text{true}, S')) \vee$
$(\exists S' : System \mid isAbstract(c, S') \wedge$
$(S = addClass(c', b, S') \vee$
$S = addAssociation(a, c_1, c_2, m_1, m_2, S') \vee$
$S = addGeneralization(c_1, c_2, S') \vee$
$S = addAttribute(a, t, v, c', S'))$

AxSys4 A class is subclass of another class iff they are the same class or a generalization between them was previously added:
$isSubclass(c_1, c_2, S) \Leftrightarrow$

$(c_1 = c_2)\vee$
$(\exists S' : System \mid S = addGeneralization(c_1, c_2, S')))\vee$
$(\exists S' : System \mid isSubclass(c_1, c_2, S')\wedge$
$(S = addClass(c', b, S')\vee$
$S = addAssociation(a, c'_1, c'_2, m_1, m_2, S')\vee$
$S = addGeneralization(c'_1, c'_2, S')\vee$
$S = addAttribute(a, t, v, c', S'))$

AxSys5 An attribute is a feature of a class iff it was previously added to it:
$isAttribute(a, c, S) \Leftrightarrow$
$(\exists S' : System \mid S = addAttribute(a, t, v, c, S')))\vee$
$(\exists S' : System \mid isAttribute(a, c, S')\wedge$
$(S = addClass(c', b, S')\vee$
$S = addAssociation(as, c_1, c_2, m_1, m_2, S')\vee$
$S = addGeneralization(c_1, c_2, S')\vee$
$S = addAttribute(a', t, v, c', S'))$

AxSys6 The type of an attribute of a class is the one specified when the attribute was added to the class:
$t = getAttributeType(a, c, S) \Leftrightarrow$
$(\exists S' : System \mid S = addAttribute(a, t, v, c, S')))\vee$
$(\exists S' : System \mid t = getAttributeType(a, c, S')\wedge$
$(S = addClass(c', b, S')\vee$
$S = addAssociation(as, c_1, c_2, m_1, m_2, S')\vee$
$S = addGeneralization(c_1, c_2, S')\vee$
$S = addAttribute(a', t', v, c', S'))$

AxSys7 The default value of an attribute of a class is the one specified when the attribute was added to the class:
$v = defVal(a, c, S) \Leftrightarrow$
$(\exists S' : System \mid S = addAttribute(a, t, v, c, S')))\vee$
$(\exists S' : System \mid v = defVal(a, c, S')\wedge$
$(S = addClass(c', b, S')\vee$
$S = addAssociation(as, c_1, c_2, m_1, m_2, S')\vee$
$S = addGeneralization(c_1, c_2, S')\vee$
$S = addAttribute(a', t', v', c', S'))$

AxSys8 An association exists in a system iff it was previously added:
$existsAssociation(a, S) \Leftrightarrow$
$(\exists S' : System \mid S = addAssociation(a, c_1, c_2, m_1, m_2, S')))\vee$
$(\exists S' : System \mid existsAssociation(a, S')\wedge$
$(S = addClass(c, b, S')\vee$
$S = addAssociation(a', c'_1, c'_2, m'_1, m'_2, S')\vee$
$S = addGeneralization(c'_1, c'_2, S')\vee$
$S = addAttribute(at, t, v, c, S'))$

AxSys9 An association associates the classes specified when it was added to the system:

$associates(a, c_1, c_2, S) \Leftrightarrow$
$\quad (\exists S' : System \mid S = addAssociation(a, c_1, c_2, m_1, m_2, S')))\vee$
$\quad (\exists S' : System \mid associates(a, c_1, c_2, S')\wedge$
$\quad\quad (S = addClass(c, b, S')\vee$
$\quad\quad S = addAssociation(a', c_1', c_2', m_1, m_2, S')\vee$
$\quad\quad S = addGeneralization(c_1', c_2', S')\vee$
$\quad\quad S = addAttribute(at, t, v, c, S'))$

AxSys10 The multiplicities of an association are those specified when they were added to the system:

$multiplicities(a, m_1, m_2, S) \Leftrightarrow$
$\quad (\exists S' : System \mid S = addAssociation(a, c_1, c_2, m_1, m_2, S')))\vee$
$\quad (\exists S' : System \mid multiplicities(a, m_1, m_2, S')\wedge$
$\quad\quad (S = addClass(c, b, S')\vee$
$\quad\quad S = addAssociation(a', c_1, c_2, m_1', m_2', S')\vee$
$\quad\quad S = addGeneralization(c_1, c_2, S')\vee$
$\quad\quad S = addAttribute(at, t, v, c, S'))$

AxSys11 An ancestor is a superclass or an ancestor of a superclass:

$isAncestor(c_1, c_2, S) \Leftrightarrow isSubclass(c_2, c_1, S)\vee$
$\quad (\exists c : Class \mid isSubclass(c_2, c, S) \wedge isAncestor(c_1, c, S))$

3.3.4 Example

In this section we illustrate the use of the query operations associated to type System. Assuming that instance S of that type represents the system shown in Figure 2.1, in Figure 3.1 we show the results of some invocations to query operations on S. We show some invocations to operations associated to type Multiplicity as well. For the invocations, the following variables are used:

$Person, Company, Product, Manager, Salesman : Class$
$works_for, produces, sells : Association$
$m_1, m_2, m_3 : Multiplicity$
$avail : Attribute$
$Boolean : Type$
$ff : Value$

$existsClass(Person, S) = true$
$existsClass(Company, S) = true$
$existsClass(Product, S) = true$
$existsClass(Manager, S) = true$
$existsClass(Salesman, S) = true$

$associates(works_for, Company, Person, S) = true$
$associates(produces, Company, Product, S) = true$
$associates(sells, Salesman, Product, S) = true$
$isSubclass(Manager, Person, S) = true$
$isSubclass(Salesman, Person, S) = true$

$min(m_1) = 1 \quad max(m_1) = 1$
$min(m_2) = 0 \quad max(m_2) = \infty$
$min(m_3) = 1 \quad max(m_3) = \infty$

$isAttribute(avail, Product, S) = true$
$getAttributeType(avail, Product, S) = Boolean$
$defVal(avail, Product, S) = ff$

$multiplicities(works_for, m_1, m_2, S) = true$
$multiplicities(sells, m_2, m_1, S) = true$
$multiplicities(produces, m_1, m_3, S) = true$

3.4 Specification of type State

In this section we introduce an ADT specification for type State. A state has the following properties:

- A system which describes its structure
- A collection of objects
- A collection of values for object attributes
- A collection of links between objects

For this specification, the same approach as for type System is applied. For example, values are properties of a state and are associated with objects, instead of being properties of them. State well-formedness is partially specified in this ADT. Well-formedness with respect to invariants is defined completely outside the type State, and is discussed in the next chapter. In this specification we define well-formedness with respect to multiplicities only. In particular, we specify an operation *isWellFormed* which can be used for testing whether a particular state satisfies all defined multiplicities or not. That test can be performed whenever is desired. However, as discussed in section 2.1.3, the information provided by `isWellFormed` about a state is more useful when the state is the result of a higher level operation, than in the case of intermediate ones. The primitives are not affected by well-formedness, since by being as basic as possible they do not enforce well-formedness and thus can potentially, and in some cases inevitably, yield ill-formed states. However, in the next chapter we explore the situations in which well-formedness is preserved, gained or lost, by the application of a single primitive.

3.4.1 Functions

When created, a state is associated to a given system. Once that association is established, it cannot be changed. System state modification primitives are specified as command operations associated to this State. Query operations, particularly `existsObj`, `getVal`, `areLinked` and `isInstanceOf`, play an important role in the next chapter.

Creators

`newState : System→State`
  - - creates a new state from a given system

Queries

`isEmpty : State→Boolean`
  - - tells if a state is the empty state
`structure : State→System`
  - - returns the system that structures the state
`existsObj : Object×State→Boolean`
  - - tells if certain object exists in a given state
`getVal : Object×Attribute×State↪Value`
  - - returns the value of an attribute of a given object
`areLinked : Object×Object×Association×State↪Boolean`
  - - tells if two objects are linked together by an association
`isInstanceOf : Object×Class×State↪Boolean`
  - - tells if an object is an instance of a given class
`isWellFormed : State→Boolean`
  - - tells if a state is well-formed

Commands

`create : Object×Class×State↪State`
  - - returns a new state containing a new instance of the specified class
`destroy : Object×State↪State`
  - - removes the specified object returning the resulting state
`link : Object×Object×Association×State↪State`
  - - connects a pair of objects returning the resulting state
`unlink : Object×Object×Association×State↪State`
  - - removes the connection between a pair of objects returning the
  - - resulting state
`set : Object×Attribute×Value×State↪State`
  - - changes the attribute value for another returning the resulting state

3.4.2 Preconditions

We now specify preconditions for every partial function in the previous section. In this specification primitives preconditions are in general more restrictive than their analogues in UML Action Semantics [OMG03c, p. 2-199]. In that specification some particular scenarios are not restricted, but the semantics of the action is left unspecified. For example, the semantics of creating an object from an abstract class (i.e. having an instance of CreateObjectAction class, associated to an abstract classifier) is undefined, arguing that it is possible that someone could be able to find a meaning for that. That kind of flexibility is paid with undefinedness, which is not desirable for us. In our specification we avoid such situations by strengthening the preconditions.

PreStt1 An object must exist and an attribute must be owned by the class of the object for retrieving its value:
$getVal(o, a, s)$ requires $existsObj(o, s) \wedge$
$(\exists c : Class \mid isInstanceOf(o, c, s) \wedge$
$isAttribute(a, c, structure(s)))$

PreStt2 An association and two objects must exist, and they must be instances of the classes associated by the association for them to be linked:
$areLinked(o_1, o_2, a, s)$ requires
$existsAssociation(a, structure(s)) \wedge existsObj(o_1, s) \wedge existsObj(o_2, s)$
$\wedge (\exists c_1, c_2 : Class \mid isInstanceOf(o_1, c_1, s) \wedge$
$isInstanceOf(o_2, c_2, s) \wedge associates(a, c_1, c_2, structure(s)))$

PreStt3 An object and a class must exist for the object to be an instance of the class:
$isInstanceOf(o, c, s)$ requires $existsClass(c, structure(s)) \wedge existsObj(o, s)$

PreStt4 An object may not be created twice. The class to be instantiated must exist in the system of the state and it may not be abstract:
$create(o, c, s)$ requires $existsClass(c, structure(s)) \wedge$
$\neg isAbstract(c, structure(s)) \wedge \neg existsObj(o, s)$

PreStt5 An object must exist in the state for being destroyed:
$destroy(o, s)$ requires $existsObj(o, s)$

PreStt6 For two objects to be linked through an association, the association and the two objects must exist, they must be instances of the classes associated by the association and may not be already linked through the association:
$link(o_1, o_2, a, s)$ requires
$existsAssociation(a, structure(s)) \wedge existsObj(o_1, s) \wedge$
$existsObj(o_2, s) \wedge \neg areLinked(o_1, o_2, a, s)$

PreStt7 For two objects to be unlinked with respect to an association, the as-

sociation and the two objects must exist, they must be instances of the classes associated by the association and they must be already linked through the association:

$unlink(o_1, o_2, a, s)$ requires
$existsAssociation(a, structure(s)) \wedge existsObj(o_1, s) \wedge$
$existsObj(o_2, s) \wedge areLinked(o_1, o_2, a, s)$

PreStt8 For the attribute value of an object to be updated with a new value, the object must exist, the attribute must be owned by the object class, and the type of the value must match the type of the attribute:

$set(o, a, v, s)$ requires $existsObj(o, s) \wedge$
$(\exists c : Class \mid isInstanceOf(o, c, s) \wedge$
$isAttribute(a, c, structure(s)) \wedge$
$hasType(v, getAttType(a, c, structure(s))))$

3.4.3 Relations between States

The following relations between states are the very heart of the specification of the primitives which is introduced in the next section. Inspired by a set of equivalences between memories of Smart Cards defined in [BCST00], these relations focus on differences between states in terms of their basic properties (i.e. objects, links and attribute values). They could be included as extensor operations, but for reasons of clarity we specify them separately. For the same reason we omit preconditions such as "for every relation $structure(s) = structure(s')$ must hold".

Relations on Objects

$\sim_{obj}$, such that $s \sim_{obj} s'$ holds whenever objects in s are the same as in s':

$$s \sim_{obj} s' \Leftrightarrow (\forall o : Object)(existsObj(o, s) \Leftrightarrow existsObj(o, s'))$$

$\sim_{obj+o}$, such that $s \sim_{obj+o} s'$ holds whenever objects in s' are the same as in s but also contain object o:

$$s \sim_{obj+o} s' \Leftrightarrow \neg existsObj(o, s) \wedge existsObj(o, s') \wedge (\forall o' : Object \mid o' \neq o)(existsObj(o', s) \Leftrightarrow existsObj(o', s'))$$

$\sim_{obj-o}$, such that $s \sim_{obj-o} s'$ holds whenever objects in s' are the same as in s except for object o, which exists in s but not in s':

$$s \sim_{obj-o} s' \Leftrightarrow existsObj(o, s) \wedge \neg existsObj(o, s') \wedge (\forall o' : Object \mid o' \neq o)(existsObj(o', s) \Leftrightarrow existsObj(o', s'))$$

Relations on Links

$\sim_{link}$, such that $s \sim_{link} s'$ holds whenever links in s are the same as in s':

$$s \sim_{link} s' \Leftrightarrow (\forall o_1, o_2 : Object; \forall a : Association) \\ (areLinked(o_1, o_2, a, s) \Leftrightarrow areLinked(o_1, o_2, a, s'))$$

$\sim_{link+(o_1,o_2,a)}$, such that $s \sim_{link+(o_1,o_2,a)} s'$ holds whenever links in s' are the same as in s but also contain a link between o_1 and o_2 through association a:

$$\begin{aligned} s \sim_{link+(o_1,o_2,a)} s' \Leftrightarrow\ & \neg areLinked(o_1, o_2, a, s) \wedge areLinked(o_1, o_2, a, s') \wedge \\ & (\forall o_3, o_4 : Object \mid o_3 \neq o_1 \vee o_4 \neq o_2) \\ & \quad (areLinked(o_3, o_4, a, s) \Leftrightarrow areLinked(o_3, o_4, a, s')) \wedge \\ & (\forall a' : Association \mid a' \neq a; \forall o_3, o_4 : Object) \\ & \quad (areLinked(o_3, o_4, a', s) \Leftrightarrow areLinked(o_3, o_4, a', s')) \end{aligned}$$

$\sim_{link-(o_1,o_2,a)}$, such that $s \sim_{link-(o_1,o_2,a)} s'$ holds whenever links in s' are the same as in s except for a link between o_1 and o_2 through association a:

$$\begin{aligned} s \sim_{link-(o_1,o_2,a)} s' \Leftrightarrow\ & areLinked(o_1, o_2, a, s) \wedge \neg areLinked(o_1, o_2, a, s') \wedge \\ & (\forall o_3, o_4 : Object \mid o_3 \neq o_1 \vee o_4 \neq o_2) \\ & \quad (areLinked(o_3, o_4, a, s) \Leftrightarrow areLinked(o_3, o_4, a, s')) \wedge \\ & (\forall a' : Association \mid a' \neq a; \forall o_3, o_4 : Object) \\ & \quad (areLinked(o_3, o_4, a', s) \Leftrightarrow areLinked(o_3, o_4, a', s')) \end{aligned}$$

$\sim_{link-o}$, such that $s \sim_{link-o} s'$ holds whenever links in s' are the same as in s except for those in which object o participates:

$$\begin{aligned} s \sim_{link-o} s' \Leftrightarrow\ & (\forall a : Association; \forall o' : Object) \\ & \quad (\neg areLinked(o, o', a, s') \wedge \neg areLinked(o', o, a, s')) \wedge \\ & (\forall a : Association; \forall o_1, o_2 \mid o_1 \neq o \wedge o_2 \neq o) \\ & \quad (areLinked(o_1, o_2, a, s) \Leftrightarrow areLinked(o_1, o_2, a, s')) \end{aligned}$$

Relations on Attribute Values

$\sim_{att}$, such that $s \sim_{att} s'$ holds whenever the values of all the attributes of all objects in s are the same as their corresponding in s':

$$s \sim_{att} s' \Leftrightarrow (\forall o : Object; \forall a : Attribute) \\ (getVal(o, a, s) = getVal(o, a, s'))$$

$\sim_{att-o}$, such that $s \sim_{att-o} s'$ holds whenever the values of all the attributes of all objects of s', except for those of object o, are the same as in s:

$$s \sim_{att-o} s' \Leftrightarrow (\forall o' : Object \mid o' \neq o; \forall a : Attribute) \\ (getVal(o', a, s) = getVal(o', a, s'))$$

$\sim_{att-(o,a)}$, such that $s \sim_{att-(o,a)} s'$ holds whenever the values of all the attributes of all objects of s', except for the value of attribute a of object o, are the same as in s:

$$
\begin{aligned}
s \sim_{att-(o,a)} s' \Leftrightarrow\ & (\forall o' : Object; \forall a' : Attribute \mid a' \neq a) \\
& \quad (getVal(o', a', s) = getVal(o', a', s')) \wedge \\
& (\forall o' : Object \mid o' \neq o; \forall a' : Attribute) \\
& \quad (getVal(o', a', s) = getVal(o', a', s'))
\end{aligned}
$$

3.4.4 Axioms

In this section, functions declared in section 3.4.1 are specified. A resulting state from a command application is associated to the same system as the originating state. Thanks to the relations specified in the previous section, a compact specification of the primitives is achieved. The basic operations in this ADT specification are the query operations: `existsObj`, `getVal` and `areLinked`.

AxStt1 A new state is empty:
$isEmpty(newState(S))$

AxStt2 The empty state has no objects:
$isEmpty(S) \Leftrightarrow (\forall o : Object)(\neg existsObj(o, S))$

AxStt3 The structure of a state is the one specified at state creation and is preserved across command applications:

$$
\begin{aligned}
& structure(s) = S \Leftrightarrow \\
& \quad s = newState(S) \vee \\
& \quad (\exists s' : State \mid structure(s') = S \wedge \\
& \quad\quad (s = create(o, c, s') \vee \\
& \quad\quad s = destroy(o, s') \vee \\
& \quad\quad s = link(o_1, o_2, a, s') \vee \\
& \quad\quad s = unlink(o_1, o_2, a, s') \vee \\
& \quad\quad s = set(o, a, v, s'))
\end{aligned}
$$

AxStt4 An object is an instance of the class from which it was created, and it is also instance of any of its ancestors (subsumption property). Classification is preserved across command applications:

$$
\begin{aligned}
& isInstanceOf(o, c, s) \Leftrightarrow \\
& \quad existsObj(o, s) \wedge \\
& \quad ((\exists c' : Class \mid isAncestor(c, c', structure(s)) \wedge isInstanceOf(o, c', s)) \vee \\
& \quad\ (\exists s' : State \mid s = create(o, c, s')) \vee \\
& \quad\ (\exists s' : State \mid isInstanceOf(o, c, s') \wedge \\
& \quad\quad (s = create(o', c', s') \vee \\
& \quad\quad\ (o \neq o' \wedge s = destroy(o', s')) \vee \\
& \quad\quad\ s = link(o_1, o_2, asc, s') \vee
\end{aligned}
$$

$$s = unlink(o_1, o_2, asc, s')\vee$$
$$s = set(o', att, v, s'))))$$

AxStt5 Semantics for create:

$$s' = create(o, c, s) \Leftrightarrow$$
$$s \sim_{obj+o} s' \wedge s \sim_{link} s' \wedge s \sim_{att-o} s'\wedge$$
$$(\forall c' : Class \mid isAncestor(c', c, structure(s));$$
$$\forall a : Attribute \mid isAttribute(a, c', strucure(s)))$$
$$(getVal(o, a, s') = defVal(a, c', structure(s)))$$

AxStt6 Semantics for destroy:

$$s' = destroy(o, s) \Leftrightarrow s \sim_{obj-o} s' \wedge s \sim_{link-o} s' \wedge s \sim_{att-o} s'$$

AxStt7 Semantics for link:

$$s' = link(o_1, o_2, a, s) \Leftrightarrow s \sim_{obj} s' \wedge s \sim_{link+(o_1,o_2,a)} s' \wedge s \sim_{att} s'$$

AxStt8 Semantics for unlink:

$$s' = unlink(o_1, o_2, a, s) \Leftrightarrow s \sim_{obj} s' \wedge s \sim_{link-(o_1,o_2,a)} s' \wedge s \sim_{att} s'$$

AxStt9 Semantics for set:

$$s' = set(o, a, v, s) \Leftrightarrow$$
$$s \sim_{obj} s' \wedge s \sim_{link} s' \wedge s \sim_{att-(o,a)} s' \wedge getVal(o, a, s') = v$$

AxStt10 A state is well-formed iff every multiplicity is satisfied:

$$isWellFormed(s) \Leftrightarrow$$
$$(\forall a : Association \mid existsAssociation(a, structure(s));$$
$$\forall c_1, c_2 : Class \mid associates(a, c_1, c_2, structure(s));$$
$$\forall m_1, m_2 : Multiplicity \mid multiplicities(a, m_1, m_2, structure(s)))$$
$$((\forall o_1 : Object \mid existsObj(o_1, s) \wedge isInstanceOf(o_1, c_1, s))$$
$$(inRange(|\{o_2 : Object \mid existsObj(o_2, s)\wedge$$
$$isInstanceOf(o_2, c_2, s)\wedge$$
$$areLinked(o_1, o_2, a, s)\}|, m_2))\wedge$$
$$(\forall o_2 : Object \mid existsObj(o_2, s) \wedge isInstanceOf(o_2, c_2, s))$$
$$(inRange(|\{o_1 : Object \mid existsObj(o_1, s)\wedge$$
$$isInstanceOf(o_1, c_1, s)\wedge$$
$$areLinked(o_1, o_2, a, s)\}|, m_1)))$$

3.4.5 Example

In this section we continue our example initiated in the previous chapter. We show invocations to commands of type State by rewriting the examples informally introduced in section 2.2.3. We assume the same variables and operation results as in the example in section 3.3.4, and in addition:

$s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8 : State$
$Company, Product : Class$
$produces : Association$
$c, p, s : Object$
$tt : Value$

The first modification takes a state with no objects in it and produces a state as shown in Figure 2.2 (a). For this we also assume that $s_1 = newState(S)$, which means that s_1 has no objects and is a state associated to S. That modification can be rewritten as:

$s_2 = create(c, Company, s_1);$
$s_3 = create(p, Product, s_2);$
$s_4 = link(c, p, produces, s_3);$
$s_5 = set(p, avail, tt, s_4);$

Then the state shown in Figure 2.2 (a) is represented by s_5. We now consider the second modification. There, state s_5 is modified to produce a resulting state corresponding to that shown in Figure 2.2 (b). That modification can be rewritten as:

$s_6 = create(s, Salesman, s_5);$
$s_7 = link(c, s, works_for, s_6);$
$s_8 = link(s, p, sells, s_7);$

Similarly as before, state s_8 represents the state shown in Figure 2.2 (b). It is interesting to observe that these last invocations can be regarded as the implementation of some system level operation *hireSalesman*. Such operation would take as arguments a company and a product, and create a new salesman. That person is then "hired" by the company and "assigned" to sell the product. With this simple example we demonstrate that it is possible to represent the essence of a possible implementation of an operation like *hireSalesman*. At a conceptual level, what really matters is that an object of class *Salesman* is created and linked to given instances of classes *Company* and *Product*. An implementation in a language like Java would probably involve some object lookup, adding the salesman into the company ArrayList and vice versa, etc. This level is usually too detailed for globally thinking about system behavior, especially in early stages of the development process.

