Software Specification

MohammadReza Mousavi Michel Reniers

September 1, 2008

Contents

Chapter 1

Introduction

1.1 Models in Software Engineering

Software is a novel type of man-made product and its "engineering" is a relatively young discipline. Making software starts from requirements analysis, i.e., eliciting and specifying the goals of the stakeholders and learning about the domain in which and about which the software is going to be developed. The requirements have to be put down in terms of *models* or *specifications*. These specifications provide a firm basis for both stakeholders and software developers. They will be massaged and transformed into an architectural description of the system and further into the detailed design and finally into the implementation. In other words, all artifacts throughout the process of software development evolve from the abstract model resulting from requirements analysis and are, themselves, *models* at different levels of abstraction.

Software specification is about such *models of software*: *how* to write them, *what* they mean and *when* to use them. It is also about methods for analyzing models: *comparison* of models of the same kind, *checking consistency* among models of different kinds and *reasoning* about properties of models. Before we go into more details about our choice for models and their corresponding techniques, we would like to underscore the relevance of this notion, i.e., model, in software engineering.

In all mature engineering disciplines, before building a complex structure models of such structure are made. Many calculations and experiments are made on the produced models and only when the model is proven to meet the requirements, they are going to be turned into the real constructions. Software is by no means less complex than many other products made in such branches of engineering. From early stages of requirement analysis, we need rigorous models to capture our understanding of the requirements and put them in an unambiguous and analyzable form. After having such a rigorous model, all other steps in the development process should ideally be *sound* model transformations. Each of these transformations, takes a number of models at a certain abstraction level and makes a new model which is more detailed and is closer to the actual implementation, yet it preserves the correctness properties proven for the earlier models. In this sense, the final implementation is nothing but a more detailed model, which is not created from scratch but systematically developed from earlier models.

Models are not only essential for software development but also for its maintenance. Without sufficiently abstract and precise models, one cannot communicate the idea behind many aspects of software clearly and unambiguously.

Also, testing, i.e., validating and verifying, the produced artifact at each stage of software development relies on the abstract model defining the specification of that artifact. Otherwise said, testing the final product as well as any of the intermediate artifacts should be performed against the corresponding specification. At the highest level of abstraction, final acceptance of the software product is determined by its conformance to the abstract specification of requirements.

An excerpt of the IEEE Standard 830-1998 on requirements specification reads as follows:

> An SRS (a software requirement specification) should be ... unambiguous,... consistent, and verifiable. An SRS is unambiguous if, and only if, every requirement stated therein has only

> one interpretation. Requirements are often written in natural language (e.g., English). Natural language is inherently ambiguous. One way to avoid the ambiguity inherent in natural language is to write the SRS in a particular requirements specification language. Its language processors automatically detect many lexical, syntactic, and semantic errors. An SRS is internally consistent if, and only if, no subset of individual requirements stated therein conflict. An SRS is verifiable if, and only if, every requirement stated therein is verifiable. A requirement is verifiable if, and only if, there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement. In general any ambiguous requirement is not verifiable.

Models provide us with notations to specify software and the artifacts towards making it. The notations used in software specification should be easy and unambiguous to read. Members of the software development team (and stakeholders) should be able to reach a common understanding of the models. Moreover, to have any algorithmic tool-support for checks and transformations among different models, having a well-defined meaning for the notation is indispensable. Thus, the notation should not only be easily readable, but also carry a precise meaning. The source of precision in all fields of engineering is mathematics and thus, providing a mathematical meaning, called *formal semantics*, for modeling notations should provide us with the basics for such a precise and well-defined meaning. Unfortunately, many of the common notations for software specification lack such a formal semantics.