3.5 Discussion

In this section we discuss some alternative specifications of the concepts treated in this chapter, and particularly the impact of their use on the specification of the primitives.
Our specification has a high level of abstraction. Abstract data types enabled

a simple specification of the concepts that constitute the foundation for our semantics. Based on that simplicity, a compact definition of the semantics of the primitives was produced. By abstracting away actual representations, we were able to focus on the effect of each primitive without getting distracted by the complexities of data structures. For example, it is not necessary to know how a state is represented to understand the meaning of creating an object in it. It is possible to express the effect of such modification in terms of observable properties of both source and final states. However, a semantics of the primitives as the one presented in this chapter, and therefore the framework to be introduced next, can be based on other specifications which are compatible with the concepts specified here. The difference between such semantics is a matter of complexity, which is a subjective aspect. However, our specification has shown to have the power to support the primitives, and we consider that is simple enough to allow a smooth translation into the Calculus of Inductive Constructions.

Other works, such as [Ric02] and [OMG03c], include a specification of the necessary types for specifying the primitives. The context of those works was different, as is the level of abstraction of their specifications. The motivation of those specifications is tool-oriented, so representations naturally show some degree of detail which, in our case, is completely unnecessary, as discussed above. The first specification is based on set theory, and is the foundation of the development of the USE Tool [Ric01]. It proposes structures representing systems and states, and defines an interpretation of systems as the set of all possible states. A specification of the Object Constraint Language is also included. The main goal of the implemented tool is to provide an environment where a OCL constraints (i.e. invariants, and pre- and postconditions) can be evaluated for validation purposes. In such environment, a UML model specifying a system can be instantiated, states can be interactively changed via the same set of primitives we specified, and OCL constraints are evaluated against the current state. The second specification is the UML metamodel. It is an object-oriented representation of object-oriented system and states, among others. Its primary audience includes advanced modelers, metamodelers, and tool builders [OMG03b, p. 2-2]. Possible target tools are modeling tools and code generators. Despite being intended as a logical model instead of a physical one, it is very concrete. In fact, major parts of the implementation of a tool which stores systems and state descriptions can be easily produced with help of a code generation tool. Microsoft Visio [MC03] is a good example of a modeling tool whose implementation closely follows that specification.

Some of the flavour of a semantics of the primitives defined using the alternative specifications is shown in Appendix A, where we experiment with the create primitive.

Summary

In this chapter, an ADT-based specification of the notions of system and state was presented. This specification is the foundation for the definition of the semantics of system state modification primitives. The specification here introduced possesses a high level of abstraction. This feature results in a simple specification of the types necessary to define the semantics of the primitives. In turn, we were able to propose that semantics in a compact and understandable way, which is defined making use of well understood and widely used mathematical constructs. Before we move into the formalization of the specification in the Calculus of the Inductive Constructions, we first investigate a natural application of the defined semantics. In the next chapter, we address the definition of a framework which aids in formally reasoning about the use of the primitives.

Chapter 4

Reasoning about the Use of Primitives

System state modification primitives can be regarded, in the sense of imperative programming, as basic commands for changing system states. In the previous chapter it was shown that these commands can be used for representing higher level programs with effects on states. This naturally raises the idea of writing programs at a conceptual level as sequences of basic state modifications or system state modification primitives, and also considering the problem of how to prove that such programs do what we require from them. The specification presented in the previous chapter provides a solid foundation to address this problem.

In this chapter, we explore the systematic verification of programs written as sequences of modification primitives. For that, we define a framework where correctness assertions about programs can be formulated and proved. Type State is the basis of such a framework. Correctness assertions are formulated using operations associated to that type. We use commands for writing programs, and queries for expressing assertions about properties of states. For proving formulae we use the specification of the primitives as an axiomatic definition of commands, and we also define specific rules.

The framework is actually an inference system where we will try to build proofs for correctness assertions. As discussed in Chapter 2, those assertions express some interesting properties about the resulting state of a program (postconditions), provided that its execution started under some specified conditions (preconditions). Choosing the right pre- and postconditions and then proving the correctness assertion means ensuring that the program does what we expect from it. The use of preconditions and postconditions as specifications for programs, constitutes the most popular technique for specifying and documenting operations in object-orientation.

This chapter is structured as follows. In section 4.1 we describe our approach to program verification. In section 4.2 the definition of the framework for reasoning about the use of primitives is presented. An extension to our framework where new assertions about state properties are provided is proposed in section 4.3. Section 4.4 explores a further extension to the framework for reasoning about state well-formedness.

4.1 Our Approach to Program Verification

In this section we explain our approach to program verification. First, we discuss in more detail the motivation for defining a framework for reasoning about the use of primitives. Then we overview inference systems and briefly describe the key elements we will define for our framework.

4.1.1 Motivation

System state modification primitives can be used as commands of a basic command language for building programs which operate on system states. The idea of such a command language is actually not new. In fact, the USE Tool [Ric01] provides an environment where such programs can be executed on particular states, in what is called *animation*. There, conforming to a given structure, a system changes from one state to another. Commands are issued sequentially and each one triggers a transition between states. Based on the idea of correctness assertions, as discussed in section 2.3.1, the main purpose of that tool is to provide an environment for validating UML models and OCL constraints. Concretely, it accepts a system structure description, operation specifications and invariants expressed in OCL, and programs implementing those operations written in the command language. At any point, an invocation to an operation can be started. The environment evaluates its preconditions against the current state, and if satisfied lets the user execute the program. When the program is completed the postconditions are in turn evaluated again against the current state, which is the resulting state of the program. Trusting the program correctness, it is possible to check whether the pre- and postconditions are right or not. That is what the author mean by validation. Of course, if the specification is trusted instead, then validation applies to the program. By this process, specifications or programs are validated with respect to particular state configurations. In practice, a number of test cases should be validated. However, complete reliability is achieved only by being exhaustive, which in general is not possible.
We are interested in a stronger use of correctness assertions, particularly, we are interested in proving them. A proof of a correctness assertion can be informally understood as a proof of the fact that a program starting in *any* state that satisfies the precondition, if it terminates, it will do so in a resulting state that will satisfy the postconditions. Such proofs are based on properties of the states and the program, rather than on the values in the state. In other words, proofs

are inferences and not computations of particular test cases. It is for this reason that such a high level of abstraction could apply.
The use of inference systems is a possible approach for systematic program verification. As we define our framework as an inference system, in what follows we overview their main concepts first.

4.1.2 Inference Systems

Formulae of an inference system are *properties*, and derivation trees that constitute proofs for formulae are called *inference trees* because they show how to infer that certain property holds.
To recall, an inference system has two major parts: a language and a set of rules. The language is used for expressing the formulae or properties, while the set of rules restricts the form which proofs can take. In inference systems, this means that rules limit the admissible inference steps in a proof. In our case, we will handle a proof system were formulae are just correctness assertions. As usual, some rules are actually axioms. These axioms ultimately describe the meaning of the constructs in the language used to write the programs in the correctness assertions [Win93]. That is the way in which the framework uses or includes the specification of the system state modification primitives of the previous chapter. State queries in turn, are used for expressing pre- and postconditions in correctness assertions, since they are the means we defined for accessing state properties.
In the next section we fully define the inference system which will be our framework for reasoning about the use of primitives.

4.2 The Framework

In this section the framework for reasoning about the use of primitives is defined as an inference system. Formulae are partial correctness assertions. For formulating those assertions we introduce the notion of program, and describe the structure of pre- and postconditions. For building proofs, axioms are introduced and a set of inference rules are defined.

4.2.1 Formulae

In this inference system, *formulae* are partial correctness assertions of the form

$$\{ P \} \; S \; \{ Q \}$$

where S is a program, and P and Q are assertions on the state. Assertion P is called the *precondition* and Q is called the *postcondition*. The operational interpretation of such formula is:

if P holds in the initial state, and
if the execution of S terminates when started in that state,
then Q will hold in the state in which S halts.

Note that for $\{\ P\ \}\ S\ \{\ Q\ \}$ to hold it is *not* required that S terminates when started in states satisfying P. What it is required is that Q must hold *if* it terminates. We will come back to termination issues later in this section.

Programs

We define a *program* as a finite number of composite applications of commands to a given state, such as:

$$s_n = cmd_n(cmd_{n-1}(\ldots cmd_1(s)))$$

for a state s and a finite and positive n. Then a program can be understood as a sequence of applications of primitives where the first (innermost) is applied to an *initial state* and the $i+1$th primitive is applied to the result of the ith application; the result of the last (outmost) application is the *resulting state* of the program. For clarity, we write program in an imperative-like style, using operator ';' for denoting functional composition:

$$\begin{aligned} s_2 &= cmd_1(s); \\ &\vdots \\ s_n &= cmd_n(s_{n-1}); \end{aligned}$$

We also call each application a *step* of the program, and at one step the program performs an *action* on the current state (i.e. modifies the state according to the corresponding primitive of the step). However, it is important to note that both programs and steps are functions. For example, a step of a program like: $s' = create(o, c, s)$ does not actually modify state s. Rather, it yields a new state s'. When compared to the initial state, the final state shows some differences: object o, which was not present in state s, is in state s', as defined in the previous chapter.

Assertions

An assertion language allows us to express pre- and postconditions. In our system, assertions are based on query operations associated to types System and State. Queries returning boolean values can be combined using any of the usual connectives: $\wedge, \vee, \neg$ and $\Rightarrow$. For operations returning other types of values, operator $=$ may be used as well. Quantifiers $\exists$ and $\forall$ are allowed too.
Queries associated to type State are intended to express properties about initial and final states, therefore are naturally used in pre- and postconditions. Queries associated to type System are intended to express properties about the system associated to the initial state. For this reason, they are likely to be used in preconditions. In addition, *structure* would occur in preconditions too, since all

states involved in a program, particularly source and final states, share the same structure. The use within the framework of operations concerning multiplicities (e.g. $isWellFormed$ and *multiplicities*) will be explored in section 4.4.

Termination

As already discussed, in a partial correctness assertion, postconditions are guaranteed only when the execution starts in a state in which preconditions hold, and it terminates. On the other hand, total correctness deals with program termination, requiring an explicit proof of it for every program. In our framework programs always terminate, since they were defined as finite sequences of transformations, where no iteration or recursion is supported. We say our framework uses partial correctness only because we do not provide proofs of termination. However, as termination is guaranteed by construction, the particular form of partial correctness we use actually acts as total correctness.

4.2.2 Axioms

Axioms in this kind of systems describe the meaning of the program constructs. In our case, we already have that ingredient for our framework, and it is the specification of system state modification primitives introduced in the previous chapter. Particularly, we use axioms AxStt5, AxStt6, AxStt7, AxStt8, and AxStt9. For completeness we recall:

AxStt5 $s' = create(o, c, s) \Leftrightarrow$
$$s \sim_{obj+o} s' \wedge s \sim_{link} s' \wedge s \sim_{att-o} s' \wedge$$
$$(\forall c' : Class \mid isAncestor(c', c, structure(s);$$
$$\forall a : Attribute \mid isAttribute(a, c', strucure(s)))$$
$$(getVal(o, a, s') = defVal(a, c', structure(s)))$$

AxStt6 $s' = destroy(o, s) \Leftrightarrow s \sim_{obj-o} s' \wedge s \sim_{link-o} s' \wedge s \sim_{att-o} s'$

AxStt7 $s' = link(o_1, o_2, a, s) \Leftrightarrow s \sim_{obj} s' \wedge s \sim_{link+(o_1,o_2,a)} s' \wedge s \sim_{att} s'$

AxStt8 $s' = unlink(o_1, o_2, a, s) \Leftrightarrow s \sim_{obj} s' \wedge s \sim_{link-(o_1,o_2,a)} s' \wedge s \sim_{att} s'$

AxStt9 $s' = set(o, a, v, s) \Leftrightarrow$
$$s \sim_{obj} s' \wedge s \sim_{link} s' \wedge s \sim_{att-(o,a)} s' \wedge getVal(o, a, s') = v$$

There exist a tight connection between these axioms and the assertion language, which will become more evident in the next section. The semantics of the primitives heavily relies on the relations between states introduced in section 3.4.3, which in turn were defined completely in terms of query operations associated to type State. All the rules were derived from the axioms, therefore there is nothing really new in them. This means that proofs could be built just using

these axioms. Our rules ultimately are a more convenient means for performing inference steps.

4.2.3 Inference Rules

We now introduce the set of inference rules for proving correctness assertions. The following rules are derived from the axioms defined for the type State, and we present them organized according to the kind of property they allow us to infer. Each rule specifies the form of individual steps which make up a proof. They involve only consecutive states. In the premises only one state appears, the one taken as argument by a command, and the consequent refers to the result of the command.
For the rules we assume the following symbols:

$s, s' : State$
$o, o', o_1, o_2, o_3, o_4 : Object$
$c, c', c_1, c_2 : Class$
$v, v' : Value$
$att : Attribute$
$asc : Association$

Symbols a and a' refer to instances of both *Attribute* and *Association*, although in disjoint contexts.

Rules for Object Existence

The following set of rules allows the inference of the existence of an object. The object could have been created in the previous state, or the object already existed and was preserved in the last step. We similarly include rules for inferring the non-existence of objects.
Using the first rule we infer that an object exists because it was created in the transition to the final state.

$$\frac{\begin{array}{c}\neg existsObj(o, s) \\ \neg isAbstract(c, structure(s)) \quad s' = create(o, c, s)\end{array}}{existsObj(o, s')}\ \mathsf{Ex1}$$

The next group of rules allows us to infer that an object exists because it already existed before the transition, and it was not destroyed.

$$\frac{existsObj(o, s) \quad o \neq o' \quad s' = create(o', c, s)}{existsObj(o, s')}\ \mathsf{Ex2Cr}$$

$$\frac{existsObj(o, s) \quad o \neq o' \quad s' = destroy(o', s)}{existsObj(o, s')}\ \mathsf{Ex2Dt}$$

$$\frac{existsObj(o,s) \quad s' = link(o_1,o_2,a,s)}{existsObj(o,s')}\ \mathsf{Ex2Lk}$$

$$\frac{existsObj(o,s) \quad s' = unlink(o_1,o_2,a,s)}{existsObj(o,s')}\ \mathsf{Ex2Ul}$$

$$\frac{existsObj(o,s) \quad s' = set(o',a,v,s)}{existsObj(o,s')}\ \mathsf{Ex2St}$$

Rule Ex3 expresses a sufficient condition for object existence. It is introduced by precondition PreStt3.

$$\frac{isInstanceOf(o,c,s)}{existsObj(o,s)}\ \mathsf{Ex3}$$

The following rule allows us to infer that an object does not exist in a state when it is destroyed in the transition to that state.

$$\frac{existsObj(o,s) \quad s' = destroy(o,s)}{\neg existsObj(o,s')}\ \mathsf{NEx1}$$

In this group of rules the inference is based on the case where the object did not exist and was not created in the transition.

$$\frac{\neg existsObj(o,s) \quad o \neq o' \quad s' = create(o',c,s)}{\neg existsObj(o,s')}\ \mathsf{NEx2Cr}$$

$$\frac{\neg existsObj(o,s) \quad s' = destroy(o',s)}{\neg existsObj(o,s')}\ \mathsf{NEx2Dt}$$

$$\frac{\neg existsObj(o,s) \quad s' = link(o_1,o_2,a,s)}{\neg existsObj(o,s')}\ \mathsf{NEx2Lk}$$

$$\frac{\neg existsObj(o,s) \quad s' = unlink(o_1,o_2,a,s)}{\neg existsObj(o,s')}\ \mathsf{NEx2Ul}$$

$$\frac{\neg existsObj(o,s) \quad s' = set(o,a,v,s)}{\neg existsObj(o,s')}\ \mathsf{NEx2St}$$

Rules for Attribute Values

The following set of rules allows the inference of attributes holding particular values. The value could have been set in the previous state, or it was already held by the attribute and was not changed in the last step. As before, we also include rules for inferring that an attribute does not hold a particular value.
The first rule is used to infer the value of an attribute based on the update performed on it in the transition.

$$\frac{s' = set(o, a, v, s)}{getVal(o, a, s') = v}\ \mathsf{GV1}$$

The next rule is used to infer the value of an attribute based on the fact that the target object was created in the transition.

$$\frac{isAncestor(c', c, structure(s)) \quad isAttribute(a, c', structure(s)) \quad defVal(a, c', structure(s)) = v \quad s' = create(o, c, s)}{getVal(o, a, s') = v}\ \mathsf{GV2}$$

Rules in the following group are used to infer the value of an attribute based on the fact that the last known value is preserved in the transition.

$$\frac{getVal(o, a, s) = v \quad s' = create(o', c, s)}{getVal(o, a, s') = v}\ \mathsf{GV3Cr}$$

$$\frac{getVal(o, a, s) = v \quad o \neq o' \quad s' = destroy(o', s)}{getVal(o, a, s') = v}\ \mathsf{GV3Dt}$$

$$\frac{getVal(o, a, s) = v \quad s' = link(o_1, o_2, asc, s)}{getVal(o, a, s') = v}\ \mathsf{GV3Lk}$$

$$\frac{getVal(o, a, s) = v \quad s' = unlink(o_1, o_2, asc, s)}{getVal(o, a, s') = v}\ \mathsf{GV3Ul}$$

$$\frac{getVal(o, a, s) = v \quad a \neq a' \quad s' = set(o, a', v', s)}{getVal(o, a, s') = v}\ \mathsf{GV3St1}$$

$$\frac{getVal(o, a, s) = v \quad o \neq o' \quad s' = set(o', a', v', s)}{getVal(o, a, s') = v}\ \mathsf{GV3St2}$$

With this rule we can infer that the value of an attribute is not a given one when some different value was set to it in the transition.

$$\frac{v \neq v' \quad s' = set(o, a, v', s)}{getVal(o, a, s') \neq v}\ \mathsf{NGV1}$$

Using rules of the following group it is possible to infer that when an attribute of a particular object does not hold some value and in the transition that value was not set to the attribute, then the attribute still does not hold that value.

$$\frac{getVal(o, a, s) \neq v \quad s' = create(o', c, s)}{getVal(o, a, s') \neq v}\ \mathsf{NGV2Cr}$$

$$\frac{getVal(o, a, s) \neq v \quad o \neq o' \quad s' = destroy(o', s)}{getVal(o, a, s') \neq v}\ \mathsf{NGV2Dt}$$

$$\frac{getVal(o,a,s) \neq v \quad s' = link(o_1,o_2,asc,s)}{getVal(o,a,s') \neq v} \text{ NGV2Lk}$$

$$\frac{getVal(o,a,s) \neq v \quad s' = unlink(o_1,o_2,asc,s)}{getVal(o,a,s') \neq v} \text{ NGV2Ul}$$

$$\frac{getVal(o,a,s) \neq v \quad (o \neq o' \vee a \neq a') \quad s' = set(o',a',v',s)}{getVal(o,a,s') \neq v} \text{ NGV2St}$$

Rules for Object Linkage

The following set of rules allows the inference of the existence of a link between a pair of objects. The link could have been created in the previous state, or it could already exist and is preserved in the last step. Rules for inferring the non-existence of a link are also included.
With this first rule we infer that two objects are linked in some state when the link was created in the transition to that state.

$$\frac{s' = link(o_1,o_2,a,s)}{areLinked(o_1,o_2,a,s')} \text{ AL1}$$

With the rules of the following group, inference is based on the fact that the link existed before the transition and is preserved in it.

$$\frac{areLinked(o_1,o_2,a,s) \quad s' = create(o,c,s)}{areLinked(o_1,o_2,a,s')} \text{ AL2Cr}$$

$$\frac{areLinked(o_1,o_2,a,s) \quad o \neq o_1 \quad o \neq o_2 \quad s' = destroy(o,s)}{areLinked(o_1,o_2,a,s')} \text{ AL2Dt}$$

$$\frac{areLinked(o_1,o_2,a,s) \quad a \neq a' \quad s' = link(o_3,o_4,a',s)}{areLinked(o_1,o_2,a,s')} \text{ AL2Lk1}$$

$$\frac{areLinked(o_1,o_2,a,s) \quad (o_1 \neq o_3 \vee o_2 \neq o_4) \quad s' = link(o_3,o_4,a,s)}{areLinked(o_1,o_2,a,s')} \text{ AL2Lk2}$$

$$\frac{areLinked(o_1,o_2,a,s) \quad a \neq a' \quad s' = unlink(o_3,o_4,a',s)}{areLinked(o_1,o_2,a,s')} \text{ AL2Ul1}$$

$$\frac{areLinked(o_1,o_2,a,s) \quad (o_1 \neq o_3 \vee o_2 \neq o_4) \quad s' = unlink(o_3,o_4,a,s)}{areLinked(o_1,o_2,a,s')} \text{ AL2Ul2}$$

$$\frac{areLinked(o_1,o_2,a,s) \quad s' = set(o,att,v,s)}{areLinked(o_1,o_2,a,s')} \text{ AL2St}$$

With this rule we infer that a link does not exist in a state because it was removed in the transition to it.

$$\frac{areLinked(o_1, o_2, a, s) \quad s' = unlink(o_1, o_2, a, s)}{\neg areLinked(o_1, o_2, a, s')} \text{NAL1}$$

The destruction of an object let us infer that any possible link which involved that object no longer exist.

$$\frac{s' = destroy(o_1, s)}{\neg areLinked(o_1, o_2, a, s') \land \neg areLinked(o_2, o_1, a, s')} \text{NAL2}$$

The following rules are used to infer that a link does not exist based on the fact that before the transition it did not exist and in the transition it was not created.

$$\frac{\neg areLinked(o_1, o_2, a, s) \quad s' = create(o, c, s)}{\neg areLinked(o_1, o_2, a, s')} \text{NAL3Cr}$$

$$\frac{\neg areLinked(o_1, o_2, a, s) \quad o_1 \neq o \quad o_2 \neq o \quad s' = destroy(o, s)}{\neg areLinked(o_1, o_2, a, s')} \text{NAL3Dt}$$

$$\frac{\neg areLinked(o_1, o_2, a, s) \quad (o_1 \neq o_3 \lor o_2 \neq o_4) \quad s' = link(o_3, o_4, a, s)}{\neg areLinked(o_1, o_2, a, s')} \text{NAL3Lk1}$$

$$\frac{\neg areLinked(o_1, o_2, a, s) \quad a \neq a' \quad s' = link(o_1, o_2, a', s)}{\neg areLinked(o_1, o_2, a, s')} \text{NAL3Lk2}$$

$$\frac{\neg areLinked(o_1, o_2, a, s) \quad (o_1 \neq o_3 \lor o_2 \neq o_4) \quad s' = unlink(o_3, o_4, a, s)}{\neg areLinked(o_1, o_2, a, s')} \text{NAL3Ul1}$$

$$\frac{\neg areLinked(o_1, o_2, a, s) \quad a \neq a' \quad s' = unlink(o_1, o_2, a', s)}{\neg areLinked(o_1, o_2, a, s')} \text{NAL3Ul2}$$

$$\frac{\neg areLinked(o_1, o_2, a, s) \quad s' = set(o, att, v, s)}{\neg areLinked(o_1, o_2, a, s')} \text{NAL3St}$$

Rules for Object Classification

The following set of rules allows the inference of the classification of an object. Rules for inferring that an object is not instance of a class are also included. The first rules are directly derived from the specification of create. We can infer the class of an object knowing the class used for its creation.

$$\frac{s' = create(o, c, s)}{isInstanceOf(o, c, s')} \text{IO1}$$

The next rule is the subsumption property [AC96], also known as substitutability principle [RJB98].

$$\frac{\mathit{isInstanceOf}(o, c_1, s) \quad \mathit{isAncestor}(c_2, c_1, \mathit{structure}(s))}{\mathit{isInstanceOf}(o, c_2, s)} \ \mathsf{Subs}$$

By the next rules we infer, while still existing, an object does not change its classification.

$$\frac{\mathit{isInstanceOf}(o, c, s) \quad s' = \mathit{create}(o', c', s)}{\mathit{isInstanceOf}(o, c, s')} \ \mathsf{IO5Cr}$$

$$\frac{\mathit{isInstanceOf}(o, c, s) \quad o \neq o' \quad s' = \mathit{destroy}(o', s)}{\mathit{isInstanceOf}(o, c, s')} \ \mathsf{IO5Dt}$$

$$\frac{\mathit{isInstanceOf}(o, c, s) \quad s' = \mathit{link}(o_1, o_2, a, s)}{\mathit{isInstanceOf}(o, c, s')} \ \mathsf{IO5Lk}$$

$$\frac{\mathit{isInstanceOf}(o, c, s) \quad s' = \mathit{unlink}(o_1, o_2, a, s)}{\mathit{isInstanceOf}(o, c, s')} \ \mathsf{IO5Ul}$$

$$\frac{\mathit{isInstanceOf}(o, c, s) \quad s' = \mathit{set}(o', a, v, s)}{\mathit{isInstanceOf}(o, c, s')} \ \mathsf{IO5St}$$

4.2.4 Using the Framework

Our framework is complete and ready for start working. In this section we illustrate its use through an example and discuss the necessary steps to prove correctness assertions. We use once again the case introduced in Chapter 2. A correctness assertion for operation *hireSalesman* is formulated and proved. The program used for this operation is that presented in section 3.4.5.

Formulating Correctness Assertions

For formulating a correctness assertion we need a specification and a program. The specification expresses our intentions with respect to the program behavior, and the program specifies the commands that realize these intentions. The precondition expresses the conditions that needs to be guaranteed on the initial state for the program to execute. The postcondition in turn, expresses what properties, in our judgement, should hold when the program is complete. A proof of a correctness assertion means that the program meets the specification. Usually, we first write the specification and then the program. A framework like this can be used to make sure the second task was performed correctly. However, correctness is a relative notion because we rely on the specification. The specification of a program is crucial, simply because a wrong specification can

make a right program look as if it is wrong, and more dangerously, can make a wrong program look as if it is right. For the *hireSalesman* operation, we write the following correctness assertion :

$$
\begin{array}{l}
\{\ \neg existsObj(s, s_1) \wedge \neg isAbstract(Salesman, structure(s_1)) \wedge \\
\quad works_for \neq sells \wedge areLinked(c, p, produces, s_1)\ \} \\
\qquad s_2 = create(s, Salesman, s_1); \\
\qquad s_3 = link(c, s, works_for, s_2); \\
\qquad s_4 = link(s, p, sells, s_3); \\
\{existsObj(s, s_4) \wedge areLinked(c, s, works_for, s_4) \wedge areLinked(s, p, sells, s_4)\}
\end{array}
$$

The program in this formula is is the same as that in section 3.4.5, although using the composition operator. The postcondition is a conjunction which expresses what we expect from the program: the salesman should have been created, and it should have been linked to both the company and product. Note all those properties refer only to state s_4, the final state. The precondition is also a conjunction of information we need to know in order to properly run the program. Note again that properties in the precondition refer only to state s_1, the initial state. Now, we have formulated our correctness assertion for *hireSalesman*, but what is exactly what we have to prove? For any assertion $\{\ P\ \}\ S\ \{\ Q\ \}$ we need to prove the following proposition:

$$
\frac{P(s) \quad s' = S(s)}{Q(s')}
$$

This can be understood as assuming that the precondition holds on the initial state, and assuming the program yields the final state, then prove that the post-condition holds for that final state. This is consistent with the interpretation of the correctness assertion given at the beginning of this section. Note that the equality used here is computable, meaning that S always delivers a result, or in other words, it always terminates. In our particular case, we should prove the following:

$$
\frac{
\begin{array}{l}
\neg existsObj(s, s_1) \\
\neg isAbstract(Salesman, structure(s_1)) \\
works_for \neq sells \\
areLinked(c, p, produces, s_1) \\
s_2 = create(s, Salesman, s_1) \\
s_3 = link(c, s, works_for, s_2) \\
s_4 = link(s, p, sells, s_3)
\end{array}
}{existsObj(s, s_4) \wedge areLinked(c, s, works_for, s_4) \wedge areLinked(s, p, sells, s_4)}
$$

Next, we discuss how inference is drawn in our framework, and we build and explain the proof for our example.

Inference and Proof Building

In our framework we perform top-down inference, which is the manner in which solutions are most often discovered [Kow79]. Top-down inference is the analysis of goals into subgoals. It consists in proceeding backwards from the conclusion by using implications to reduce problems to subproblems. There, the aim is to reduce the original problem to a set of subproblems each of which has been solved. In other words, we match the conclusion against the conclusion of some rule, doing which we generate subgoals (proof obligations) for the premises of the rule, if any. The subgoals are then proved in the same fashion. The process ends when all the premises are found as hypothesis in the context of the proof. In our system, we are likely to find many rules whose conclusion matches the conclusion of the main proposition. Assuming this conclusion refers to state s_i, we look for the command that produces that state, and simply apply the rule which has the same command in its premises.
In our example, the conclusion is a conjunction, so the first thing we do is to split it and prove each conjunct separately. Let us start with $existsObj(s, s_4)$. The command yielding state s_4 is link, therefore we apply rule Ex2Lk since its conclusion matches our current goal and one of its premises is proved by hypothesis. However, a new subgoal is generated corresponding to the other premise. Now we have to infer $existsObj(s, s_3)$. Again we apply rule Ex2Lk, and again a new subgoal is generated, now $existsObj(s, s_2)$. For this subgoal we apply rule Ex1, concluding this part of the proof since all its premises are part of the hypotheses.
We summarize this procedure by annotating the correctness assertion similarly as in [Kal90], instead of using an inference tree. Since the proof of individual steps are simply rule applications, we justify each inference step by including in the annotation the name of the applied rule.

$$\begin{array}{l}
\{\ \neg existsObj(s, s_1) \wedge \neg isAbstract(Salesman, structure(s_1)) \wedge \\
\quad works_for \neq sells \wedge areLinked(c, p, produces, s_1)\ \} \\
\qquad s_2 = create(s, Salesman, s_1); \\
\qquad \{\ existsObj(s, s_2), \mathsf{Ex1}\ \} \\
\qquad s_3 = link(c, s, works_for, s_2); \\
\qquad \{\ existsObj(s, s_3), \mathsf{Ex2Lk}\ \} \\
\qquad s_4 = link(s, p, sells, s_3); \\
\{\ existsObj(s, s_4), \mathsf{Ex2Lk}\ \}
\end{array}$$

We proceed similarly to prove the rest of the conjuncts in the original goal. The details of the complete proof can be expressed in one annotated correctness assertion, or if it is more convenient, separately.
Finally, it can be seen that it is feasible to build proofs of this kind (at least semi-) automatically. A report of experiments on automatic proof generation can be found in Chapter 5. We experimented on the formalized version of the framework in the System Coq.

4.2.5 Analysis

Previously, we discussed the important role played by specifications in correctness assertions, particularly the consequences of committing errors when writing specifications. There is another issue concerning specifications we need to discuss. Postconditions impose conditions which must be fulfilled by the final state. However, situations where postconditions completely characterize the final state are very atypical, simply because for large states that would be tedious or even impracticable. For simplicity, postconditions are generally weakened under the assumption that unreferenced variables remain unchanged. In other situations, postconditions are inadvertently weakened just by omission. In any case, partially specified final states are an open door for unexpected things to happen.
Let us consider again the correctness assertion just discussed. Now consider a program that *also* changes the value of `avail` in object `p`. It should be noticed that this program can also be proved to be correct with respect to the *same* specification. Therefore, someone who only reads a program specification, typically a user, in this kind of situations could be unaware of all the things that the program actually does. Whenever these kind of scenarios should be prevented a possible solution is being exhaustive when writing postconditions. A more practical approach is addressed in the next section, where an extension to our framework is introduced.

4.3 An Extension

In this section we extend our framework for providing a practical means by which the complete effect of programs can be specified. New relations for specifying postconditions are defined. They relate the initial state of the program to its final state. A new set of rules for deriving properties about the final state from these relations is also defined.

4.3.1 Assertions and Inference Rules

We are looking for a practical way of expressing the exact differences that the final state has with respect to the initial state in a postcondition. Specifically, our extension should enable us to focus only on differences, without being forced to explicitly handle all the possible cases. For that to be achieved, the unchanged-by-default basis must be defined and not simply assumed.
We define a relation between two states and a set of differences between them, from which we then infer properties on the final state like those in the previous section. The semantics of that relation must be such that the two states are equal except for exactly those changes. More precisely, we define a relation for every sort of state changes already identified: changes on objects, links and attribute values. In fact, such relations can be understood as a generalization of relations introduced in section 3.4.3. For example, relations $\sim_{obj+o}$ and $\sim_{obj-o}$, which were meant to handle differences between states consisting of a single object and in a single direction, are generalized into $\sim obj\ C\ D$. This particular

relation handles sets C and D of created and destroyed objects. In what follows we define such relations.

Relation on Objects

Elements in the relation *objectsChanged* represent the observable differences in terms of objects between two states. The domain of this relation is defined as:

$$\texttt{objectsChanged} \subseteq \texttt{Set(Object)}\times\texttt{Set(Object)}\times\texttt{State}\times\texttt{State}$$

We define the relation such that $objectsChanged(C, D, s, s')$ holds whenever the objects in s' are the same as those in s, except for those included in D, and including those in C. We denote $objectsChanged(C, D, s, s')$ as $s \sim_{obj\ C\ D} s'$, and define it inductively by the following constructors:

$$\frac{s \sim_{obj} s'}{s \sim_{obj\ \emptyset\ \emptyset} s'}\ \textsf{nilnilO}$$

$$\frac{s \sim_{obj\ C\ D} s' \quad s' \sim_{obj} s''}{s \sim_{obj\ C\ D} s''}\ \textsf{indEqO}$$

$$\frac{s \sim_{obj\ C\ D} s' \quad s' \sim_{obj+o} s'' \quad \neg existsObj(o, s) \quad o \notin C \quad o \notin D}{s \sim_{obj\ C\cup\{o\}\ D} s''}\ \textsf{indPlusO1}$$

$$\frac{s \sim_{obj\ C\ D} s' \quad s' \sim_{obj+o} s'' \quad existsObj(o, s) \quad o \notin C \quad o \in D}{s \sim_{obj\ C\ D-\{o\}} s''}\ \textsf{indPlusO2}$$

$$\frac{s \sim_{obj\ C\ D} s' \quad s' \sim_{obj-o} s'' \quad existsObj(o, s) \quad o \notin C \quad o \notin D}{s \sim_{obj\ C\ D\cup\{o\}} s''}\ \textsf{indMinusO1}$$

$$\frac{s \sim_{obj\ C\ D} s' \quad s' \sim_{obj-o} s'' \quad \neg existsObj(o, s) \quad o \in C \quad o \notin D}{s \sim_{obj\ C-\{o\}\ D} s''}\ \textsf{indMinusO2}$$

The first two constructors, nilnilO and indEqO, handle the cases where there are no changes occurring in the step. The next two constructors handle cases where an object is created. Since an object may not be in the two sets simultaneously, if the object is in set D then it is removed by constructor indPlusO2, otherwise constructor indPlusO1 adds it to set C. Note that although these collections are sets, before adding an element to a collection we explicitly require that the element may not be in that collection already. This is to prevent erroneous cases where a program attempts to issue a create command on an object already created and not destroyed. Similarly, constructors indMinusO1 and indMinusO2 handle cases where an object is destroyed. Finally, note that there are no rules for base steps involving $s \sim_{obj+o} s'$ and $s \sim_{obj-o} s'$, but only for $s \sim_{obj} s'$. Since

$s \sim_{obj} s$ holds for every state s, any of the two cases above can be resolved using $s \sim_{obj} s$ in addition to $s \sim_{obj+o} s'$, or $s \sim_{obj-o} s'$ respectively.

Relation on Links

Elements in the relation *linksChanged* represent the observable differences in terms of links between two states. The domain of this relation is defined as:

$$\texttt{linksChanged} \subseteq \texttt{Set(Link)}\times\texttt{Set(Link)}\times\texttt{State}\times\texttt{State}$$

In turn, type Link is defined inductively by the following constructors:

$$\frac{o_1 : Object \quad o_2 : Object \quad a : Association}{(o_1, o_2, a) : Link}\ \textsf{oneL} \qquad \frac{o : Object}{(o, *) : Link}\ \textsf{allL}$$

We define the relation such that $linksChanged(C, D, s, s')$ holds whenever the links in s' are the same as those in s, except for those included in D, and including those in C. An instance of type Link of the form (o_1, o_2, a) represents a link between objects o_1 and o_2 through association a. An instance of type Link of the form $(o, *)$ represents all links involving object o. We use a special infix operator:

$$\ominus \ : \ \texttt{Set(Link)}\times\texttt{Object}\rightarrow\texttt{Set(Link)}$$

where $(st \ominus o)$ is equal to set st where all elements involving o in any position are removed. We denote $linksChanged(C, D, s, s')$ as $s \sim_{link\ C\ D} s'$, and define it inductively by the following constructors:

$$\frac{s \sim_{link} s'}{s \sim_{link\ \emptyset\ \emptyset} s'}\ \textsf{nilnilL}$$

$$\frac{s \sim_{link\ C\ D} s' \quad s' \sim_{link} s''}{s \sim_{link\ C\ D} s''}\ \textsf{indEqL}$$

$$\frac{s \sim_{link\ C\ D} s' \quad s' \sim_{link+(o_1,o_2,a)} s'' \quad \neg areLinked(o_1, o_2, a, s) \quad (o_1, o_2, a) \notin C \quad (o_1, o_2, a) \notin D}{s \sim_{link\ C\cup\{(o_1,o_2,a)\}\ D} s''}\ \textsf{indPlusL1}$$

$$\frac{s \sim_{link\ C\ D} s' \quad s' \sim_{link+(o_1,o_2,a)} s'' \quad areLinked(o_1, o_2, a, s) \quad (o_1, o_2, a) \notin C \quad (o_1, o_2, a) \in D}{s \sim_{link\ C\ D-\{(o_1,o_2,a)\}} s''}\ \textsf{indPlusL2}$$

$$\frac{s \sim_{link\ C\ D} s' \quad s' \sim_{link-(o_1,o_2,a)} s'' \quad areLinked(o_1, o_2, a, s) \quad (o_1, o_2, a) \notin C \quad (o_1, o_2, a) \notin D}{s \sim_{link\ C\ D\cup\{(o_1,o_2,a)\}} s''}\ \textsf{indMinusL1}$$

$$\frac{s \sim_{link\ C\ D} s' \quad s' \sim_{link-(o_1,o_2,a)} s'' \quad \neg areLinked(o_1,o_2,a,s) \quad (o_1,o_2,a) \in C \quad (o_1,o_2,a) \notin D}{s \sim_{link\ C-\{(o_1,o_2,a)\}\ D} s''} \ \mathsf{indMinusL2}$$

$$\frac{s \sim_{link\ C\ D} s' \quad s' \sim_{link-o} s'' \quad existsObj(o,s')}{s \sim_{link\ C\ominus o\ (D\ominus o)\cup\{(o,*)\}} s''} \ \mathsf{indMinusLo}$$

The idea behind these constructors is completely analogous to that of the previous relation. Constructors nilnilL and indEqL handle cases where no changes occur. Cases where a link is created are handled by constructors indPlusL1 and indPlusL2. Constructors indMinusL1 and indMinusL2 are used for cases where a link is removed. Finally, when all links concerning an object are removed (i.e. caused by the destruction of that object), constructor indMinusLo is used. Again, there are no rules for base cases involving relations $s \sim_{link+(o_1,o_2,a)} s'$, $s \sim_{link-(o_1,o_2,a)} s'$ and $s \sim_{link-o} s'$. As before, $s \sim_{link} s$ is used.

Relation on Attribute Values

Elements in the relation *valuesChanged* represent the observable differences in terms of attribute values between two states. The domain of this relation is defined as:

`valuesChanged` $\subseteq$ `Sequence(AttValue)`$\times$`State`$\times$`State`

In turn, type AttValue is defined inductively by the following constructors:

$$\frac{o : Object \quad a : Attribute \quad v : Value}{(o,a,v) : AttValue} \ \mathsf{oneAV}$$

$$\frac{o : Object}{(o,+) : AttValue} \ \mathsf{allAVpo} \qquad \frac{o : Object}{(o,-) : AttValue} \ \mathsf{allAVmo}$$

We define the relation such that $valuesChanged(C,s,s')$ holds whenever for every object o the values of its attributes are the same in s' as in s, except in the case that there is a element in C involving o. An instance of AttValue of the form (o,a,v) represents attribute a of o holding value v. An instance of that type of the form $(o,+)$ represents every attribute of o holding its default value. This is the case when o does not exist in s, but it do exist in s'. An instance of AttValue of the form $(o,-)$ represents that values of every attribute of object o are ignored. This is the case when o exists in s, but it does not in s'. We use the following special infix operators:

- $\ominus$: `Sequence(AttValue)`$\times$`Object`$\times$`Attribute`$\rightarrow$`Sequence(AttValue)` where $(sq \ominus (o,a))$ is equal to sequence sq where all triples involving o and a are removed.

- $\oplus$: `Sequence(AttValue)`$\times$`AttValue`$\rightarrow$`Sequence(AttValue)`
  which is a classic *insBack* operator.
- $\overline{\in}$: `AttValue`$\times$`Sequence(AttValue)`$\rightarrow$`Boolean`
  where $t \overline{\in} sq$ is $true$ if the last element of sq in which the object in t occurs is t, and $false$ otherwise.

We denote $valuesChanged(C, s, s')$ as $s \sim_{att\,C} s'$, and define it inductively by the following constructors (for indMinusAVo1 and indMinusAVo2 we assume that c is a class satisfying $isInstanceOf(o, c, s'')$ and a is an attribute satisfying $isAttribute(a, c, structure(s''))$):

$$\frac{s \sim_{att} s'}{s \sim_{att\,[]} s'}\ \mathsf{nilnilAV}$$

$$\frac{s \sim_{att\,C} s' \quad s' \sim_{att} s''}{s \sim_{att\,C} s''}\ \mathsf{indEqAV}$$

$$\frac{\begin{array}{c} s \sim_{att\,C} s' \quad s' \sim_{att-(o,a)} s'' \\ existsObj(o, s') \quad v = getVal(o, a, s'') \end{array}}{s \sim_{att\,(C \ominus (o,a)) \oplus (o,a,v)} s''}\ \mathsf{indMinusAVoa}$$

$$\frac{\begin{array}{c} s \sim_{att\,C} s' \quad s' \sim_{att-o} s'' \quad s' \sim_{obj+o} s'' \quad (o,+) \overline{\notin} C \\ defVal(a, c, structure(s'') = getVal(o, a, s'')) \end{array}}{s \sim_{att\,C \oplus (o,+)} s''}\ \mathsf{indMinusAVo1}$$

$$\frac{s \sim_{att\,C} s' \quad s' \sim_{att-o} s'' \quad s' \sim_{obj-o} s'' \quad (o,-) \overline{\notin} C}{s \sim_{att\,C \oplus (o,-)} s''}\ \mathsf{indMinusAVo2}$$

The first two constructors, nilnilAV and indEqAV, handle cases where no changes are produced in the transition between the states. Constructor indMinusAVoa handles cases where an attribute changed its value. Note that the new value is part of the constructed element. The cases where all attributes of an object are set to their default values (caused by the creation of the object) are handled by constructor indMinusAVo1. Constructor indMinusAVo2 similarly handles the cases where all attribute values of an object should be ignored in the final state. This kind of cases corresponds to object destruction. Finally, $s \sim_{att} s$ holds for every state s, and can be used to construct base cases.