1.2 Our Choice of Models

We aimed at notations that provide the right balance between ease of use (with a visual notation, large community of users, existence of standard patterns) and precise meaning (formal semantics, precise notions of equivalence and consistency). There are many aspects of a software system that need to be specified and formalized. These aspects include (but are not limited to): structure and functionality, behavior, physical distribution, timing and performance and security. We believe that there is no single notation that can address all different aspects of software. Even if such a notation existed, it would be a strange and monstrous mix, which would allow for specifications that are neither readable nor analyzable. Thus, we think that it is beneficial to choose and exploit a number of different formalisms, each tailored for a particular aspect of design. Part of information used in one model may originate from or have commonalities with information in another model (in a different formalism); thus, studying the consistency among models in the above-mentioned formalisms is of essential importance.

We decided to focus on two main aspects in software specification, namely, structure and functionality, as one aspect, and behavior. These two aspects are by far the most important ones and are common to many different application types. Besides these two aspects, we would like to present a way to specify high-level correctness criteria and reasoning about specifications. For the two aspects of software specification, we chose a small subset of the Unified Modeling Language, comprising three diagram types: Class Diagrams (representing structure and functionality), Sequence- and State Diagrams (representing the behavioral aspect). Fortunately, Sequence- and State-Diagrams have a long tradition of formal semantics in form of Message Sequence Charts and Statecharts. For Class Diagrams, no stable formal semantics has been defined to date. To remedy this, we decided to provide Algebraic Specifications and Z Formal Notation as formalizations of the structure and functionality aspect of software. Moreover, we fill the gap of a high-level logical specification language in UML by introducing the modal μ-calculus and using it as a specification language for the correctness criteria of the system.

Chapter 2

Use Case Diagrams

2.1 Introduction

The *use case diagram* provides us with the first model to classify and represent the user requirements. Hence, it is of essential importance in driving the rest of the development process. Being the first model has some implications on its level of formality, namely, a use case diagram is very close to the natural language specification of the requirements. It needs to be developed, formalized and detailed in the subsequent stages by means of other models/diagrams.

2.2 Use Cases

A *use case* is a a collection of (inter)actions which delivers a result of some value to a user. The fact that use cases must be of some independent value to the user is of utmost importance; in other words, each use case should realize a *goal* of a particular type of users. They should represent the underlying reason for developing a software system, namely, to provide some useful service to its users. They should be a means to the user's goals or requirements. In [2], use case is defined as follows:

> A use case captures a *contract* between the stakeholder of a system [and its developers] about its *behavior*. The use case describes the system's behavior under various conditions as the system responds to a request from one of its stakeholders, called the *primary actor*.

Use cases should be about *what* the system should do, from the perspective of a user, and not *how* it should behave. In other words, a use case should be evident as a desired behavior without knowing anything about the implementation details. Use cases should thus capture all system-level functions that the users may envision. Use cases are given a name in order to make them traceable in the subsequent models and documents. Use cases are graphically represented by a solid ellipse with their name written in the middle.

Suppose that we would like to specify a telephone system, which comprises a number of telephone sets, a number of telephone lines and a switch center. The telephone system is used to make and receive telephone calls. Each telephone set is connected to a telephone line with a unique telephone number. (There may be more telephone sets connected to one line.) In each telephone set, there is an answering machine functionality, which receives telephone calls, records and playbacks voice messages. The answering machine may be turned on or off. Moreover, there is a caller-id system, which, if enabled, shows the telephone number of the calling party when a call is received. The switch center takes care of receiving dial requests, and sending appropriate tones and signals to the lines (and eventually to the telephone sets). Figure 2.1 depicts some use cases of this system.

In order to detect and specify use cases, it is essential to explore the scope of the system. Use cases are about what users (external actors) require from the system and thus, define the responsibilities of the system under specification.