4.3.2 Additional Rules

The relations introduced so far can be used to relate the initial and final states of a program to the changes from one to the other introduced by the program. They are a possible solution to the problem of completely characterize the final state of a program in a practical way. But how do these relations characterize

the final state? The final state is characterized partly by its similarities to the initial state, and partly by the results of the changes introduced by the program. We need a set of rules for inferring explicit properties on the final state as we did in the original framework.
In what follows we define a set of inference rules based on the relations just introduced. Particularly, they allow us to infer the same kind of properties on the final state of a program as rules in section 4.2.3 do. In the premises of the rules, there will occur one of the three relations, some conditions on their collections, as well as properties on the initial state. The conclusion refers to the final state of the program. As usual, we organize the rules into groups according to the property they allow to infer.

Rules for Object Existence

The following set of rules allows the inference of the existence and non-existence of an object in the final state of a program. An object in C exists in the final state, an object in D does not, and an object in neither of them is unchanged with respect to the initial state. For the rules we assume the following symbols: $s, s' : State$; $o : Object$; and $C, D : Set(Object)$. We believe that rules are understood straightforwardly, therefore we omit further comments.

$$\frac{existsObj(o,s) \quad s \sim_{obj\ C\ D} s' \quad o \notin D}{existsObj(o,s')}\ \mathsf{EOr1}$$

$$\frac{\neg existsObj(o,s) \quad s \sim_{obj\ C\ D} s' \quad o \in C}{existsObj(o,s')}\ \mathsf{EOr2}$$

$$\frac{existsObj(o,s) \quad s \sim_{obj\ C\ D} s' \quad o \in D}{\neg existsObj(o,s')}\ \mathsf{EOr3}$$

$$\frac{\neg existsObj(o,s) \quad s \sim_{obj\ C\ D} s' \quad o \notin C}{\neg existsObj(o,s')}\ \mathsf{EOr4}$$

Rules for Object Linkage

The following set of rules allows the inference of whether two objects are linked through an association or not. A triple of the form (o_1, o_2, a) in C means that the two objects are linked, and the same triple in D means that the objects are not linked. An element of the form $(o, *)$ in D means that the object is linked to no other. Linkage of objects not involved in any Link element is unchanged with respect to the initial state. For the rules we assume the following symbols: $s, s' : State$; $o_1, o_2 : Object$; $a : Association$; and $C, D : Set(Link)$.

$$\frac{\begin{array}{c} areLinked(o_1,o_2,a,s) \quad s \sim_{link\ C\ D} s' \\ (o_1,o_2,a) \notin D \quad (o_1,*) \notin D \quad (o_2,*) \notin D \end{array}}{areLinked(o_1,o_2,a,s')}\ \mathsf{ALr1}$$

$$\frac{areLinked(o_1, o_2, a, s) \qquad s \sim_{link\ C\ D} s' \qquad (o_1, o_2, a) \in D}{\neg areLinked(o_1, o_2, a, s')}\ \text{ALr2-1}$$

$$\frac{areLinked(o_1, o_2, a, s) \qquad s \sim_{link\ C\ D} s' \qquad (o_1, *) \in D}{\neg areLinked(o_1, o_2, a, s')}\ \text{ALr2-2}$$

$$\frac{areLinked(o_1, o_2, a, s) \qquad s \sim_{link\ C\ D} s' \qquad (o_2, *) \in D}{\neg areLinked(o_1, o_2, a, s')}\ \text{ALr2-3}$$

$$\frac{\neg areLinked(o_1, o_2, a, s) \qquad s \sim_{link\ C\ D} s' \qquad (o_1, o_2, a) \in C}{areLinked(o_1, o_2, a, s')}\ \text{ALr3}$$

$$\frac{\neg areLinked(o_1, o_2, a, s) \qquad s \sim_{link\ C\ D} s' \qquad (o_1, o_2, a) \notin C}{\neg areLinked(o_1, o_2, a, s')}\ \text{ALr4}$$

Rules for Attribute Values

The following set of rules allows the inference of whether an attribute of an object holds a given value or not. If an object is not involved in any triple of the sequence, then all its attribute values are unchanged with respect to the initial state. If it does, the form and values of the triple in which the object lastly occurs defines what can be inferred. If it occurs in a triple of the form $(o, -)$, then the object does not exist in the final state. If it occurs in a triple of the form $(o, +)$, then the object holds in all its attributes their default value. Finally, a last occurrence of the form (o, a, v) means that o holds in attribute a the value v. For the rules we assume the following symbols: $s, s' : State$; $o : Object$; $a : Attribute$; $v : Value$; $c : Class$; and $C : Sequence(AttValue)$.

$$\frac{\begin{array}{c} v = getVal(o, a, s) \qquad s \sim_{att\ C} s' \\ (o, a, ?)\ \overline{\notin}\ C \qquad (o, +)\ \overline{\notin}\ C \qquad (o, -)\ \overline{\notin}\ C \end{array}}{v = getVal(o, a, s')}\ \text{GVr1}$$

$$\frac{\begin{array}{c} s \sim_{att\ C} s' \qquad isInstanceOf(o, c, s') \\ isAttribute(a, c, structure(s')) \qquad (o, +)\ \overline{\in}\ C \end{array}}{getVal(o, a, s') = defVal(o, c, structure(s'))}\ \text{GVr2}$$

$$\frac{s \sim_{att\ C} s' \qquad (o, a, v)\ \overline{\in}\ C}{getVal(o, a, s') = v}\ \text{GVr3}$$

$$\frac{s \sim_{att\ C} s' \qquad (o, -)\ \overline{\in}\ C}{\neg existsObj(o, s')}\ \text{GVr4}$$

4.3.3 Example

The extension to our framework is complete, we defined special relations to express in a compact way the changes produced by a program on its final state, and a set of rules for inferring properties from the relations about it were provided. Next, we illustrate the use of the extended framework. The example of *hireSalesman* of the previous section is reformulated. In that case, the program creates a new salesman s and connects it to company c and to product p. Here is the proposition we should prove in order to prove the corresponding correctness assertion:

$$\frac{\begin{array}{l}\neg existsObj(s, s_1)\\ \neg isAbstract(Salesman, structure(s_1))\\ works_for \neq sells\\ areLinked(c, p, produces, s_1)\\ s_2 = create(s, Salesman, s_1)\\ s_3 = link(c, s, works_for, s_2)\\ s_4 = link(s, p, sells, s_3)\end{array}}{\begin{array}{c}objectsChanged(\{s\}, \emptyset, s_1, s_4)\wedge\\ linksChanged(\{(c, s, works_for), (s, p, sells)\}, \emptyset, s_1, s_4)\wedge\\ valuesChanged([(s, +)], s_1, s_4)\end{array}}$$

Our goal is a conjunction of the three relations. The first conjunct specifies that the only changes in state s_4 regarding objects with respect to state s_1 is that object s was created. Similarly, there are two new links, between c and s through works_for, and between s and p through sells. No links were removed. Finally, if class Salesman had attributes, the only changes in attribute values would have been attributes of s, having their default value.
Now we prove only the conjunct concerning the *linksChanged* relation. Despite being possible to reason in a top-down fashion, in this example we perform bottom-up inference. This means that we combine the solutions of known subproblems into the main goal. The annotated correctness assertion for the chosen conjunct is:

$$\begin{array}{l}\{\ \neg existsObj(s, s_1) \wedge \neg isAbstract(Salesman, structure(s_1))\wedge\\ \quad works_for \neq sells \wedge areLinked(c, p, produces, s_1)\wedge\\ \quad linksChanged(\emptyset, \emptyset, s_1, s_1), \text{Proof 0}\ \}\\ \qquad s_2 = create(s, Salesman, s_1);\\ \qquad \{\ linksChanged(\emptyset, \emptyset, s_1, s_2), \text{Proof 1}\ \}\\ \qquad s_3 = link(c, s, works_for, s_2);\\ \qquad \{\ linksChanged((c, s, works_for), \emptyset, s_1, s_3), \text{Proof 2}\ \}\\ \qquad s_4 = link(s, p, sells, s_3);\\ \{\ linksChanged(\{(c, s, works_for), (s, p, sells)\}, \emptyset, s_1, s_4), \text{Proof 3}\ \}\end{array}$$

and the proofs are presented below.

Proof 0: $linksChanged(\emptyset, \emptyset, s_1, s_1)$
This trivially holds. Since $s \sim_{link} s$ holds for every state s, and particularly for s_1, we apply constructor nilnilL to s_1 and prove the thesis.

Proof 1: $linksChanged(\emptyset, \emptyset, s_1, s_2)$
From axiom AxStt5 we can prove that $s_2 = create(s, Salesman, s_1)$ implies $s_1 \sim_{link} s_2$. We can apply constructor indEqL to prove the thesis.

Proof 2: $linksChanged((c, s, works_for), \emptyset, s_1, s_3)$
From axiom AxStt7 we can prove that $s_3 = link(c, s, works_for, s_2)$ implies both $\neg areLinked(c, s, works_for, s_2)$ and $s_2 \sim_{link+(c,s,works_for)} s_3$. We can apply constructor indPlusL1, since $(c, s, works_for) \notin \emptyset$, to prove the thesis.

Proof 3: $linksChanged(\{(c, s, works_for), (s, p, sells)\}, \emptyset, s_1, s_4)$
From axiom AxStt7 we can prove that $s_4 = link(s, p, sells, s_3)$ implies both $s_3 \sim_{link+(s,p,sells)} s_4$ and $\neg areLinked(s, p, sells, s_3)$. Since $(s, p, sells) \notin \emptyset$ and $(s, p, sells) \notin \{(c, s, works_for)\}$, we can apply again indPlusL1 to prove the thesis.

To complete our example, now we apply additional rules to infer the same properties as we did in the previous version. From $\neg existsObj(s, s_1)$ from the precondition, $objectsChanged(\{s\}, \emptyset, s_1, s_4)$ from the recently proved postcondition, and $s \in \{s\}$, applying rule EOr2 we prove $existsObj(s, s_4)$. Then from $\neg areLinked(c, s, works_for, s_1)$ which is implied by $\neg existsObj(s, s_1)$ from the precondition, with $linksChanged(\{(c, s, works_for), (s, p, sells)\}, \emptyset, s_1, s_4)$ from the postcondition, and $(c, s, works_for) \in \{(c, s, works_for), (s, p, sells)\}$, applying rule ALr3 we prove $areLinked(c, s, works_for, s_4)$. We complete our example applying ALr3 to prove $areLinked(s, p, sells, s_4)$. In the application we use $\neg areLinked(s, p, sells, s_1)$ which is also implied by $\neg existsObj(s, s_1)$, again $linksChanged(\{(c, s, works_for), (s, p, sells)\}, \emptyset, s_1, s_4)$ from the postcondition, and $(s, p, sells) \in \{(c, s, works_for), (s, p, sells)\}$, .

4.4 Reasoning about State Well-formedness

In this section we explore the applicability of the framework to reason about state well-formedness. First we analyze system invariants and propose a mechanism to integrate them into the framework. Then, the support for multiplicities is discussed.

4.4.1 System Invariants

The UML Reference Manual [RJB98, p. 317] defines the notion of *invariant* as a constraint that must be true at all times, and then weakens the definition by adding that the constraint must be true at least when no operation is incomplete. In turn, applying to the entire system the definition by Bertrand Meyer

of class invariant [Mey97, p. 363], invariants express global properties of states, characterizing deeper semantic properties and integrity constraints. Then he adds that invariants must be preserved by all operations.
Analyzing these, at least not incompatible, definitions we identify two fundamental issues, one giving sense to the other. We discuss them separately. The first issue is, *what* is really an invariant? We could summarize that it is a constraint, particularly expressing some semantic properties about states. To illustrate this, let us use again the example we developed in the last chapters. A semantic condition we are not able to express in the system structure, and thus remained unconsidered so far is that the product that a salesman sells should be produced by the company in which he or she works. We can express such a condition in our framework as:

$$\begin{aligned} I \equiv\ & (\forall stt : State; \forall c, p, s : Object) \\ & ((areLinked(c, s, works_for, stt) \wedge areLinked(s, p, sells, stt)) \\ & \Rightarrow areLinked(c, p, produces, stt)) \end{aligned}$$

Now the second issue is, *when* does such a constraint must be satisfied? Since states are continuously evolving, the answer may not be obvious. However, it can be stated that invariants can be broken during the execution of an operation, but it is mandatory to reestablish them at completion. Then it seems reasonable to ask that for every program in a system, invariants must be satisfied right before a program starts its execution, and immediately after its completion. More precisely, they must be satisfied for the initial state and for the final state. A natural form to ensure this is by adding the invariants to every correctness assertion:

$$\{\ P \wedge I\ \}\ S\ \{\ Q \wedge I\ \}$$

Note that for the occurrence of I in the precondition, the variable *stt* in the invariant definition is bound to the initial state, and similarly in the postcondition, *stt* is bound to the final state. Let us show the concrete proposition to prove in this case. For clarity, we only concern ourselves about invariant preservation thus omitting the properties already proved in sections 4.2.4 and 4.3.3. We also omit properties on the structure of s_1, despite they are required for the actual proof:

$$\frac{\begin{array}{l} (\forall c', p', s' : Object) \\ \quad ((areLinked(c', s', works_for, s_1) \wedge areLinked(s', p', sells, s_1)) \\ \quad \Rightarrow areLinked(c', p', produces, s_1)) \\ s_2 = create(s, Salesman, s_1) \\ s_3 = link(c, s, works_for, s_2) \\ s_4 = link(s, p, sells, s_3) \end{array}}{\begin{array}{l} (\forall c'', p'', s'' : Object) \\ \quad ((areLinked(c'', s'', works_for, s_4) \wedge areLinked(s'', p'', sells, s_4)) \\ \quad \Rightarrow areLinked(c'', p'', produces, s_4)) \end{array}}$$

In conclusion, our framework gives support to the notion of system invariants according to the approach discussed in this section. If we are able to express an invariant within our framework, then we will have the possibility to prove whether a program preserves it or not.

4.4.2 Multiplicities

Multiplicities are semantic restrictions on states which can be understood as a particular case of system invariants. As discussed in the previous chapter, multiplicities aim at restricting the cardinality of the set of objects that are linked to every object in the state. For example, in the case of the system studied in this text, the cardinal of the set of companies which are allowed to be linked to a given person is exactly 1. This does not restrict which company is linked to a particular person, just demands that there must be one. Modifications to a state like those that are performed by the primitives may have effect on the fulfillment of the multiplicities. For example, if we remove the link between a person and its corresponding company, the multiplicity specified for class person is violated, making the whole state ill-formed. The discussion in the previous section for invariants applies to multiplicities, in the sense that well-formedness with respect to multiplicities may be broken during the execution of an operation, but it must be reestablished by the time of its completion. This means that for dealing with this form of well-formedness, just as for invariants, it may be assumed in the precondition, but must be ensured in the postcondition. For our framework to support reasoning about well-formedness, it would be necessary to extend our assertion language including operation $isWellFormed$ to it. In such a case, an extended correctness assertion would be written as:

$$\{\, P \wedge I \wedge isWellFormed \,\}\ S\ \{\, Q \wedge I \wedge isWellFormed \,\}$$

As before, the precondition P, invariant I and function $isWellFormed$ are expressed in terms of the initial state, while on the other hand postcondition Q, invariant I and function $isWellFormed$ are expressed in terms of the final state. An annotated correctness assertion corresponding to $hireSalesman$ where only multiplicity related well-formedness is to be proved could simply be:

$$
\begin{array}{l}
\{\, isWellFormed(s_1) \,\} \\
\quad s_2 = create(s, Salesman, s_1); \\
\quad \{\, \neg isWellFormed(s_2), \text{Proof 1} \,\} \\
\quad s_3 = link(c, s, works_for, s_2); \\
\quad \{\, \neg isWellFormed(s_3), \text{Proof 2} \,\} \\
\quad s_4 = link(s, p, sells, s_3); \\
\{\, isWellFormed(s_4), \text{Proof 3} \,\}
\end{array}
$$

Note that in this case all intermediate states “happen” to be ill-formed. However, for some other programs, all intermediate states could be well-formed, or even some could be well-formed and the rest ill-formed. This use of our framework imposes a requirement which must be fulfilled. We now need some special

rules for justifying each step of inference. In this work we investigated under what conditions it is possible to ensure that the resulting state of the application of a primitive is well-formed or not, in both cases where the initial state is well-formed too and when it is not. The propositions introduced next provide necessary and sufficient conditions for well-formedness of the state delivered as a result by a primitive. We stressed all possible cases, so therefore a complete set of inference rules may be extracted from them. The proofs for all the propositions can be found in Appendix B. They are thorough, although they were written in natural language.
As final remark, as discussed in section 4.2.5, postconditions are commonly underspecified, and so happens with preconditions for similar reasons. As the reader may soon notice, some of the hypotheses required by the propositions could be very demanding. In scenarios where states are partially specified by developers, the chances of a practical application of this approach for reasoning about well-formedness could be compromised.

Modification of Well-formed States

Proposition When creating an instance in a well-formed state, it is a necessary and sufficient condition for the resulting state to be well-formed that the minimum of all opposite multiplicities through all the possible associations with respect to the class instantiated is zero. In symbols:

$$
\begin{array}{l}
isWellFormed(s) \wedge s' = create(o, c, s) \Rightarrow \\
\quad ((((\forall a : Association \mid existsAssociation(a, structure(s)) \wedge \\
\quad\quad associates(a, c, c', structure(s)); \\
\quad\quad \forall m : Multiplicity \mid multiplicities(a, m', m, structure(s))) \\
\quad\quad\quad (min(m) = 0)) \wedge \\
\quad\quad ((\forall a : Association \mid existsAssociation(a, structure(s)) \wedge \\
\quad\quad associates(a, c', c, structure(s)); \\
\quad\quad \forall m : Multiplicity \mid multiplicities(a, m, m', structure(s))) \\
\quad\quad\quad (min(m) = 0))) \Leftrightarrow isWellFormed(s'))
\end{array}
$$

Proposition When destroying an instance in a well-formed state, it is a necessary and sufficient condition for the resulting state to be well-formed that every instance of every class associated to the class of the object to be destroyed is linked at least to a number of objects (excluding the one to be destroyed) that equals the minimum of the multiplicity at its opposite end. In symbols:

$$
\begin{array}{l}
isWellFormed(s) \wedge s' = destroy(o, s) \Rightarrow \\
\quad ((((\forall a : Association \mid associates(a, c, c', structure(s)) \wedge \\
\quad\quad\quad isInstanceOf(o, c, s); \\
\quad\quad \forall m : Multiplicity \mid multiplicities(a, m, m', structure(s)); \\
\quad\quad \forall o' : Object \mid isInstanceOf(o', c', s)) \\
\quad\quad\quad (|\{o'' : Object \mid (o'' \neq o) \wedge \\
\quad\quad\quad\quad\quad isInstanceOf(o'', c, s) \wedge
\end{array}
$$

$$
\begin{aligned}
&\qquad\qquad\qquad areLinked(o'', o', a, s)\}| \geq min(m))\wedge \\
&(((\forall a : Association \mid associates(a, c', c, structure(s))\wedge \\
&\quad isInstanceOf(o, c, s); \\
&\forall m : Multiplicity \mid multiplicities(a, m', m, structure(s)); \\
&\forall o' : Object \mid isInstanceOf(o', c', s)) \\
&\quad (|\{o'' : Object \mid (o'' \neq o)\wedge \\
&\qquad\qquad isInstanceOf(o'', c, s)\wedge \\
&\qquad\qquad areLinked(o', o'', a, s)\}| \geq min(m)))))) \\
&\Leftrightarrow isWellFormed(s'))
\end{aligned}
$$

Proposition When creating a link between two instances through an association in a well-formed state, it is a necessary and sufficient condition for the resulting state to be well-formed that the number of objects linked through the association to both objects is strictly less than the maximum of its opposite multiplicity. In symbols:

$$
\begin{aligned}
&isWellFormed(s) \wedge s' = link(o_1, o_2, a, s) \Rightarrow \\
&\quad (((\forall m_1, m_2 : Multiplicity \mid multiplicities(a, m_1, m_2, strucutre(s))) \\
&\qquad (|\{o : Object \mid areLinked(o_1, o, a, s)\}| < max(m_2)\wedge \\
&\qquad |\{o : Object \mid areLinked(o, o_2, a, s)\}| < max(m_1))) \\
&\quad \Leftrightarrow isWellFormed(s'))
\end{aligned}
$$

Proposition When removing a link between two instances through an association in a well-formed state, it is a necessary and sufficient condition for the resulting state to be well-formed that the number of objects linked through the association to both objects is strictly greater than the minimum of its opposite multiplicity. In symbols:

$$
\begin{aligned}
&isWellFormed(s) \wedge s' = unlink(o_1, o_2, a, s) \Rightarrow \\
&\quad (((\forall m_1, m_2 : Multiplicity \mid multiplicities(a, m_1, m_2, strucutre(s))) \\
&\qquad (|\{o : Object \mid areLinked(o_1, o, a, s)\}| > min(m_2)\wedge \\
&\qquad |\{o : Object \mid areLinked(o, o_2, a, s)\}| > min(m_1))) \\
&\quad \Leftrightarrow isWellFormed(s'))
\end{aligned}
$$

Proposition When updating the value of an attribute of an instance in a well-formed state, the resulting state is well-formed. In symbols:

$$
\begin{aligned}
&isWellFormed(s) \wedge s' = set(o, a, v, s) \Rightarrow \\
&\quad isWellFormed(s')
\end{aligned}
$$

Modification of Ill-formed States

Proposition When creating an instance in an ill-formed state, the resulting state is ill-formed. In symbols:

$$\neg isWellFormed(s) \land s' = create(o, c, s) \Rightarrow$$
$$\quad \neg isWellFormed(s')$$

Proposition When destroying an instance in an ill-formed state, it is a necessary and sufficient condition for the resulting state to be well-formed that (i) every object of the state (except the one to be removed) satisfies the multiplicities constraints (i.e. the object to be removed is the only cause for the ill-formedness of the state) and (ii) every object linked to it through any association is unaffected by losing a link (i.e. after being unlinked from the object to be destroyed they still satisfy the multiplicities). In symbols:

$$
\begin{array}{l}
\neg isWellFormed(s) \land s' = destroy(o, s) \Rightarrow \\
\quad ((isWellFormed(s)|_o \land \\
\quad (((\exists a : Association \mid associates(a, c, c', structure(s)) \land \\
\quad\quad isInstanceOf(o, c, s); \\
\quad\quad \forall m : Multiplicity \mid multiplicities(a, m, m', structure(s))) \\
\quad\quad\quad (\neg inRange(|\{o' : Object \mid areLinked(o, o', a, s)\}|, m'))) \lor \\
\quad ((\exists a : Association \mid associates(a, c, c', structure(s)) \land \\
\quad\quad isInstanceOf(o, c', s); \\
\quad\quad \forall m : Multiplicity \mid multiplicities(a, m, m', structure(s))) \\
\quad\quad\quad (\neg inRange(|\{o' : Object \mid areLinked(o', o, a, s)\}|, m)))) \land \\
\quad ((\forall a : Association \mid associates(a, c, c', structure(s)) \land \\
\quad\quad isInstanceOf(o, c, s); \\
\quad\quad \forall m : Multiplicity \mid multiplicities(a, m, m', structure(s)); \\
\quad\quad \forall o' : Object \mid isInstanceOf(o', c', s)) \\
\quad\quad\quad (|\{o'' : Object \mid (o'' \neq o) \land areLinked(o'', o', a, s)\}| \geq min(m)) \land \\
\quad ((\forall a : Association \mid associates(a, c, c', structure(s)) \land \\
\quad\quad isInstanceOf(o, c', s); \\
\quad\quad \forall m : Multiplicities \mid multiplicities(a, m', m, structure(s)); \\
\quad\quad \forall o' : Object \mid isInstanceOf(o', c, s)) \\
\quad\quad\quad (|\{o'' : Object \mid (o'' \neq o) \land areLinked(o', o'', a, s)\}| \geq min(m')))) \\
\quad \Leftrightarrow isWellFormed(s'))
\end{array}
$$

Proposition When creating a link between two instances through an association in an ill-formed state, it is a necessary and sufficient condition for the resulting state to be well-formed that every object of the state (except for at least one of those involved in the primitive) satisfies the multiplicities, and at least one of them needs one link through the association to satisfy the multiplicities. In symbols:

$$
\begin{array}{l}
\neg isWellFormed(s) \land s' = link(o_1, o_2, a, s) \Rightarrow \\
\quad ((isWellFormed(s)|_a \land \\
\quad ((\forall c_1, c_2 : Class \mid associates(a, c_1, c_2, structure(s)); \\
\quad\quad \forall m_1, m_2 : Multiplicity \mid multiplicities(a, m_1, m_2, structure(s))) \\
\quad\quad\quad ((\forall o : Object \mid (o \neq o_1) \land isInstanceOf(o, c_1, s)) \\
\quad\quad\quad\quad (inRange(|\{o' : Object \mid areLinked(o, o', a, s)\}|, m_2)) \land
\end{array}
$$

$$
\begin{aligned}
&(\forall o : Object \mid (o \neq o_2) \land isInstanceOf(o, c_2, s)) \\
&\quad (inRange(|\{o' : Object \mid areLinked(o', o, a, s)\}|, m_1))))\land \\
&(max(m_2) > |\{o : Object \mid areLinked(o, o_2, a, s)\}| \geq min(m_2) - 1 \lor \\
&\ max(m_1) > |\{o : Object \mid areLinked(o_1, o, a, s)\}| \geq min(m_1) - 1)) \\
&\Leftrightarrow isWellFormed(s'))
\end{aligned}
$$

Proposition When removing a link between two instances through an association in an ill-formed state, it is a necessary and sufficient condition for the resulting state to be well-formed that every object of the state (except for at least one of those involved in the primitive) satisfies the multiplicities, and at least one of them exceeds by one the maximum of the multiplicity with respect to the association. In symbols:

$$
\begin{aligned}
&\neg isWellFormed(s) \land s' = unlink(o_1, o_2, a, s) \Rightarrow \\
&\quad ((isWellFormed(s)|_a \land \\
&\quad ((\forall c_1, c_2 : Class \mid associates(a, c_1, c_2, structure(s)); \\
&\quad\ \ \forall m_1, m_2 : Multiplicity \mid multiplicities(a, m_1, m_2, structure(s))) \\
&\quad\quad ((\forall o : Object \mid (o \neq o_1) \land isInstanceOf(o, c_1, s)) \\
&\quad\quad\quad (inRange(|\{o' : Object \mid areLinked(o, o', a, s)\}|, m_2))\land \\
&\quad\quad (\forall o : Object \mid (o \neq o_2) \land isInstanceOf(o, c_2, s)) \\
&\quad\quad\quad (inRange(|\{o' : Object \mid areLinked(o', o, a, s)\}|, m_1))))\land \\
&\quad (min(m_2) < |\{o : Object \mid areLinked(o, o_2, a, s)\}| \leq max(m_2) + 1 \lor \\
&\quad\ min(m_1) < |\{o : Object \mid areLinked(o_1, o, a, s)\}| \leq max(m_1) + 1)) \\
&\quad \Leftrightarrow isWellFormed(s'))
\end{aligned}
$$

Proposition When updating the value of an attribute of an instance in an ill-formed state, the resulting state is ill-formed. In symbols:

$$
\begin{aligned}
&\neg isWellFormed(s) \land s' = set(o, a, v, s) \Rightarrow \\
&\quad \neg isWellFormed(s')
\end{aligned}
$$

Summary

In this chapter, we introduced a framework for reasoning about the use of primitives. We presented the notion of program, which was defined as a finite composition of primitives, for representing simple imperative programs. We described an assertion language composed by queries associated to the types specified in the previous chapter. In combination, these two elements allow the formulation of correctness assertions. Such formulae can be proved by inference using a set of axioms corresponding to those that defined the semantics of the primitives, and a set of inference rules specifically introduced. Program correctness is then inferred and not computed. An extension to the framework was also presented. It provides a practical means for specifying the exact effect of program on states. For that, new constructs were added to the assertion language, and new rules were provided for the inference of state properties. Finally, the support of the

framework for reasoning about well-formedness was explored. Proof of preservation for some form of invariants was shown to be available in the current version of the framework. We introduced a number of propositions from whom new rules can be generated in order to infer state well-formedness with respect to multiplicities. In the next chapter, we report the formalization of the framework in the System Coq.

Chapter 5

Formalization of the Framework in Coq

Our theory is completely introduced and our framework can be used for reasoning about system state modification primitives. However, any proof should be constructed and verified by hand. The last step of our work consists in formalizing our framework in a proof assistant. Such a formalization carries over all the benefits we reviewed in Chapter 2. A precise meaning is assigned to every element in our theory, and proofs are taken to a more rigorous level in which mechanical verification is possible. In addition, the process of construction of proofs is aided by the proof assistant. The scope of the formalization includes the system defined in section 4.2 and the extension introduced in section 4.3. Formulae such as those presented in sections 4.2.4, 4.3.3 and 4.4.1 can be handled. A formalization of the ideas discussed in section 4.4.2 was not included in this version of our work. In this chapter we report the formalization of our framework using the Calculus of Inductive Constructions supported by the System Coq version 7.3 [INR02]. We discuss the criteria for representing our ADT-based specification in that formal language, and illustrate some key points of the results. We also report on some simple experiments carried out using the System Coq for testing the feasibility of automatic proof construction in our framework.
This chapter is structured as follows. Section 5.1 includes our criteria for representing our theory in the Calculus of Inductive Constructions. Section 5.2 shows the key points in the formalized specification of our framework. Results on automatic proof generation are reported in section 5.3.

5.1 Formalization of Abstract Data Types

In this section we show the criteria adopted for representing our specification in the Calculus of Inductive Constructions. The ideas are organized for matching the structure of the ADT-based specification introduced in Chapter 3.

5.1.1 Types

In our specification, all types were defined abstractly. This means that we did not provide an internal representation for any of them. We formalized types as parameters of type Set. For example, in code:

```
Parameter Class :  Set.
```

In this way a type is declared and not defined. If at one point a representation for a type is to be given, this declaration is replaced for an actual definition. Its type would still be Set, and an internal representation is then supported. Equalities for all the types are decidable, therefore each type definition is followed by an axiom expressing such property. For example:

```
Axiom EqClass : (c1,c2:Class){c1 = c2} + ~{c1 = c2}
```

5.1.2 Functions

In our specification many functions were specified as partial. In Coq partial functions are not supported, and the only way to interpret them is as relations, most commonly, inductively defined. The reasons for this are discussed in a later section. In our framework, functions were defined as inductive predicates. For example, in code:

```
Inductive existsClass [c:Class;S:System] :  Prop := ...
```

5.1.3 Axioms

In ADTs, axioms are assumptions we make about the behavior of functions. In some cases, they express some conditions that must be satisfied, and in others, they completely define an operation. Axioms like those defined in association to ADTs are of the form *func* $\Leftrightarrow$ *def*, and an example of them could be axiom AxSys2 which defines function *existsClass*. As said before, the function is formalized as an inductive predicate, and the definition of the predicate is the *def* part of the axiom statement. In particular, if *def* is a disjunction, then each disjunct introduces its own constructor in the inductive definition. By this means, we ensure that those are the only cases where *func* is defined. For example:

```
Inductive existsClass [c:Class;S:System] :  Prop :=
exClBase:
     (b:bool)
     (EX S':System | (addClass c b S' S))
        ->(existsClass c S)
⋮
```

5.1.4 Preconditions

As discussed before, partial functions are formalized in Coq as relations, most commonly defined as inductive predicates. Inductive definitions in Coq support the closure property. This means that an element is an object of an inductively defined predicate if and only if it can be generated according to the formation rules (constructors). This is enough to read the formation rules backwards to derive the necessary conditions for a given instance to hold [CT96]. In Coq this is called *inversion*. Then extra arguments can be added to constructors for representing preconditions. In this way, to construct an instance of an inductive predicate which was originated from a partial function, we must provide a proof for each precondition, otherwise the introduction rule could not be applied. Conversely, from an instance of such predicate, it is possible to perform an inversion to get proofs of its preconditions. For example, in code:

```
Inductive destroy :  Object->State->State->Prop :=
consDs:
     (o:Object;s,s':State)
     (existsObj o s)->(Rel_objminuso o s s')->
     (Rel_linkminuso o s s')->(Rel_attminuso o s s')
        ->(destroy o s s').
```

The inductive definition shown above is the formalization of the partial function *destroy* of type State. Note that its definition is very similar to that of axiom AxStt6. The argument of type `(existsObj o s)` in its constructor is due to precondition PreStt5. Then, a proof of `(destroy o s s')` without a proof of `(existsObj o s)` would never be constructed, and from a proof of `(destroy o s s')`, a proof of `(existsObj o s)` is obtained using the tactic `Inversion`. Additional information on inductive definitions can be found in [Gim98].
The primitives could have been alternatively represented as different constructors in a single inductive definition, enabling a separate definition of pre- and postconditions for each primitive, and an explicit representation of the language used to write programs. However, as discussed above, the set of possible primitives becomes unnecessarily closed, and the incorporation of new primitives would require the update of existing proofs. As the framework is intended to be further extended with new primitives, we preferred to define them separately.

5.1.5 Other Definitions

In our specification, other elements were defined besides functions, axioms and preconditions. In section 3.4.3 we defined a set of relations between states. These relations were defined as Coq terms, using the `Definition` construct. For example, specified the relation $s \sim_{obj} s'$ as:

```
Definition Rel_obj :  State->State->Prop
     := [s,s':State]
        (o:Object)(existsObj o s) <-> (existsObj o s').
```

5.2 Formalization of the Framework

In this section we address the formalization of the framework. We produced seven Coq files with approximately five thousands lines of code. This amount includes both definitions and proof scripts. About 15% of the code is dedicated to infrastructure (i.e. types, functions and axioms), and the rest corresponds to the framework itself (i.e. predicates, inference rules together with their proofs). We summarize the key points of the formalized specification. Finally, we discuss a number of issues concerning the formalization process.

5.2.1 The Core

In what follows, we summarize the key aspects of the formalization of the core part of the framework introduced in section 4.2.

Inference Rules

Inference rules were formalized as lemmas and proved making extensive use of the tactic `Inversion` on the premises concerning the primitives. As an example, this is the type of rule `Ex2Dt`:

```
Lemma RuleExists2vDestroy:
    (o,o':Object;s,s':State)
    (existsObj o s)->~o=o'->(destroy o' s s')
      ->(existsObj o s').
```

Correctness Assertions

The concrete proposition derived from the correctness assertions are expressed as theorems in Coq. Here is the formalized version of the correctness assertion for the *hireSalesman* example.

```
Theorem correct_hireSalesman:
    (s,c,p:Object;Salesman:Class;
    sells,works_for,produces:Association;s1,s2,s3,s4:State)
    (existsObj c s1)->(existsObj p s1)->~(existsObj s s1)->
    ~(isAbstract Salesman (structure s1))->
    ~works_for=sells->(areLinked c p produces s1)->
    (create s Salesman s1 s2)->
    (link c s works_for s2 s3)->
    (link s p sells s3 s4)->
      (existsObj s s4) /\ (areLinked c s works_for s4) /\
      (areLinked s p sells s4).
```

Note that variables need to be explicitly quantified. The next three lines after the quantifications correspond to preconditions, the next three to the program

and the last two to postconditions. To show some of the flavor of the proof assistant, here is a proof script for the theorem above.

```
Do 11 Intro.
Intros ExObj_c ExObj_p NExObj_s NAbs_Sales.
Intros Neq_wf_s AL_cp Cr_s Lk_cs Lk_sp.
Split.
Apply (RuleExists2vLink s s p sells s3 s4).
Apply (RuleExists2vLink s c s works_for s2 s3).
Apply (RuleExists1 s Salesman s1 s2 NExObj_s NAbs_Sales Cr_s).
Do 2 Assumption.
Split.
Apply (RuleAreLinked2vLink1 c s s p works_for sells s3 s4).
Apply (RuleAreLinked1 c s works_for s2 s3 Lk_cs).
Do 2 Assumption.
Apply (RuleAreLinked1 s p sells s3 s4 Lk_sp).
```

The first three lines introduces the preconditions and the program as hypotheses, leaving the postcondition as the current goal. Using `Split`, we replace the current goal with two subgoals: one with the leftmost conjunct in the postcondition (i.e. `(existsObj s s4)`), and another with the other two (i.e. `(areLinked c s works_for s4) /\ (areLinked s p sells s4)`). As showed in section 4.2.4, the first subgoal is proved by applying rules `Ex2Lk` twice and `Ex1` once. Subgoals `~(existsObj s s1)` and `~(isAbstract Salesman (structure s1))` generated from the application of `Ex1` are trivially proved using tactic `Assumption` since they are hypotheses. The second subgoal is split into conjuncts `(areLinked c s works_for s4)` and `(areLinked s p sells s4)` respectively. They are similarly proved using rules `AL2Lk` and `AL1`.
Propositions that state program correctness share the same overall structure: queries on the initial state and successive state changes as hypotheses, and a conjunction of queries on the final state as the goal. In addition, proofs share the same structure too and are constructed applying top-down reasoning, generally in a very similar fashion, by splitting conjuncts and applying inference rules until premises can be assumed. These commonalities inspired the experiment on automatic proof construction discussed in section 5.3.

5.2.2 The Extension

Next is a summary of the key ideas of the formalization of the extension to the framework presented in section 4.3. New relations for specifying the exact effect of programs were defined, and also a set of additional rules for inferring properties about final states from them.

Relations between States

Recall that, relations *objectsChanged*, *linksChanged* and *valuesChanged* were introduced for their use in postconditions. Those relations involve collections that register changes produced from one state to its subsequent. Proposition *objectsChanged* uses sets of objects, while *linksChanged* and *valuesChanged* use sets and sequences of some specific elements respectively. Both kind of elements are defined inductively. For example, the type Link used in *linksChanged* is defined as follows:

```
Inductive Link :  Set :=
     oneL : Object->Object->Association->Link
     | allL : Object->Link.
```

Here, the triple is composed by two objects and one association. The second constructor of the type allows the introduction of the special value $(o, *)$, which represents "all the links where o participates".
In turn, the three relations are also defined inductively from the specification. For example, relation *linksChanged*, denoted as $s \sim_{link\ C\ D} s'$, is partially specified as follows:

```
Inductive linksChanged :
 SetLink->SetLink->State->State->Prop :=
 nilnilL:
    (s,s':State)(Rel_link s s')
       ->(linksChanged (empty Link) (empty Link) s s')
 | indEqL:
    (C,D:SetLink;s,s',s'':State)
    (linksChanged C D s s')->(Rel_link s' s'')
       ->(linksChanged C D s s'')
 ⋮
```

where SetLink is a specification of a set of instances of type Link, and Rel_link is the relation $\sim_{link}$.

Correctness Assertions

The formulation of correctness assertions using the extension is carried out straightforwardly. The precondition and the program occur similarly as in the previous section, and the postcondition is replaced by a conjunction of applications of the defined relations between states with appropriate arguments. The *hireSalesman* example reformulated in section 4.3.3 can be formalized as follows. For clarity, only the application of *linksChanged* is shown, and `Ins` is used as a shortcut for `InsertSetLink`.

```
Theorem correct_hireSalesman2:
    (s,c,p:Object;Salesman:Class;
    sells,works_for,produces:Association;s1,s2,s3,s4:State)
    (existsObj c s1)->(existsObj p s1)->~(existsObj s s1)->
    ~(isAbstract Salesman (structure s1))->
    ~works_for=sells->(areLinked c p produces s1)->
    (create s Salesman s1 s2)->
    (link c s works_for s2 s3)->
    (link s p sells s3 s4)->
       (linksChanged
          (Ins (Ins (emptyLink) (oneL c s works_for))
             (oneL s p sells))
          (empty Link) s1 s4).
```

Additional Rules

Finally, as we did with the set of derivation rules in the previous section, all the rules in the additional rules were represented as lemmas, also directly from the specification. As an example, rule ALr3 which allows proving that two objects are linked through an association was specified as:

```
Lemma ALr3:
    (o1,o2:Object;a:Association;C,D:SetLink;stt1,stt2:State)
    ~(areLinked o1 o2 a stt1)->(linksChanged C D stt1 stt2)->
    (MemberSetLink C (oneL o1 o2 a))->
       ->(areLinked o1 o2 a stt2).
```

Now, having a proof of `correct_hireSalesman2` it is possible to prove `(areLinked c s works_for s4)`, the second conjunct proved in the previous section. Assuming the same preconditions as those for the original correctness assertion, we simply apply rule ALr3. For the application, we bind the variable $o1$ to c, $o2$ to s, a to $works_for$, D to $(emptyLink)$, $stt1$ to $s1$, and $stt2$ to $s4$. Variable C is bound to $(Ins\,(Ins\,(emptyLink)\,(oneL\,c\,s\,works_for))\,(oneL\,s\,p\,sells))$. Finally, `~(areLinked o1 o2 a stt1)` is assumed, `(linksChanged C D stt1 stt2)` is exactly `correct_hireSalesman2`, and `(MemberSetLink C (oneL o1 o2 a))` is easily proved.

5.3 Experiments on Automatic Proof Construction

In this section we show the results of our simple experiment on automatic proof construction. The objective of the experiment was to explore the application of some of the tactics provided by Coq for automatic proof generation to simple correctness assertions. As we discussed in section 5.2.1, this kind of propositions

are usually structured as a number of hypotheses, one for every precondition and program step, and the goal is commonly a conjunction of queries on the final state. An approach to proof construction based on top-down inference was outlined in section 4.2.4. There, an inference tree is constructed from its root to the leaves. In such a tree, the root represents the original goal, a change in the level of the tree represents an inference step, and a leaf represents the application of an axiom or a rule where all premises are satisfied by hypotheses. We start by briefly reviewing the tools provided by Coq for automatic proof generation, and then we show how they were applied.

5.3.1 The Coq Approach to Automatizing

In Coq, tactics implement top-down reasoning. A simple example is tactic `Split`, its associated deduction rule reduces a goal like $A \wedge B$ into two subgoals: A and B. An important tactic for us is `Apply` and its variants because it enables top-down reasoning with our own rules. This rule is applied to any term and tries to match the current goal against the conclusion of the term. If it succeeds, then a subgoal is generated for every instantiation of premises of the term. For example, `EApply` postpones to a later moment in the proof the instantiation of variables in the premises that may not be deduced. This is usually the case when dependent quantifications occur. We refer to [CDT02] for further details. There are tactics in Coq that implement heuristics or decision procedures to build a complete proof of a goal. Tactics of this kind, such as `Auto` and `EAuto`, are used for automatic proof generation. Tactic `Auto` implements a resolution procedure to solve the current goal, solving it completely or leaving it intact. It tries to prove the goal directly from hypotheses, or by applying the tactics in a hints database, and proceeding recursively for the generated subgoals. A hint in a database will be tried by `Auto` if the conclusion of the current goal matches the pattern of the hint. In turn `EAuto` is similar to `Auto`, but it uses unification of the goal against the hints, instead of pattern-matching as `Auto`. In [Mon98], a concrete example of `EAuto` is discussed.
Hints are added to a database using the command `Hint`. Adding a term to a database with the `Resolve` option, adds `Apply term` to the hint list. If the inferred type of term contains a dependent quantification, `EApply term` is added instead, and thus the hint will be tried only if `EAuto` is used.

5.3.2 Application to the Framework

The decision procedure implemented by tactics like `Auto` seem to be compatible with the approach to proof construction in our framework. In this simple experiment we tried to prove the proposition `correct_hireSalesman` introduced in section 5.2.1. We added our inference rules to a database hints, as well as all axioms such as `AxObjDiff` introduced in section 5.1.3, because they may be needed to prove an eventual subgoal such as `~(o = o')` introduced by the application of rule Ex2Cr. We added those terms to the database issuing the

following command:

```
Hints Resolve RuleExists1 RuleExists2vCreate ... :  hints.
```

At the moment of adding hints to the database, Coq detected dependent quantifications in our rules; for that reason we tried the `EAuto` variant. The first test was unsuccessful and the goal was left intact. When we tried `EAuto` to atomic goals, after splitting the original conjunction, the tactic successfully proved the three of them. As Coq provides the ability to construct new tactics from existing ones, this simple result was enough to show that it could be worth the effort to develop specific `EAuto`-based tactics for automatic or semiautomatic construction of more complex proofs. For example, a tactic which splits conjuncts occurring in subgoals before applying `EAuto` would suffice to automatically prove `correct_hireSalesman`. Moreover, such tactics would provide an algorithm for the construction of proofs in our framework, which could be of methodological interest.

Summary

In this chapter, we presented the most important issues in the formalization of our framework in Coq. We discussed the specification of every construct used in the definition of our theory. Examples of the most interesting parts of the final specification were also given. Finally, the results of an experiment on automatic proof construction suggest the development of customized tactics in that respect as a possible future improvement to our framework.