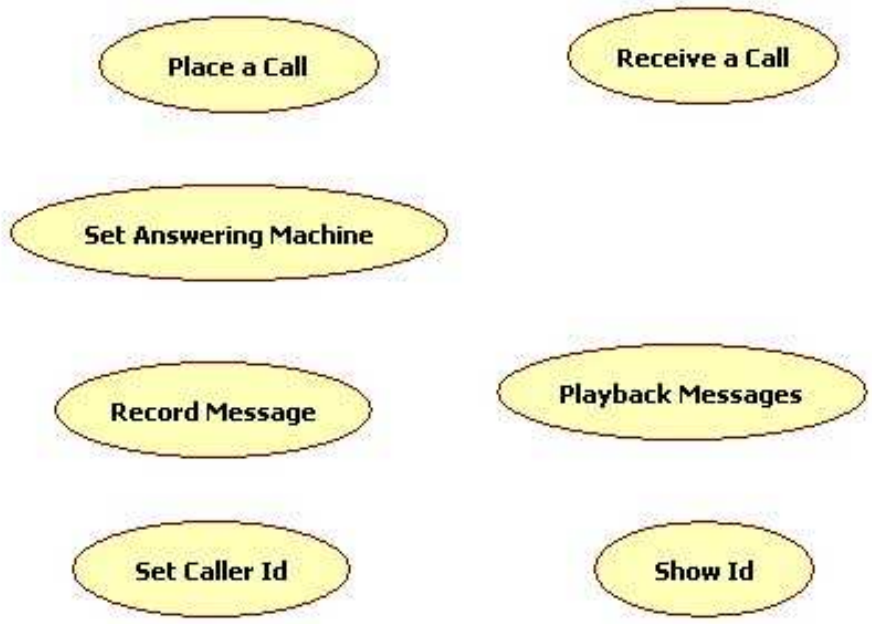


Figure 2.1: Some Use Cases of the Telephone System

At the first stage, use cases are accompanied with a specification in a natural language (e.g., English), which describes the following items:

1. pre-condition: when a use case is available to its user,

2. trigger: what interactions initiate the use case,

3. guarantee (post-condition): what the use case achieves through this use case,

4. scenarios: a collection of (inter)actions realizing the use case. Note that this collection need not be a plain sequence of actions; rather in most cases, it comprises a number of different sequences (or even partially-ordered set of actions) in which alternative or optional sub-sequences may appear. Thus, a typical way of classifying such use case descriptions is to specify the typical scenario (sequence of interactions) followed by all alternative (exceptional) scenarios. Exceptional and optional scenario's are branches of the main scenario. At each step of the scenario, you may ask yourself: Are there any other alternatives? Is there anything that can go wrong at this stage? These will provide you with the exceptional and optional branches. For each exceptional or scenario the branching / choice point in the main scenario should be given explicitly.

5. references to other use cases: use cases may use or use, include or specialize other use cases (cf. Section 2.4 for more information).

Note that use cases are specified at the system level, that is they capture interactions that are initiated by and/or deliver some results to the actors outside the system. Thus, there pre-conditions, triggers, and guarantees are all specified at the system level and do not contain details, e.g., from the internal state of a particular implementation.

Figure 2.2 gives the case specification for the use case "Place a Call" in Figure 2.1. Note that "Record Message" is a sub-scenarios that is referenced here and must be specified elsewhere. Sub-scenario (e-g) is an internal reference to the scenario specified in this use-case.

Place a call

Pre-condition The telephone set is connected to the telephone line “A”, it is on-hook, the line is free and there is no incoming call (it is not ringing).

Trigger The user picks up the telephone hook connected of the telephone set (connected to line “A”), which is free and dials number “B”.

Guarantee A communication between “A” and “B” will commence.

Main Scenario
(a) The user picks up the telephone hook connected to the telephone line “A”.
(b) If the line is free, the user receives a dial tone sent by the line .
(c) The user dials number “B”.
(d) The call request is forwarded to the switch center.
(e) If line “B” is not busy, the call request is forwarded to “B” and a tone is sent to “A”.
(f) “B”’s telephone rings.
(g) If somebody at “B” picks up the hook, the ringing tone at “A” is stopped and a telephone connection will commence.