Chapter 6

A Case Study

In this chapter we present a case study in which we illustrate a practical use of the framework in a complete example treated in the bibliography [Lar02]. We start by introducing the problem domain and a specification of system behavior for the addressed use case. It consists in a brief description of the use case and a system sequence diagram showing the interaction between the system and external agents along the main success scenario of the use case. A specification of the static structure of the system is presented next. Also, a software contract for every system operation in the diagram is included. We use the framework for formulating a correctness assertion for each system operation. A proof of program correctness for one of them is discussed. Finally we identify a system invariant, and its preservation is justified.

This chapter is structured as follows. In section 6.1 we introduce the example to be developed. The system, and its behavior is specified using UML diagrams, use case descriptions and software contracts. Section 6.2 presents how we used our framework for that case.

6.1 The Point-of-Sale System

In this section we introduce the example and a specification for it. This case is used throughout [Lar02] as its driving case study. That work presents a description of a Unified Process [JBR99] based software development process.
The case is based on a system for a retail store register, also known as a Point-of-Sale, or POS for short. It is a simple problem, however it is realistic because organizations do write POS systems using object technologies. A POS system is a computerized application used (in part) to record sales and handle payments, used typically in a retail store. It includes hardware components such as a computer and a bar code scanner, and software to run the system. It interfaces with various service applications, such as third-party tax calculator, and inventory control. The addressed use case is the *Process Sale* use case, since it is central

in the case study on which we base ours.
An outline of the system behavior for the selected use case is given. Next, we include a specification of the system structure. A detail of the system behavior is finally presented.

6.1.1 The Process Sale Use Case

We start with a textual description for the Process Sale use case, in which only the main success scenario is included. For example, we do not consider the case when a customer asks to cancel the purchase of one of the products. Next a representation of that scenario is shown as an interaction diagram, where the primary actor and the system are involved.

Use Case Description

Use Case: Process Sale

Primary Actor: Cashier

Overview: A Customer arrives at the checkout with items to purchase. The Cashier records the purchase items and collects the payment. On completion, the Customer leaves with the items.

Main Success Scenario:

1. Customer arrives at POS check out with goods to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.
   Cashier repeats steps 3-4 until indicates done.
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
8. System logs completed sale and sends sale and payment information to the external Accounting system and Inventory system.
9. System presents receipt.
10. Customer leaves with receipt an goods.

External System Interaction

The diagram in Figure 6.1 represents the interaction between the primary actor and the system described above. The instance on the left represents the role of a cashier, while the instance on the right represents the running system. Messages are abstractions of information exchange between the participants in the dialogue depicted by the use case description. Time ordering of messages is represented vertically from top to bottom, message makeNewSale occurs before than makePayment. The box around the second message models that addLineItem may occur any number of times before the next message is delivered.

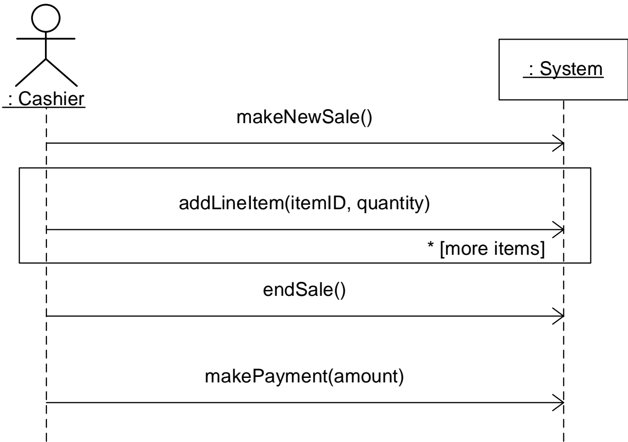

Figure 6.1: System sequence diagram for Process Sale

6.1.2 Point-of-Sale System Structure

The system structure used for the forthcoming specification is illustrated in Figure 6.2. Class Store models the organization that owns the registers where the system runs. The store also logs every completed sale. The register knows its current sale by association captured_on. A sale contains line items for every different product purchased. A line item knows the specification of the product and the number of pieces purchased. Information about the payment is associated to a sale when it is completed.

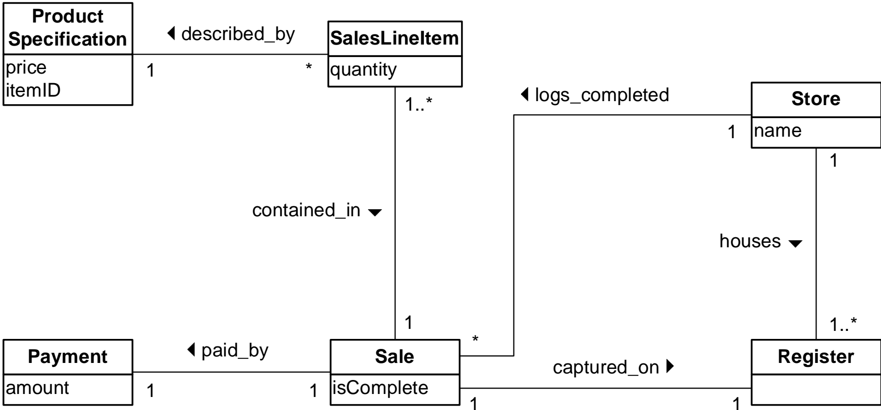

Figure 6.2: POS system structure

The following condition was identified as a system invariant by us, however many others may also exist:

System invariant: *Every logged sale must be paid.*

In other words, every sale which is associated to a store must be also associated to a payment. This system invariant will be specified in our framework later in the present case study.

6.1.3 Software Contracts

In what follows, we include a reduced version, containing essential information only, of the software contracts for the Process Sale use case. Every contract specifies a system operation. The information included is the operation signature, and the pre- and postconditions, which are expressed in terms of the system structure showed in Figure 6.2.

Operation: makeNewSale()
Preconditions: – None
Postconditions: – A Sale instance s was created
– s was associated with the Register
– Attributes of s were initialized

Operation: enterItem(itemID:ItemID,quantity:Integer)
Preconditions: – There is a sale underway
Postconditions: – A SaleLineItem sli was created
– sli was associated with the current Sale
– $sli.quantity$ became quantity
– sli was associated with a ProductSpecification

Operation: endSale()
Preconditions: – There is a sale underway
Postconditions: – $s.isComplete$ became true

Operation: makePayment(amount:Money)
Preconditions: – There is a sale underway
Postconditions: – A Payment instance p was created
– $p.amount$ became amount
– p was associated with the current Sale
– The current Sale was associated with the Store

6.2 The Framework in Practice

The problem is completely presented; now we show how our framework can be used. First, we formulate correctness assertions for every system operation, and the proof of one of them is discussed. We formulate the system invariant identified in the previous section. Finally, we show that the invariant is preserved by the system operations.

6.2.1 Correctness Assertions for Process Sale Operations

Next are the correctness assertions for every system operation. For reasons of clarity, information about the system structure is included in the precondition only when it is required. Names used for structural features correspond to those in Figure 6.2, that is, variable $Sale : Class$ represents class Sale. For the formulae, let variables stt_i denote states, and variables s and r denote arbitrary objects. Variables which are specific to a formula are introduced next to it.

makeNewSale

For the following formula, additionally assume the variable $\mathit{ff} : Value$.

$$\begin{array}{l} \{\, existsObj(r, stt_1) \wedge defVal(isComplete, Sale, structure(stt_1)) = \mathit{ff} \wedge \\ \quad \neg isAbstract(Sale, structure(stt_1)) \,\} \\ \quad\quad stt_2 = create(s, Sale, stt_1); \\ \quad\quad stt_3 = link(s, r, captured_on, stt_2); \\ \{\, existsObj(s, stt_3) \wedge areLinked(s, r, captured_on, s_3) \wedge \\ \quad getVal(s, isComplete, stt_3) = \mathit{ff} \,\} \end{array}$$

enterItem

For the following formula, additionally assume variables sli, $ps : Object$. Operation arguments are represented by variables qty, $itID : Value$.

$$\begin{array}{l} \{\, areLinked(s, r, captured_on, stt_1) \wedge getVal(ps, itemID, stt_1) = itID \,\} \\ \quad\quad stt_2 = create(sli, SalesLineItem, stt_1); \\ \quad\quad stt_3 = link(sli, s, contained_in, stt_2); \\ \quad\quad stt_4 = set(sli, quantity, qty, stt_3); \\ \quad\quad stt_5 = link(ps, sli, described_by, stt_4); \\ \{\, existsObj(sli, stt_5) \wedge areLinked(sli, s, contained_in, stt_5) \wedge \\ \quad getVal(sli, quantity, stt_5) = qty \wedge areLinked(ps, sli, described_by, stt_5) \,\} \end{array}$$

endSale

For the following formula, additionally assume the variable $tt : Value$.

$$\begin{array}{l} \{\, areLinked(s, r, captured_on, stt_1) \,\} \\ \quad\quad stt_2 = set(s, isComplete, tt, stt_1); \\ \{\, getVal(s, isComplete, stt_2) = tt \,\} \end{array}$$

makePayment

For the following formula, additionally assume variables st, p : $Object$. The operation argument is represented by variable $amt : Value$.

$$
\begin{array}{l}
\{ \; areLinked(s, r, captured_on, stt_1) \wedge areLinked(st, r, houses, stt_1) \wedge \\
\quad \neg existsObj(p, stt_1) \wedge \neg isAbstract(Payment, structure(s_1)) \; \} \\
\qquad stt_2 = create(p, Payment, stt_1); \\
\qquad stt_3 = set(p, amount, amt, stt_2); \\
\qquad stt_4 = link(p, s, paid_by, stt_3); \\
\qquad stt_5 = link(st, s, logs_completed, stt_4); \\
\qquad stt_6 = unlink(s, r, captured_on, stt_5); \\
\{ \; existsObj(p, stt_6) \wedge getVal(p, amount, stt_6) = amt \wedge \\
\quad areLinked(p, s, paid_by, stt_6) \wedge areLinked(st, s, logs_completed, stt_6) \\
\quad \neg areLinked(s, r, captured_on, stt_6) \; \}
\end{array}
$$

Conjunct $\neg areLinked(s, r, captured_on, stt_6)$ in this postcondition is not present in the contract defined in [Lar02]. However, we identified it as necessary since a register is linked to the current sale only. Once completed, the sale is logged and the register may not have any associated sale. This is because in the next call to $makeNewSale$ another sale will be associated to the register becoming the new (and only) current sale.

For constructing a proof for this formula we proceed as described in section 4.2.4. For every conjunct in the postcondition we successively apply inference rules propagating the condition upward the program until every generated subgoal is proved by hypotheses. Let us start with $existsObj(p, stt_6)$. Applying rule Ex2Ul we generate according to its premises, subgoals $existsObj(p, stt_5)$ and $stt_6 = unlink(o_1, o_2, a, stt_5)$ for some objects o_1 and o_2, and association a. Since we have as hypothesis $stt_6 = unlink(s, r, captured_on, stt_5)$ the second subgoal is proved by binding o_1 to s, o_2 to r and a to $captured_on$ respectively. For proving the first subgoal we proceed similarly with rule Ex2Lk, which in turn generates subgoals $existsObj(p, stt_4)$, and $stt_5 = link(o_1, o_2, a, stt_4)$. Again the second subgoal is proved by hypothesis, and we proceed with the first one for which we apply rule Ex2Lk once more. Finally, we finish the proof for the conjunct applying rule Ex2St followed by rule Ex1. The premises of the last rule are satisfied by the preconditions.
The rest of the conjuncts are proved in a very similar fashion. For conjunct $getVal(p, amount, stt_6) = amt$ we apply successively rules GV3Ul, GV3Lk twice, and GV1. For conjunct $areLinked(p, s, paid_by, stt_6)$ we apply rules AL2Ul1, AL2Lk1, and AL1. For conjunct $areLinked(st, s, logs_completed, stt_6)$ the proof proceeds applying rules AL2Ul1 and AL1. Finally, the proof is completed when proving conjunct $\neg areLinked(s, r, captured_on, stt_6)$ applying only rule NAL1.

6.2.2 An Invariant of the POS System

In this section we precisely define the invariant that has been identified in section 6.1.2 and outline the arguments that would allow us to justify that such invariant is preserved by every system operation.
The invariant can be formulated as:

$$
\begin{aligned}
I \equiv\ & (\forall state : State; \forall store, sale : Object) \\
& ((areLinked(store, sale, logs_completed, state) \\
& \Rightarrow (\exists payment : Object \mid areLinked(payment, sale, paid_by, state))
\end{aligned}
$$

Let us start seeing in more detail the most simple among the system operations, *endSale*. Since set preserves all links, it can be proved that if an arbitrary sale $\overline{s}$ is linked to a store $\overline{st}$ in the final state, it must have been linked to $\overline{st}$ in the initial state too. Since by hypothesis every sale that is linked to a store in the initial state is also linked to a payment, there must exist a payment $\overline{p}$ linked to $\overline{s}$. Then using rule AL2St we prove that in the final state $\overline{s}$ is still linked to payment $\overline{p}$. Which means that in fact there is a payment linked to $\overline{s}$ in the final state. The cases of operations *makeNewSale* and *enterItem* are very similar since their programs do not affect either links through associations logs_completed or paid_by. Finally, the case introduced by *makePayment* is a little more complicated. There, we need to discuss whether the sale $\overline{s}$ we consider is the same sale s that occurs in the program. This is because it could be false that s is linked to a store in the initial state, particularly, we know that its link to st does not exist in the initial state. If $\overline{s}$ is the same as s, then the postcondition ensures that s is linked to p, and the thesis is simply proved. If $\overline{s}$ is not the same as s, then it is true that $\overline{s}$ preserves its link to a store from the postcondition to the precondition, and then the same argument as for the other operations holds.

Summary

In this section we showed how to use our framework in a complete and non-trivial case study taken from the bibliography. Correctness assertions were defined for every system operation in a use case of a realistic system. Proofs for such assertions were constructed and explained. In addition, we identified a system invariant which was also defined in our framework, and the preservation of that invariant through every system operation was justified.

Chapter 7

Conclusions and Further Work

In this thesis, a semantics for system state modification primitives and a framework for reasoning about programs based on them were formally specified in Type Theory. The main results and conclusions are summarized in section 7.1. Further work is outlined in section 7.2.

7.1 Summary and Conclusions

In this work, a mathematical specification of state modification primitives of object-oriented systems has been proposed. This specification was formalized in Type Theory using the proof assistant Coq [INR02]. Such a formal semantics provided a precise understanding of the most basic behavioral constructs removing subtle ambiguities, and enabled us to apply formal methods to object-oriented systems development. In that context, a framework for reasoning about simple but powerful programs expressed as sequences of state changes or application of the primitives has been developed. The core of the framework allows to formulate specifications of programs and rigorously prove their correctness. Moreover, a method to prove system invariant preservation across programs using the core constructs was outlined. To deal with partial specifications, an extension was proposed where the exact effect of a program can be specified. We studied under what conditions the resulting state of a primitive can be assured to be well-formed with respect to multiplicities. The results were a number of propositions which have been also proved. These propositions can be incorporated to the framework for additionally reason about state well-formedness. A simple environment for specifying and verifying system behavior has been prototyped using the Coq. Within this system, users are assisted in the task of proof construction, and most importantly proofs are mechanically checked. In addition, automatic proof construction for simple correctness assertions was explored, showing that some degree of automation is feasible to be achieved.

Related formalizations, such as [AL98] or [Mül02], are usually close to programming languages like Java, and therefore do not include concepts not supported by them as for example links and associations. Works from the object community instead cover a wider range of concepts whether they are supported by programming languages or not. However, their approach is commonly informal. The work presented in this thesis specifically addressed modification primitives with a formal foundation, including concepts already formalized, such as objects and attributes, as well as links which are normally treated with less rigor. We believe that this work then contributes to fill the gap that exists between these two areas of study. Furthermore, as it makes use of concepts widely applied to conceptual modeling, our formal approach can be applied even in early phases of the development process.

The semantics of the primitives that has been presented is expressed in terms of a ADT-based specification of object-oriented concepts, which was specifically developed for being used in this work. This specification possesses a high level of abstraction, since functions do not operate on concrete data structures. Our approach does not necessarily have to rely on our specification for being applied, in other words, other specifications could have been used to base the semantics of the primitives as well. However, some reviewed specifications [Ric02, OMG03c] were found unnecessarily detailed for our purposes, which was in fact reasoning instead of testing. Based on our own specification, the resulting semantics of the primitives also possesses a high level of abstraction and have been presented in a simple and compact manner.

In addition, the formal semantics can serve as a specification of the primitives for an eventual implementation of them in a concrete system. This could simply be done by providing an implementation for the basic operations associated to the ADT specifications. Every other operation, including the primitives themselves, could be defined then in terms of these basic operations, as specified in the ADTs. This is another benefit of having used an abstract ADT-based specification.

Our approach of incorporating widely used object-oriented concepts to system states, and manipulating these concepts through specific primitives in a platform independent approach, was found to be in compatibility with recent trends in the object community. The Object Management Group recently incorporated into the UML the UML Action package [OMG03c, p. 2-199], providing new constructs for modeling behavior, also platform independently. In fact, the primitives specified in this work are in close correspondence to some of the actions defined within UML. Model Driven Architecture [OMG01] defines the context in which that behavior is defined. In turn, xUML [KC03] implements a particular action language which is compliant with the UML Action Semantics for a tool-based execution of models in the context of MDA. As tools are involved, a formal approach as ours could be of interest in that domain.

Formal verification techniques are usually applied to simple or reduced systems. However, we have applied the framework to a realistic information system within a known problem already addressed in the object-oriented software engineering bibliography. The case study demonstrated the applicability and power of our approach, which leads to our final conclusion. We believe that the results here reported could be a good starting point to an improved and more powerful framework for specifying and reasoning about object-oriented systems behavior.

7.2 Further Work

An important topic for further work is related to the UML Action Semantics and other UML-related specifications. Particularly, it is our interest to reduce the gap between our approach and that of the UML Actions Semantics. On one hand, a complete alignment of the two semantics could be addressed. Possible mismatches between corresponding concepts should be studied. Elements discussed in section 3.1.1 should be integrated to our specification, as well as other UML concepts not yet included. This could result in new queries in our specification, thus potentially increasing the expressiveness of the assertion language. On the other hand, the framework could be extended with a formalization of new actions. These actions could be other state modification actions, but particularly other control structures such as branching and iteration, which would enable more complex programs to be written. A possible connection between our framework and the Object Constraint Language [OMG03a] could be also investigated. First, OCL could be used in our framework as an assertion language, or for writing expressions within assertions. Additionally, a translation mechanism from our program specifications to OCL expressions could also be of interest.

Finally, our proposal for reasoning about state well-formedness could be improved, including the generation of inference rules from propositions in section 4.4.2. Optimized tactics for automatic, or semiautomatic, proof construction can be developed. One of the main drawbacks in the use of the prototypical implementation of the framework is probably the cost of learning Coq from some of the potential users. In this direction, it could be of interest the development of a front-end for our environment, which would assist users in formulating actual propositions and automating (parts of) proofs.

Appendix A

Other Semantics

The semantics of system state modification primitives introduced in Chapter 3 is based on the specification of types System and State, and other associated notions. That specification possesses a high level of abstraction, which allowed the defined semantics to be simple and compact. However, our specification is not the only existing specification of all those concepts. In fact, [Ric02] and [OMG03c] include alternative specifications which could have been used instead of ours as a foundation for the definition of the semantics of the primitives. In this section, we explore those alternatives showing the specification of the primitive create based on each of them. As those specifications are more concrete than ours, this exploration can reveal the complexities that could have been introduced in our semantics by such specifications. In section A.1 we overview specification in [Ric02] and use it for specifying the semantics of the create primitive. We follow the same approach in section A.2 for the specification included in [OMG03c].

A.1 A Set Theory based Specification

In [Ric02], a set theory based specification of UML concepts for modeling systems and states is included. There, our notions of system and state are called *object model* and *system state* respectively. They are specified as related structures involving a number of accessory sets and functions. We overview the essential parts of this specification before introducing the semantics of create. A model $\mathcal{M}$, is structured as follows:

$$\mathcal{M} = (\textsc{Class}, \textsc{Abstract}, \textsc{Att}_c, \textsc{Assoc}, associates, multiplicities, \prec)$$

where

- $\textsc{Class}$ is a set of names representing all the classes in the object model.
- $\textsc{Abstract}$ is a subset of $\textsc{Class}$ representing abstract classes.

- $\textsc{Att}_c$ is a set of signatures indexed by $\textsc{Class}$. The signatures are of the form $a : t_c \rightarrow t$ and represent attributes of class c, where a is the name of the attribute, t_c is the type induced by class c, and t is the attribute type.
- $\textsc{Assoc}$ is a set of names representing all the associations in the object model.
- *associates* is a function mapping each association name to a list of participating classes.
- *multiplicities* is a function assigning each end of an association a multiplicity specification.
- $\prec$ is a partial order on $\textsc{Class}$ reflecting the generalization hierarchy of classes.

In turn, a system state for a model $\mathcal{M}$ is a structure:

$$\sigma(\mathcal{M}) = (\sigma_{\textsc{Class}}, \sigma_{\textsc{Att}}, \sigma_{\textsc{Assoc}})$$

where

- $\sigma_{\textsc{Class}}(c)$ are finite sets containing all objects identifiers of a class $c \in \textsc{Class}$ existing in the system state. The set $\sigma_{\textsc{Class}}(c)$ is a subset of $\mathrm{oid}(c)$, the set of object identifiers of class c.
- $\sigma_{\textsc{Att}}$ are functions for retrieving attribute values, matching the signatures in $\textsc{Att}^*_c$. This set contains the union of all signatures for class c (i.e. $\textsc{Att}_c$), with the signatures for all its ancestors classes through $\prec$.
- $\sigma_{\textsc{Assoc}}$ are sets containing links between objects. The set $\sigma_{\textsc{Assoc}}(as)$ is a subset of the cartesian product of the interpretation of all classes participating in as. The interpretation of a class is the union of all object identifiers of that class with object identifiers of its ancestors classes through $\prec$.

Finally, the interpretation of an object model $\mathcal{M}$ is defined as the set of all possible states $\sigma(\mathcal{M})$. We refer to [Ric02] for further information about this specification.

Now, we define the primitive create as a relation between states typed as follows, where set Oid is defined as the union of all object identifiers of all classes in $\textsc{Class}$:

```
create:  Oid→Class→Model→State→State→Prop
```

For the specification of create we use the function *parents* mapping a class with the set of its ancestors classes, and define a function $defval$ which maps attributes with their default value.

Let

i. $\mathcal{M} = (\textsc{Class}, \textsc{Abstract}, \textsc{Att}, \textsc{Assoc}, \mathit{associates}, \mathit{multiplicities}, \prec)$
An object model

ii. $c \in \textsc{Class}$, $c \notin \textsc{Abstract}$
A concrete class defined in $\mathcal{M}$

iii. $\sigma(\mathcal{M}) = (\sigma_{\textsc{Class}}, \sigma_{\textsc{Att}}, \sigma_{\textsc{Assoc}})$
A state of model $\mathcal{M}$

iv. $\sigma'(\mathcal{M}) = (\sigma'_{\textsc{Class}}, \sigma'_{\textsc{Att}}, \sigma'_{\textsc{Assoc}})$
A state of model $\mathcal{M}$

v. $o \in (oid(c) - \sigma_{\textsc{Class}}(c))$
A valid object identifier for class c not already in use in state $\sigma(\mathcal{M})$

then $\mathsf{create}(o, c, \mathcal{M}, \sigma(\mathcal{M}), \sigma'(\mathcal{M}))$ iff

i. Object identifier o refers to the created instance and is included, in the resulting state $\sigma'(\mathcal{M})$, in the set of instances corresponding to class c. Since o is also an instance of all parent classes of class c, o is also included in every set of instances corresponding to each parent of class c (if any).

$$\sigma'_{\textsc{Class}}(x) = \begin{cases} \sigma_{\textsc{Class}}(x) & \text{if } x \notin (parents(c) \cup c) \\ \sigma_{\textsc{Class}}(x) \cup \{o\} & \text{if } x \in (parents(c) \cup c) \end{cases}$$

ii. Function $\sigma'_{\textsc{Att}}$ in the resulting state $\sigma'(\mathcal{M})$ returns the default value of each attribute for the created instance o.

$$\sigma'_{\textsc{Att}}(a)(x) = \begin{cases} \sigma_{\textsc{Att}}(a)(x) & \text{if } x \neq o, \forall z \in \textsc{Class} \mid x \in \sigma_{\textsc{Class}}(z), \\ & \quad \forall a : t_z \to t \in \textsc{Att}^*_z \\ defval(a) & \text{if } x = o,\ \forall a : t_c \ \to t \in \textsc{Att}^*_c \end{cases}$$

iii. No links are changed in $\sigma'(\mathcal{M})$.

$$\sigma'_{\textsc{Assoc}} = \sigma_{\textsc{Assoc}}$$

A.2 An Object-Oriented based Specification

The UML metamodel is an object-oriented specification of constructs of the Unified Modeling Language. It is expressed in a language called Meta Object Facility (MOF), which is a subset of UML itself. The UML is intended, among others things, to be used for describing object-oriented systems from different points of view. Particularly, it has specific constructs for modeling the static structure of a system and their runtime configuration, and the metamodel includes a specification for all those constructs.

The UML metamodel is organized in packages each containing the specification of a group of related concepts. In the Core package, elements for modeling system structure are defined. Elements for modeling states are defined in the Common Behavior package. Every model element have a unique name. For reasons of brevity we will focus on concrete metaclasses, collapsing inherited properties, although we refer to abstract metaclasses if needed. From the Core package, the Class metaclass, subclass of Classifier, is a central concept. A class is associated to features, particularly attributes. An instance of metaclass Attribute can have an initial value, and is associated to a type. Generalization and Association are special cases of relationships. A generalization is associated to two classes through two different associations. One class play the role of superclass in that generalization, while the other plays the role of subclass. An association is associated to a sequence of association ends. An instance of AssociationEnd is the connection between the association and the participating classes. Multiplicities and compositions are specified in association ends. Another subclass of classifier is DataType, which we use for typing attributes.

In turn, the structure of states shows a strong correspondence with that of systems. Metaclass Instance is associated to Classifier. By this association, particular forms of instances, objects and data values are connected to classes and data types respectively. Such association represents, for example, the connection between a class and all objects created from it. The mechanisms by which an object is connected to values and links are actually very similar. An object is connected to attribute links, and link ends. An instance of AttributeLink is also connected to an attribute and a data value. That value is the value held by the object for the attribute. An instance of Link is connected to the association from which it was originated, and to the instances of LinkEnd that are the bridge to the objects participating in the link. Those link ends are also connected to their corresponding association ends.

Finally we define two special elements, System and State. System represents the system structure and is associated to the reviewed elements of the Core package. State represents the system state and is similarly associated to elements in Common Behavior package. These two elements are associated, and a system may have many states, but a state is connected to only one system. The system is accessible from the state through the role name structure. We naturally define the primitives as operations of State, which will be the context of the OCL pre- and postconditions used for specifying them. Further information on this specification can be found in the UML Semantics specification [OMG03c]. The next specification of create is based in [Vig03]. For the specification of create we define a function *ancestors* which return all direct or indirect superclasses of a given class.

```
context State::create(o:Name,c:Name)
    pre:  - - Exists a class named "c" in the associated system and
          - - is not an abstract class
          self.structure.ownedClass→exists(e | e.name = c and e.isAbstract = false)

    pre:  - - There is no object with name "o" in the state
          not self.ownedObject→exists(e | e.name = o)

    post: - - Exists a new object of class named "c" that is named "o" and all
          - - its attributes are initialized with their default values
          - - C is the class used to create the new object
          let C = self.structure.ownedClass→select(e | e.name = c)→any(true) in
          - - O is the new object
          let O = self.ownedObject→select(e | e.name = o and e.oclIsNew())→
              any(true) in
              - - C is the classifier for O
              O.classifier→includes(C) and
              - - the new instance is owned by self
              O.owner = self and
              - - self is the owner of the new instance
              self.ownedObject→includes(O) and
              - - for every feature a of class C
              C.ancestors()→collect(feature)→asSet→
                  forAll(a | a.oclIsTypeOf(Attribute) implies
                  - - if feature a is an attribute implies that a is attached
                  - - to an attribute link al that
                  a.oclAsType(Attribute).attributeLink→exists(al |
                      - - is new
                      al.oclIsNew() and
                      - - is attached to the new object O
                      al.instance = O and
                      - - its value is the default value
                      al.value.oclAsType(DataValue).value =
                          a.oclAsType(Attribute).initialValue and
                      - - is owned by the state
                      al.owner = self and
                      self.ownedAttributeLink→includes(al)
                  )
              )
```

Appendix B

Properties on Well-formedness

As discussed in section 4.4.2, it could be possible to reason about state well-formedness with respect to multiplicities. In that context, it is our interest to study under what conditions it is possible to anticipate the well-formedness of a state resulting from a modification. In this chapter, we introduce a set of propositions that provide necessary and sufficient conditions for the well-formedness of a state which is the result of the application of a command to both well-formed states and ill-formed states. These propositions can be then used to generate inference rules for constructing proofs for many steps of modifications. We prove first some basic propositions. The proofs are carried out in logic with a high degree of detail. In turn, the proofs for the main set of propositions are expressed in natural language.

B.1 Basic Properties

Proposition An empty state is well-formed. In symbols:

$$(\forall s : State)(isEmpty(s) \Rightarrow isWellFormed(s))$$

Proof.
Reasoning by contradiction, by AxS2 we have that an empty state has no objects: $isEmpty(s) \Rightarrow (\forall o : Object)(\neg existsObj(o, s))$
By hypothesis we assume that s is ill-formed, then at least one object exist that violates the multiplicities:

$\neg isWellFormed(s) \Rightarrow$
$(\exists o : Object \mid existsObj(o, s))$
$((\exists a : Association;$
$c_1, c_2 : Class;$
$m_1, m_2 : Multiplicity \mid$

$$associates(a, c_1, c_2, structure(s)) \wedge$$
$$multiplicities(a, m_1, m_2, structure(s))$$
$$(isInstanceOf(o, c_1, s) \Rightarrow$$
$$\neg inRange(|\{o' : Object \mid areLinked(o, o', a, s)\}|, m_2) \vee$$
$$isInstanceOf(o, c_2, s) \Rightarrow$$
$$\neg inRange(|\{o' : Object \mid areLinked(o', o, a, s)\}|, m_1))))$$

This is a contradiction, since we showed that s has no objets. This concludes that s is well-formed. □

Proposition (Necessary condition for well-formedness)
It is a necessary condition for a state to be well-formed that for every association a if there exists an instance of one of the classes c participating in a, then there exist at least a number of instances of class c', opposite to c across a, that equals the minimum of the multiplicity at the association end of c'. In symbols:

Given a function `objsInstanceOf : Class×State→Natural` where
$objsInstanceOf(c, s) = |\{o : Object \mid isInstanceOf(o, c, s)\}|$
For every state s we have:

$$isWellFormed(s) \Rightarrow$$
$$(\forall a : Association \mid existsAssociation(a, structure(s));$$
$$\forall c_1, c_2 : Class \mid associates(a, c_1, c_2, structure(s));$$
$$\forall m_1, m_2 : Multiplicity \mid multiplicities(a, m_1, m_2, structure(s)))$$
$$(objsInstanceOf(c_1, s) > 0 \Rightarrow objsInstanceOf(c_2, s) \geq min(m_2) \wedge$$
$$(objsInstanceOf(c_2, s) > 0 \Rightarrow objsInstanceOf(c_1, s) \geq min(m_1))$$

Proof.
Reasoning by contradiction, we assume that s is well-formed and

$$\exists a : Association \mid existsAssociation(a, structure(s)) \wedge$$
$$\exists c_1, c_2 : Class \mid associates(a, c_1, c_2, structure(s)) \wedge$$
$$\exists m_1, m_2 : Multiplicity \mid multiplicities(a, m_1, m_2, structure(s)))$$

where:

$$\neg(objsInstanceOf(c_1, s) > 0 \Rightarrow objsInstanceOf(c_2, s) \geq min(m_2) \wedge$$
$$objsInstanceOf(c_2, s) > 0 \Rightarrow objsInstanceOf(c_1, s) \geq min(m_1))$$

We proceed by proving that exists an object in s that violates the condition of $isWellFormed(s)$.

<u>Case 1</u> : $objsInstanceOf(c_1, s) > 0 \wedge objsInstanceOf(c_2, s) < min(m_2)$

We have $objsInstanceOf(c_1, s) > 0 \Rightarrow \exists o | existsObj(o, s) \wedge isInstanceOf(o, c_1, s)$ then

$$|\{o' : Object \mid existsObj(o', s) \wedge isInstanceOf(o', c_2, s) \wedge areLinked(o, o', a, s)\}| \leq$$
$$|\{o' : Object \mid existsObj(o', s) \wedge isInstanceOf(o', c_2, s)\}| =$$
$$objsInstanceOf(c_2) < min(m_2)$$

This proves that:
$\neg inRange(|\{o' : Object \mid existsObj(o', s) \wedge$
$isInstanceOf(o, c_2, s) \wedge$
$areLinked(o, o', a, s)\}|, m_2)))$
which leads to a contradiction because $isWellFormed(s)$ holds.

<u>Case 2</u> : $objsInstanceOf(c_2, s) > 0 \wedge objsInstanceOf(c_1, s) < min(m_1)$

This case is analogous. □

B.2 Modification of Well-formed States

Proposition When creating an instance in a well-formed state, it is a necessary and sufficient condition for the resulting state to be well-formed that the minimum of all opposite multiplicities through all the possible associations with respect to the class instantiated is zero. In symbols:

$$
\begin{array}{l}
isWellFormed(s) \wedge s' = create(o, c, s) \Rightarrow \\
\quad ((((\forall a : Association \mid existsAssociation(a, structure(s)) \wedge \\
\quad\quad associates(a, c, c', structure(s)); \\
\quad\quad \forall m : Multiplicity \mid multiplicities(a, m', m, structure(s))) \\
\quad\quad\quad (min(m) = 0)) \wedge \\
\quad\quad ((\forall a : Association \mid existsAssociation(a, structure(s)) \wedge \\
\quad\quad associates(a, c', c, structure(s)); \\
\quad\quad \forall m : Multiplicity \mid multiplicities(a, m, m', structure(s))) \\
\quad\quad\quad (min(m) = 0))) \Leftrightarrow isWellFormed(s'))
\end{array}
$$

Proof.
($\Rightarrow$) State s is well-formed, and we know that $s \sim_{obj+o} s'$ and $s \sim_{link} s'$ hold, and by AxStt3 the structures of s and s' are the same. This means that without considering o in s' the rest of objects and links are the same as those of s. Thus, without considering o the state s' is well-formed. For the entire s' to be well-formed it is necessary that the size of every set of objects linked to o through any association is included in the opposite multiplicity respect to o through the association. Since $s \sim_{link} s'$ holds, we know that no object is linked to o in s', then the size of all sets of objects linked to o is zero. We also know (by hypothesis) that the minimum of all multiplicities opposite to o is zero. This shows that o satisfies all multiplicities in s', concluding that s' is well-formed.

($\Leftarrow$) State s is well-formed, and we know that $s \sim_{obj+o} s'$ and $s \sim_{link} s'$ hold, and by AxStt3 the structures of s and s' are the same. Reasoning similarly as above, we show that the size of every set of objects linked to o through any association in s' is zero. Being s' well-formed, every object (in particular o) satisfies the multiplicities. Then zero must be included in every multiplicity opposite to o. Hence, zero is the minimum for all of them. □

Proposition When destroying an instance in a well-formed state, it is a necessary and sufficient condition for the resulting state to be well-formed that every instance of every class associated to the class of the object to be destroyed is linked at least to a number of objects (excluding the one to be destroyed) that equals the minimum of the multiplicity at its opposite end. In symbols:

$$
\begin{aligned}
& isWellFormed(s) \wedge s' = destroy(o,s) \Rightarrow \\
& \quad ((((\forall a : Association \mid associates(a,c,c',structure(s)) \wedge \\
& \quad\quad isInstanceOf(o,c,s); \\
& \quad\quad \forall m : Multiplicity \mid multiplicities(a,m,m',structure(s)); \\
& \quad\quad \forall o' : Object \mid isInstanceOf(o',c',s)) \\
& \quad\quad (|\{o'' : Object \mid (o'' \neq o) \wedge \\
& \quad\quad\quad isInstanceOf(o'',c,s) \wedge \\
& \quad\quad\quad areLinked(o'',o',a,s)\}| \geq min(m)) \wedge \\
& \quad (((\forall a : Association \mid associates(a,c',c,structure(s)) \wedge \\
& \quad\quad isInstanceOf(o,c,s); \\
& \quad\quad \forall m : Multiplicity \mid multiplicities(a,m',m,structure(s)); \\
& \quad\quad \forall o' : Object \mid isInstanceOf(o',c',s)) \\
& \quad\quad (|\{o'' : Object \mid (o'' \neq o) \wedge \\
& \quad\quad\quad isInstanceOf(o'',c,s) \wedge \\
& \quad\quad\quad areLinked(o',o'',a,s)\}| \geq min(m)))))) \\
& \quad \Leftrightarrow isWellFormed(s'))
\end{aligned}
$$

Proof.
($\Rightarrow$) State s is well-formed, and we know that $s \sim_{obj-o} s'$ and $s \sim_{link-o} s'$ hold, and by AxStt3 the structures of s and s' are the same. This means that only links involving o have changed from s to s'. Thus, without considering any association where o is an instance of a class on any of its ends, state s' is well-formed. Focusing on those associations, instance o does not exist in s' and we know that any object that is instance of the same class as o satisfies the multiplicity in s', since they did so in s and their links were unaffected. Finally, we know that objects from opposite classes satisfy the multiplicities in s', since they already did in s even without considering an eventual link to o. We conclude that every instance in s' satisfies the multiplicities.

($\Leftarrow$) State s is well-formed, and we know that $s \sim_{obj-o} s'$ and $s \sim_{link-o} s'$ hold, and by AxStt3 the structures of s and s' are the same. State s' is well-formed, and every instance satisfies the multiplicities without any link to o (which does not exist in s'), in particular instances of a class which is associated to a class that o was instance of in s. Those instances (without considering any link to o) have the same links in s, since $s \sim_{link-o} s'$ holds, yielding that in s they satisfy the multiplicities even without o's help. □

Proposition When creating a link between two instances through an association in a well-formed state, it is a necessary and sufficient condition for the resulting state to be well-formed that the number of objects linked through the association to both objects is strictly less than the maximum of its opposite multiplicity. In symbols:

$$\begin{aligned}&isWellFormed(s) \wedge s' = link(o_1, o_2, a, s) \Rightarrow \\ &\quad (((\forall m_1, m_2 : Multiplicity \mid multiplicities(a, m_1, m_2, structure(s))) \\ &\qquad (|\{o : Object \mid areLinked(o_1, o, a, s)\}| < max(m_2) \wedge \\ &\qquad\ |\{o : Object \mid areLinked(o, o_2, a, s)\}| < max(m_1))) \\ &\quad \Leftrightarrow isWellFormed(s'))\end{aligned}$$

Proof.
($\Rightarrow$) State s is well-formed, and we know that $s \sim_{obj} s'$ and $s \sim_{link+(o_1,o_2,a)} s'$ hold, and by AxStt3 the structures of s and s' are the same. This means that only a link between o_1 and o_2 through a differ from s' to s. Thus, without considering these objects, every object in s' satisfies the multiplicities because they did so in s. By hypothesis, object o_1 preserved in s' all of its links through any other association than a, meaning that it satisfies all their multiplicities in s' too. Object o_1 was not linked in s through a to o_2 and it was linked to a number of objects that is strictly less than the maximum of its opposite multiplicity. In s', o_1 preserves its links and is also linked to o_2 through a. This implies that o_1 is linked in s' to a number of objects that is less or equal to the maximum of its opposite multiplicity, concluding that o_1 satisfies all multiplicities. Analogously, o_2 satisfies all the multiplicities in s'. Now every object in s' satisfies the multiplicities, concluding that s' is well-formed.

($\Leftarrow$) State s is well-formed, and we know that $s \sim_{obj} s'$ and $s \sim_{link+(o_1,o_2,a)} s'$ hold, and by AxStt3 the structures of s and s' are the same. Since s' is well-formed every object satisfies the multiplicities, particularly o_1 and o_2 through a. This means that o_1 in s' is linked through a at most to a number of objects that equals the maximum of its opposite multiplicity (the link to o_2 is included). But since in s the link to o_2 is not present and all the rest is preserved, that number in s is strictly less than the maximum of the opposite multiplicity. The same result is also valid for o_2 in a similar way. □

Proposition When removing a link between two instances through an association in a well-formed state, it is a necessary and sufficient condition for the resulting state to be well-formed that the number of objects linked through the association to both objects is strictly greater than the minimum of its opposite multiplicity. In symbols:

$$\begin{aligned}&isWellFormed(s) \wedge s' = unlink(o_1, o_2, a, s) \Rightarrow \\ &\quad (((\forall m_1, m_2 : Multiplicity \mid multiplicities(a, m_1, m_2, structure(s))) \\ &\qquad (|\{o : Object \mid areLinked(o_1, o, a, s)\}| > min(m_2) \wedge\end{aligned}$$

$$|\{o : Object \mid areLinked(o, o_2, a, s)\}| > min(m_1)))$$
$$\Leftrightarrow isWellFormed(s'))$$

Proof.
($\Rightarrow$) State s is well-formed, and we know that $s \sim_{obj} s'$ and $s \sim_{link-(o_1,o_2,a)} s'$ hold, and by AxStt3 the structures of s and s' are the same. This means that only a link between o_1 and o_2 through a differ from s' to s. Thus, without considering these objects, every object in s' satisfies the multiplicities because they did so in s. By hypothesis, object o_1 preserved in s' all of its links through any other association than a, meaning that it satisfies all their multiplicities in s' too. Object o_1 was linked in s through a to a number of objects (including o_2) that is strictly greater than the minimum of its opposite multiplicity. In s', o_1 preserves its links except that to o_2 through a. This implies that o_1 is linked in s' to a number of objects that is greater or equal to the minimum of its opposite multiplicity, concluding that o_1 satisfies all multiplicities. Analogously, o_2 satisfies all the multiplicities in s'. Now every object in s' satisfies the multiplicities, concluding that s' is well-formed.