Alternatives

(b).1.1 If the telephone line is engaged in a conversation, the user will be connected to the same conversation.

(b).2.1 If the user does not dial a number for a certain amount of time, a permanent tone is emitted by the switch center, no further call will be accepted until and the user has to replace the hook.

(e).1.1 If line “B” is busy, and “B” does not have call waiting the user at a will receive a busy tone.

(e).2.1 If line “B” is busy, and “B” has call waiting the user at “A” will receive a call-waiting tone from the switch center. When line “B” becomes free, sub-scenario (e-g) follows.

(g).1.1 If nobody picks the call at “B”, after a certain amount of time and a telephone set at “B” has its answering machine enabled, then the *Record Message* scenario at “B” will follow.

(g).2.1 If nobody picks the call at “B”, after a certain amount of time and no telephone set at “B” has its answering machine enabled, then the user at “A” will receive a no response tone from the switch center (via the line).

Figure 2.2: A Textual Description of the “Place a Call” Use Case

2.3 Actors

The value of a use case is relevant for a particular class of users. Such a class of users is represented by an *actor*.s An actor represents the role of a class of users interacting with the system. An actor may characterize a number of human users, or even a computer system interacting with the system under development. One user or system may play different roles in interacting with the system and thus, be represented by more than one actor. For example, a bank manager should be able to both create new accounts, as an instance of the "manager" actor and if she has an account in the same bank, she should be able to deposit and withdraw money, as an instance of the "customer" actor.

2.4 Relationships

A *use case diagram* describes the relationship between the actors and use cases and among different use cases. The only possible relationship between actors, on one hand, and use cases, on the other hand, is *association*. An association indicates that an actor interacts with the system for the specified use cases. Associations between actors and use cases are not directed and do not have names. They are represented by solid lines between actors and use cases (see Figure 2.3). (In general, associations may assume directions or have names, but in this note, we only use undirected and unnamed associations between actors and use cases.) Note that some use cases may use actors in order to realize their goal. In this case, the actor does not require the use case but the use case requires the actor. In this manuscript, we do model such an actor in the use case diagram and put an association between the actor and the use case. This issue is not settled among UML users, however, and many users omit such an actor and its relationship to the use case altogether.

Actors may be related by generalization relations. The source of a generalization is a special case of (is generalized by) the target. This means that the target can always be replaced by the source but not vice versa. Generalization is represented by a directed arrow with a solid line and an empty triangle as arrow head (see the relation between User and Owner in Figure 2.4). In this example, in all interactions between the system and the system, a user may be replaced by an owner, but not vice versa, i.e., there may be interactions, in which an owner may participate but an ordinary user may not.

Use cases may also be related by the generalization relation, which stands for the same intuition and is denoted by the same notation (in fact both actors and use cases are specific types of classes and these relations are generically defined on classes).

Sometimes there are several use cases with a common sub-sequence of interactions. In order to avoid repeating the description of such a flow, one can make a use case for the common sub-sequence and relate all the other use cases to it, using a dependency arrow with stereotype $\ll include \gg$.

2.5 Diagrams

A use case diagram puts all the ingredients described so far together. Figure 2.3 depicts the use case diagram for the telephone system described in Section 2.2.

As another example, consider a file system on which users can create new files, execute, display (on different output devices) and delete existing files. There is a special type of delete, which removes the file permanently from the file system. The file system makes use of an access right system which specifies who the owner of each file is and what operations are allowed by which users. The owner of each file may change the access rights to the file and give or take other people's permissions to access the file. In addition to the person who creates the file, the administrator is considered the owner of all files. A use case diagram for this system is given in Figure 2.4.

Having specified the system responsibilities in terms of use cases, we proceed in the coming chapters with formalizing and structuring this specification using other specification methods.

Exercise 2.5.1 Consider the UML-Bookshop specified below.