($\Leftarrow$) State s is well-formed, and we know that $s \sim_{obj} s'$ and $s \sim_{link-(o_1,o_2,a)} s'$ hold, and by AxStt3 the structures of s and s' are the same. Since s' is well-formed every object satisfies the multiplicities, particularly o_1 and o_2 through a. This means that o_1 in s' is linked through a to at least a number of objects that equals the minimum of its opposite multiplicity (the link to o_2 is not included). But since in s the link to o_2 is present and all the rest is preserved, that number in s is strictly greater than the minimum of the opposite multiplicity. The same result is also valid for o_2 in a similar way. □

Proposition When updating the value of an attribute of an instance in a well-formed state, the resulting state is well-formed. In symbols:

$$isWellFormed(s) \wedge s' = set(o, a, v, s) \Rightarrow$$
$$isWellFormed(s')$$

Proof.
We know that $s \sim_{obj} s'$ and $s \sim_{link} s'$ hold, and by AxStt3 the structures of s and s' are the same. This means that objects and links are preserved in s'. Since they satisfy all multiplicities in s, they also do in s'. This concludes that s' is well-formed. □

B.3 Modification of Ill-formed States

Proposition When creating an instance in an ill-formed state, the resulting state is ill-formed. In symbols:

$$\neg isWellFormed(s) \wedge s' = create(o,c,s) \Rightarrow \\ \quad \neg isWellFormed(s')$$

Proof.
We know that $s \sim_{obj+o} s'$ and $s \sim_{link} s'$ hold, and by AxStt3 the structures of s and s' are the same. State s is ill-formed, so there exists an object o' that does not satisfy its multiplicities at least with respect to an association a. This object exist in s' (by $s \sim_{obj+o} s'$) and its links are preserved (by $s \sim_{link} s'$), in particular those through a. This means that object o' does not satisfy at least its opposite multiplicity through a in s', concluding that s' is ill-formed. □

Proposition When destroying an instance in an ill-formed state, it is a necessary and sufficient condition for the resulting state to be well-formed that (i) every object of the state (except the one to be removed) satisfies the multiplicities constraints (i.e. the object to be removed is the only cause for the ill-formedness of the state) and (ii) every object linked to it through any association are unaffected by losing a link (i.e. after being unlinked from the object to be destroyed they still satisfy the multiplicities). In symbols:

$$\begin{aligned}
&\neg isWellFormed(s) \wedge s' = destroy(o,s) \Rightarrow \\
&\quad ((isWellFormed(s)|_o \wedge \\
&\quad (((\exists a : Association \mid associates(a,c,c',structure(s)) \wedge \\
&\qquad isInstanceOf(o,c,s); \\
&\qquad \forall m : Multiplicity \mid multiplicities(a,m,m',structure(s))) \\
&\qquad\quad (\neg inRange(|\{o' : Object \mid areLinked(o,o',a,s)\}|, m'))) \vee \\
&\quad ((\exists a : Association \mid associates(a,c,c',structure(s)) \wedge \\
&\qquad isInstanceOf(o,c',s); \\
&\qquad \forall m : Multiplicity \mid multiplicities(a,m,m',structure(s))) \\
&\qquad\quad (\neg inRange(|\{o' : Object \mid areLinked(o',o,a,s)\}|, m)))) \wedge \\
&\quad ((\forall a : Association \mid associates(a,c,c',structure(s)) \wedge \\
&\qquad isInstanceOf(o,c,s); \\
&\qquad \forall m : Multiplicity \mid multiplicities(a,m,m',structure(s)); \\
&\qquad \forall o' : Object \mid isInstanceOf(o',c',s)) \\
&\qquad\quad (|\{o'' : Object \mid (o'' \neq o) \wedge areLinked(o'',o',a,s)\}| \geq min(m)) \wedge \\
&\quad ((\forall a : Association \mid associates(a,c,c',structure(s)) \wedge \\
&\qquad isInstanceOf(o,c',s); \\
&\qquad \forall m : Multiplicities \mid multiplicities(a,m',m,structure(s)); \\
&\qquad \forall o' : Object \mid isInstanceOf(o',c,s)) \\
&\qquad\quad (|\{o'' : Object \mid (o'' \neq o) \wedge areLinked(o',o'',a,s)\}| \geq min(m')))) \\
&\quad \Leftrightarrow isWellFormed(s'))
\end{aligned}$$

Proof.
($\Rightarrow$) We know that $s \sim_{obj-o} s'$ and $s \sim_{link-o} s'$ hold, and by AxStt3 the structures of s and s' are the same. State s is ill-formed, but o is the only object that violates its opposite multiplicities. Since object o does not exist in s', in that

state every object satisfies the multiplicities. But in s' every object linked to o in s lost their links to that object, so we need to make sure that they still satisfy the multiplicities in s'. In fact, that actually happens because by hypothesis they already did so in s even without taking into account the links to o. This concludes that s' is well-formed.

($\Leftarrow$) We know that $s \sim_{obj-o} s'$ and $s \sim_{link-o} s'$ hold, and by AxStt3 the structures of s and s' are the same. State s' is well-formed, so every object in s' satisfies the multiplicities. Respect to that set of objects, there is in s another object (object o), which must be responsible for the ill-formedness of state s, and thus it violates at least one multiplicity. If we do not take o into account in s, this state would be well-formed. In addition, there must have been some links in s that do not exit in s'. Those links (if present) involved o and some other objects. For these objects (objects linked to o in s) the multiplicities are satisfied in s' because s' is well-formed, so the number of links to other objects is included in their opposite multiplicities; in particular, their number is greater or equal to the opposite multiplicity. Despite that number is increased in s due to the existence of links to object o, if we do not consider these links to o the number stays unchanged, thus they are linked in s, excluding o, to at least a number of objects equal to the minimum of the opposite multiplicity. □

Proposition When creating a link between two instances through an association in an ill-formed state, it is a necessary and sufficient condition for the resulting state to be well-formed that every object of the state (except for at least one of those involved in the primitive) satisfies the multiplicities, and at least one of them needs one link through the association to satisfy the multiplicities. In symbols:

$$\begin{array}{l}
\neg isWellFormed(s) \wedge s' = link(o_1, o_2, a, s) \Rightarrow \\
\quad ((isWellFormed(s)|_a \wedge \\
\quad ((\forall c_1, c_2 : Class \mid associates(a, c_1, c_2, structure(s)); \\
\quad\; \forall m_1, m_2 : Multiplicity \mid multiplicities(a, m_1, m_2, structure(s))) \\
\quad\quad ((\forall o : Object \mid (o \neq o_1) \wedge isInstanceOf(o, c_1, s)) \\
\quad\quad\quad (inRange(|\{o' : Object \mid areLinked(o, o', a, s)\}|, m_2)) \wedge \\
\quad\quad\; (\forall o : Object \mid (o \neq o_2) \wedge isInstanceOf(o, c_2, s)) \\
\quad\quad\quad (inRange(|\{o' : Object \mid areLinked(o', o, a, s)\}|, m_1)))) \wedge \\
\quad (max(m_2) > |\{o : Object \mid areLinked(o, o_2, a, s)\}| \geq min(m_2) - 1 \vee \\
\quad\; max(m_1) > |\{o : Object \mid areLinked(o_1, o, a, s)\}| \geq min(m_1) - 1)) \\
\quad \Leftrightarrow isWellFormed(s'))
\end{array}$$

Proof.
($\Rightarrow$) We know that $s \sim_{obj} s'$ and $s \sim_{link+(o_1,o_2,a)} s'$ hold, and by AxStt3 the structures of s and s' are the same. State s is ill-formed, but we know that except for objects o_1 and o_2 all other object satisfies the multiplicities in s. Moreover, we know that they are not linked in s through a and at least one of

them is one link below the minimum (causing s to be ill-formed), but both are strictly below the maximum. In s' all objects are preserved, and so happens to the links but adding one between o_1 and o_2 through a. This means that for all objects but for them through a the multiplicities are satisfied in s'. In s', o_1 and o_2 are linked through a, thus the count of objects linked to them through a is increased by one. This proves that for both in s' the count is greater than or equal to the minimum, and is less than or equal to the maximum, concluding that they satisfy the multiplicities in s' too. This shows that s' is well-formed.

($\Leftarrow$) We know that $s \sim_{obj} s'$ and $s \sim_{link+(o_1,o_2,a)} s'$ hold, and by AxStt3 the structures of s and s' are the same. State s' is well-formed, so every object satisfies the multiplicities in s', particularly o_1 and o_2 through a. Since s and s' have the same objects, and the only link changed is through a, we can say that excluding objects that are instances of classes participating in association a, all the rest satisfy the multiplicities. Moreover, objects that are instances of classes associated by a that are not o_1 or o_2 are unaffected in the step from s to s', so if they satisfy the multiplicities in s' they also do in s. Finally, objects o_1 and o_2 satisfy the multiplicities in s' with the link between them included, so preserving in s all their links but the one that connect them, both are strictly below the maximum and either of them are at most one below the minimum. □

Proposition When removing a link between two instances through an association in an ill-formed state, it is a necessary and sufficient condition for the resulting state to be well-formed that every object of the state (except for at least one of those involved in the primitive) satisfies the multiplicities, and at least one of them exceeds by one the maximum of the multiplicity with respect to the association. In symbols:

$$
\begin{array}{l}
\neg isWellFormed(s) \wedge s' = unlink(o_1, o_2, a, s) \Rightarrow \\
\quad ((isWellFormed(s)|_a \wedge \\
\quad ((\forall c_1, c_2 : Class \mid associates(a, c_1, c_2, structure(s)); \\
\quad\;\; \forall m_1, m_2 : Multiplicity \mid multiplicities(a, m_1, m_2, structure(s))) \\
\quad\quad ((\forall o : Object \mid (o \neq o_1) \wedge isInstanceOf(o, c_1, s)) \\
\quad\quad\quad (inRange(|\{o' : Object \mid areLinked(o, o', a, s)\}|, m_2)) \wedge \\
\quad\quad\; (\forall o : Object \mid (o \neq o_2) \wedge isInstanceOf(o, c_2, s)) \\
\quad\quad\quad (inRange(|\{o' : Object \mid areLinked(o', o, a, s)\}|, m_1)))) \wedge \\
\quad (min(m_2) < |\{o : Object \mid areLinked(o, o_2, a, s)\}| \leq max(m_2) + 1 \vee \\
\quad\; min(m_1) < |\{o : Object \mid areLinked(o_1, o, a, s)\}| \leq max(m_1) + 1)) \\
\quad \Leftrightarrow isWellFormed(s'))
\end{array}
$$

Proof.
($\Rightarrow$) We know that $s \sim_{obj} s'$ and $s \sim_{link-(o_1,o_2,a)} s'$ hold, and by AxStt3 the structures of s and s' are the same. State s is ill-formed, but we know that except for objects o_1 and o_2 all other object satisfies the multiplicities in s. Moreover, we know that they are linked in s through a and at least one of them

is one link over the maximum (causing s to be ill-formed), but both are strictly over the minimum. In s' all objects are preserved, and so happens to the links but removing one between o_1 and o_2 through a. This means that for all objects but for them through a the multiplicities are satisfied in s'. In s', o_1 and o_2 are not linked through a, thus the count of objects linked to them through a is decreased by one. This proves that for both in s' the count is less than or equal to the maximum, and is greater than or equal to the minimum, concluding that they satisfy the multiplicities in s' too. This shows that s' is well-formed.

($\Leftarrow$) We know that $s \sim_{obj} s'$ and $s \sim_{link-(o_1,o_2,a)} s'$ hold, and by AxStt3 the structures of s and s' are the same. State s' is well-formed, so every object satisfies the multiplicities in s', particularly o_1 and o_2 through a. Since s and s' have the same objects, and the only link changed is through a, we can say that excluding objects that are instances of classes participating in association a, all the rest satisfy the multiplicities. Moreover, objects that are instances of classes associated by a that are not o_1 or o_2 are unaffected in the step from s to s', so if they satisfy the multiplicities in s' they also do in s. Finally, objects o_1 and o_2 satisfy the multiplicities in s' without the link between them included, so preserving in s all their links including the one that connect them, both are strictly over the minimum and either of them are at most one over the maximum. □

Proposition When updating the value of an attribute of an instance in an ill-formed state, the resulting state is ill-formed. In symbols:

$$\neg isWellFormed(s) \wedge s' = set(o, a, v, s) \Rightarrow \\ \quad \neg isWellFormed(s')$$

Proof.
We know that $s \sim_{obj} s'$ and $s \sim_{link} s'$ hold, and by AxStt3 the structures of s and s' are the same. State s is ill-formed, so there exists an object o' that does not satisfy its multiplicities at least with respect an association as. This object exist in s' (by $s \sim_{obj+} s'$) and its links are preserved (by $s \sim_{link} s'$), in particular those through as. This means that object o' does not satisfy at least its opposite multiplicity through as in s', concluding that s' is ill-formed. □

Bibliography

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[AL98] M. Abadi and K. Rustan M. Leino. A logic of object-oriented programs. Research Report 161, Systems Research Center, Digital Equipment Corporation, Palo Alto, California, 1998.

[ASC01] Action Semantics Consortium. Action Semantics Consortium Site, Internet: `http://www.kc.com/as_site/home.html`, 2001.

[Bar99] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris 7, 1999.

[BCST00] G. Betarte, C. Cornes, N. Szasz, and A. Tasistro. Specification of a Smart Card Operating System. In T. Coquand, P. Dybjer, B. Nordström, and J. M. Smith, editors, *Types for Proofs and Programs, International Workshop TYPES'99, Lökeberg, Sweden, June 12-16, 1999, Selected Papers*, number 1956 in Springer Lecture Notes on Computer Science, pages 77–93. Springer-Verlag, 2000.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 1994.

[Bor03] Borland. Together ControlCenter, Internet: `http://www.borland.com/together`, 2003.

[CD01] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Component Software Series. Addison-Wesley, 2001.

[CDT02] Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 7.3, May 2002*. INRIA, Internet: `http://coq.inria.fr/doc-eng.html`, 2002.

[Coc01] A. Cockburn. *Writing Effective Use Cases*. The Crystal Collection for Software Professionals. Addison-Wesley, 2001.

[CT96] C. Cornes and D. Terrasse. Automating Inversion of Inductive Predicates in Coq. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, number 1158 in Springer Lecture Notes on Computer Science, pages 85–104. Springer-Verlag, 1996.

[DW99] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis^{SM} Approach*. Addison-Wesley, 1999.

[EKW92] D. Embley, B. Kurtz, and S. Woodfield. *Object-Oriented Analysis: A Model-Driven Approach*. Yourdon Press, 1992.

[Gim98] E. Giménez. A Tutorial on Recursive Types in Coq. Technical Report 0221, INRIA, 1998.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[IBM03] IBM. IBM Rational Rose, Internet: `http://www.rational.com`, 2003.

[INR02] INRIA. *The Coq proof assistant, Version 7.3, May 2002*. Institut National de Recherche en Informatique et en Automatique, Internet: `http://coq.inria.fr/`, 2002.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[JCJÖ92] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[JSGB00] B. Joy, G. Steele, J. Gosling, and G. Bracha. *The Java^{TM} Language Specification*. Addison-Wesley, second edition, 2000.

[Kal90] A. Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice Hall International Series in Computer Science. Prentice Hall, 1990.

[KC03] Kennedy Carter. eXecutable UML, Internet: `http://www.xuml.org/MDA/xuml.html`, 2003.

[Kow79] R. Kowalski. *Logic for Problem Solving*. Elseiver North Holland, Inc., 1979.

[Kru00] P. Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, second edition, 2000.

[Lar98] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, first edition, 1998.

[Lar02] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, second edition, 2002.

[LC03] G. Leavens and Y. Cheon. *Design by Contract with JML*. Java Modeling Language Project, Internet: `http://www.jmlspecs.org`, 2003.

[MC03] Microsoft Corporation. Microsoft Visio 2003, Internet: `http://www.microsoft.com/office/visio/default.asp`, 2003.

[Mey91] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

[Mey92] B. Meyer. Design by Contract. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, Prentice Hall Object-Oriented Series, pages 1–50. Prentice Hall, 1992.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[Mon98] J. F. Monin. Proving a Real Time Algorithm for ATM in Coq. In E. Giménez and C. Paulin-Mohring, editors, *Types for Proofs and Programs, International Workshop TYPES'96, Aussois, France, December 15-19, 1996, Selected Papers*, number 1512 in Springer Lecture Notes on Computer Science, pages 277–293. Springer-Verlag, 1998.

[Mül02] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. Number 2262 in Springer Lecture Notes on Computer Science. Springer-Verlag, 2002.

[NN92] H. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, 1992.

[OK99] I. Oliver and S. Kent. Validation of object-oriented models using animation. In *25th EUROMICRO '99 Conference, Informatics: Theory and Practice for the New Millenium, 8-10 September 1999, Milan, Italy*, pages 2237–2242. IEEE Computer Society, 1999.

[Oli99] I. Oliver. 'Executing' the OCL. In A. Rashid, D. Parsons, and A. Telea, editors, *Proceedings of the ECOOP'99 Workshop for PhD Students in OO Systems (PhDOOS'99)*, 1999.

[OMG98] OMG. Request for Proposal: Action Semantics for the UML RfP. OMG Document: ad/98-11-01, Object Management Group, Inc., Framingham, Mass., Internet: `http://www.omg.org`, 1998.

[OMG01] OMG. *Model Driven Architecture Guide, Version 1.0, March 2001*. Object Management Group, Inc., Framingham, Mass., Internet: `http://www.omg.org`, 2001.

[OMG02] OMG. *Meta Object Facility (MOF) Specification, Version 1.4, April 2002*. Object Management Group, Inc., Framingham, Mass., Internet: `http://www.omg.org`, 2002.

[OMG03a] OMG. Object Constraint Language Specification. In *OMG Unified Modeling Language Specification, Version 1.5, March 2003* [OMG03b], chapter 6.

[OMG03b] OMG. *OMG Unified Modeling Language Specification, Version 1.5, March 2003*. Object Management Group, Inc., Framingham, Mass., Internet: `http://www.omg.org`, 2003.

[OMG03c] OMG. UML Semantics. In *OMG Unified Modeling Language Specification, Version 1.5, March 2003* [OMG03b], chapter 2.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE-11)*, number 607 in Springer Lecture Notes on Computer Science, pages 748–752. Springer-Verlag, 1992.

[Pau94] L.C. Paulson. *Isabelle: A generic theorem prover*, volume 828 of *Springer Lecture Notes on Computer Science*. Springer-Verlag, 1994.

[PM97] C. Paulin-Mohring. Le système Coq. Thèse d'habilitation, ENS Lyon, 1997.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Objrct-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs (NJ), 1991.

[Ric01] M. Richters. *The USE tool: a UML-based specification environment*. Internet: `http://www.db.informatik.uni-bremen.de/projects/USE/`, 2001.

[Ric02] M. Richters. *A Precise Approach to Validating UML Models and Constraints*. Number 14 in BISS Monographs. Logos Verlag, Berlin, 2002.

[RJB98] J. Rumbaugh, I. Jaconson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[SoS03] Security of Systems Group. *SoS Project*. Nijmeegs Instituut voor Informatica en Informatiekunde, Internet: `http://www.cs.kun.nl/ita/research/projects/loop/`, 2003.

[vdBJ01] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, number 2031 in Springer Lecture Notes on Computer Science, pages 299–312. Springer-Verlag, 2001.

[Vig03] A. Vignaga. An OCL-based Semantics of System State Modification Primitives. Technical Report 03-18, InCo-Pedeciba, Montevideo, Uruguay, 2003.

[Wer94] B. Werner. *Une théorie des constructions inductives.* PhD thesis, Université Paris 7, 1994.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* Foundations of Computing. The MIT Press, 1993.

[WK98] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1998.

Index

Symbols

$\hookrightarrow$, 24
$<:$, 9, 26
$\sim_{att}$, 36
$\sim_{att-o}$, 36
$\sim_{att\ C}$, 60
$\sim_{link}$, 36
$\sim_{link+(o_1,o_2,a)}$, 36
$\sim_{link-(o_1,o_2,a)}$, 36
$\sim_{link-o}$, 36
$\sim_{link\ C\ D}$, 58
$\sim_{obj}$, 35
$\sim_{obj+o}$, 35
$\sim_{obj-o}$, 35
$\sim_{obj\ C\ D}$, 57
; (sequence), 46
$\ominus$, 58, 59
$\oplus$, 60
$\overline{\in}$, 60

A

abstract data type, *see* ADT
addAssociation operation, 27, 28
addAttribute operation, 27, 29
addClass operation, 27, 28
addGeneralization operation, 27, 28
ADT, 23–24
animation, 2, 44
`Apply`, 80
`Auto`, 80
areLinked operation, 33, 34
assertions, 16
assertion language, 46, 66
associates operation, 27, 28, 31
association, 9, 22
Association type, 24
attribute, 9, 22
Attribute type, 24
automatic tactics, *see* `Auto` and `EAuto`
axiomatic, 16
Axioms (in ADTs), 24

B

behavior, 16
Booch Method, 8

C

Calculus of Inductive Constructions, 5, 19
CBD, 4
class, 9, 22
Class type, 24
command operations, 24
Component-based Development, *see* CBD
contract, *see* software contract
Coq, *see* System Coq
correctness, *see* program correctness
creator operations, 24

D

defVal operation, 27, 28, 30
Design-by-Contract, 17

E

`EApply`, 80, 81
`EAuto`, 80, 81
eXecutable UML, *see* xUML
existsAssociation operation, 27, 30
existsClass operation, 27, 29
existsObj operation, 33
extensor operations, 24

F
final state, *see* state
formalization, 26
framework, 3, 44, 45
Functions (in ADTs), 24

G
generalization, 17, 26
getAttributeType operation, 27, 28, 30
getVal operation, 33, 34
goal, 55

H
hint, 80
`Hints`, 80

I
inference
  rule, 17, 48
  system, 45
  top-down, *see* top-down inference
inheritance, 10
initial state, *see* state
initialization, 13
instantiation, 13
inRange operation, 25, 26
invariant, 11, 64–65
inversion, 75
`Inversion`, 75, 76
isAbstract operation, 27, 28, 29
isAncestor operation, 27, 29, 31
isAttribute operation, 27, 28, 30
isEmpty operation, 33, 37
isInstanceOf operation, 33, 34, 37
isSubClass operation, 27, 28, 29
isWellFormed operation, 33, 38

J
Java Modeling Language, *see* JML
JML, 17

L
Larman, 8, 17
link, 9
linksChanged relation, 58

M
max operation, 25, 26
MDA, 2, 92
metamodel, 2, 10, 22, 23, 40, 97
Meta-Object Facility, *see* MOF
Meyer, 17, 64
min operation, 25, 26
Model Driven Architecture, *see* MDA
MOF, 23, 97
multiplicities operation, 27, 28, 31
multiplicity, 11, 23, 25, 66
Multiplicity type, 25–26

N
newMultiplicity operation, 25
newState operation, 33, 37
newSystem operation, 26, 29

O
object, 8, 9, 22
Object Constraint Language, *see* OCL
object-oriented, 8
Object type, 22
objectsChanged relation, 57
OCL, 2, 17, 40, 44, 93
OMT, 8
OOSE, 8

P
partial
  correctness assertion, 16, 45, 47
  function, 24, 74, 75
  specification, 56
POS System, 83
postcondition, 16, 45
precondition, 16, 45
Preconditions (in ADTs), 24
primitives, *see* system state modification primitives
program, 46
program correctness, 15
proof
  assistant, 19, 73
  checking, 18

construction, 18, 79
script, 77
system, 18, 45

Q
query operations, 24

R
rules, 18

S
slot, 9, 12
software contract, 17, 86
specification, 16
`Split`, 77, 80
state
resulting, 46
initial, 46
State type, 32–39
structure, 9, 10, 22, 26
structure operation, 33, 37
subgoal, 55, 77, 80
System Coq, 5, 19
system
behavior, 1, 2, 83
operation, 4
state, 1, 9
structure, 9
system state modification primitives
create, 12, 33, 34, 38
destroy, 13, 33, 34, 38
informal semantics, 12–14
link, 13, 33, 34, 38
set, 14, 33, 35, 38
unlink, 13, 33, 34, 38
System type, 26–31

T
tactic, 19, 80
termination, 46, 47
top-down inference, 55, 63, 77, 80
Type type, 24

U
UML, 2, 12, 16, 22, 40, 83, 92
action semantics, 2, 12, 34, 92
Unified Modeling Language, *see* UML
Unified Process, *see* UP
UP, 4, 8
use case, 4, 8, 84
USE Tool, 2, 40, 44

V
validation, 19, 40, 44
Value type, 24
valuesChanged relation, 59
verification, 18, 19, 43, 44, 73

W
well-formedness, 10–11, 14–15, 32, 64

X
xUML, 2, 3, 92