The bookshop has a number of books from different titles. Each book may appear in two versions: hard-cover or soft-cover and thus may have two different prices. A user shops for a book by searching for

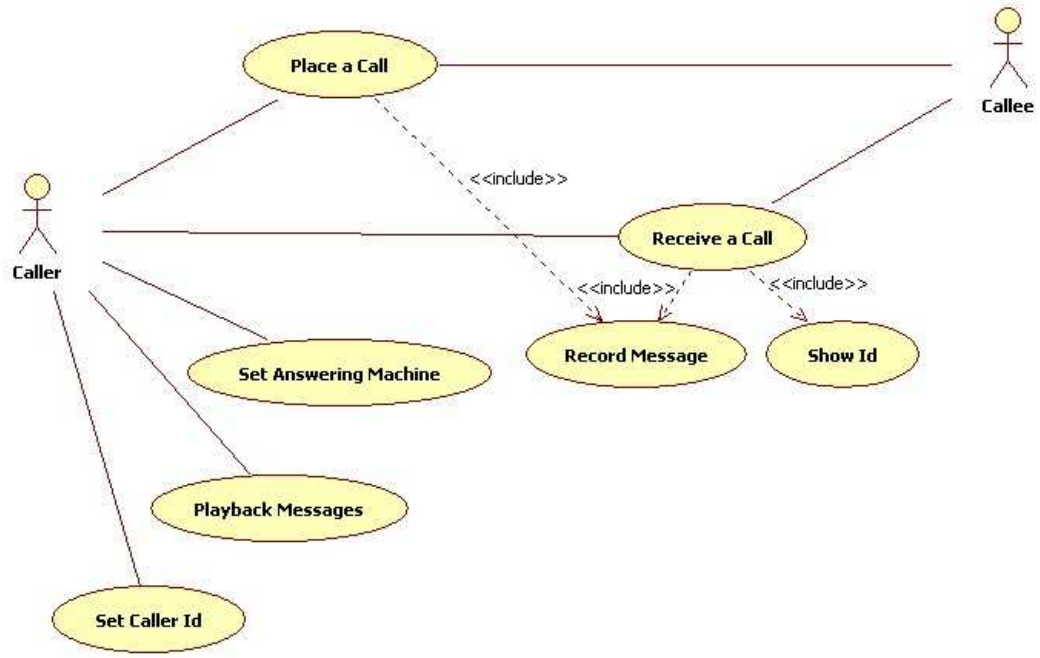


Figure 2.3: A Use Case Diagram for the Telephone System

the book title and receiving the prices of available versions of the title. Afterwards, the user either pays the price and buys the book using a credit card or cancels the purchase. Credit card payment concerns a credit card number, name of the owner, expiration date and the amount to be withdrawn. A credit card payment should be authorized by the bank. The bookshop owner can add books (of possibly new titles) to its stock.

1. Draw a use case diagram for the system.
2. Give a detailed description of each use case.

Exercise 2.5.2 Consider a transport company specified below.

The company owns a number of vehicles of different sizes which can transport goods. A client submits a request for transportation by specifying the size of the package to be transported, its source and destination. The distance between source and target determines the amount of time during which the vehicle will be en route. The company then sends an offer to the client by finding the first possible period during which a vehicle of an appropriate size is available. If the client agrees with the terms of the offer, it provides an account number and the authorization to withdraw the amount of offer from the account. Upon a successful transaction with the bank (given the account information provided by the user), the amount of money will be transfer to the company's account and the company will schedule the transport as specified in the offer.

1. Draw a use case diagram for the system.
2. Give a detailed description of each use case.

Exercise 2.5.3 Consider the embedded system of a railway controller. The railway controller controls three tracks, two signals and a crossing as depicted in Figure 2.5.

A signal shows either green or red. Each track is equipped with a sensor that shows the presence of a train and the direction of its movement. The supervisor of the control system may decide to change the status of the signal (from green to red or vice versa), or change the status of the crossing (from close to open or vice versa).

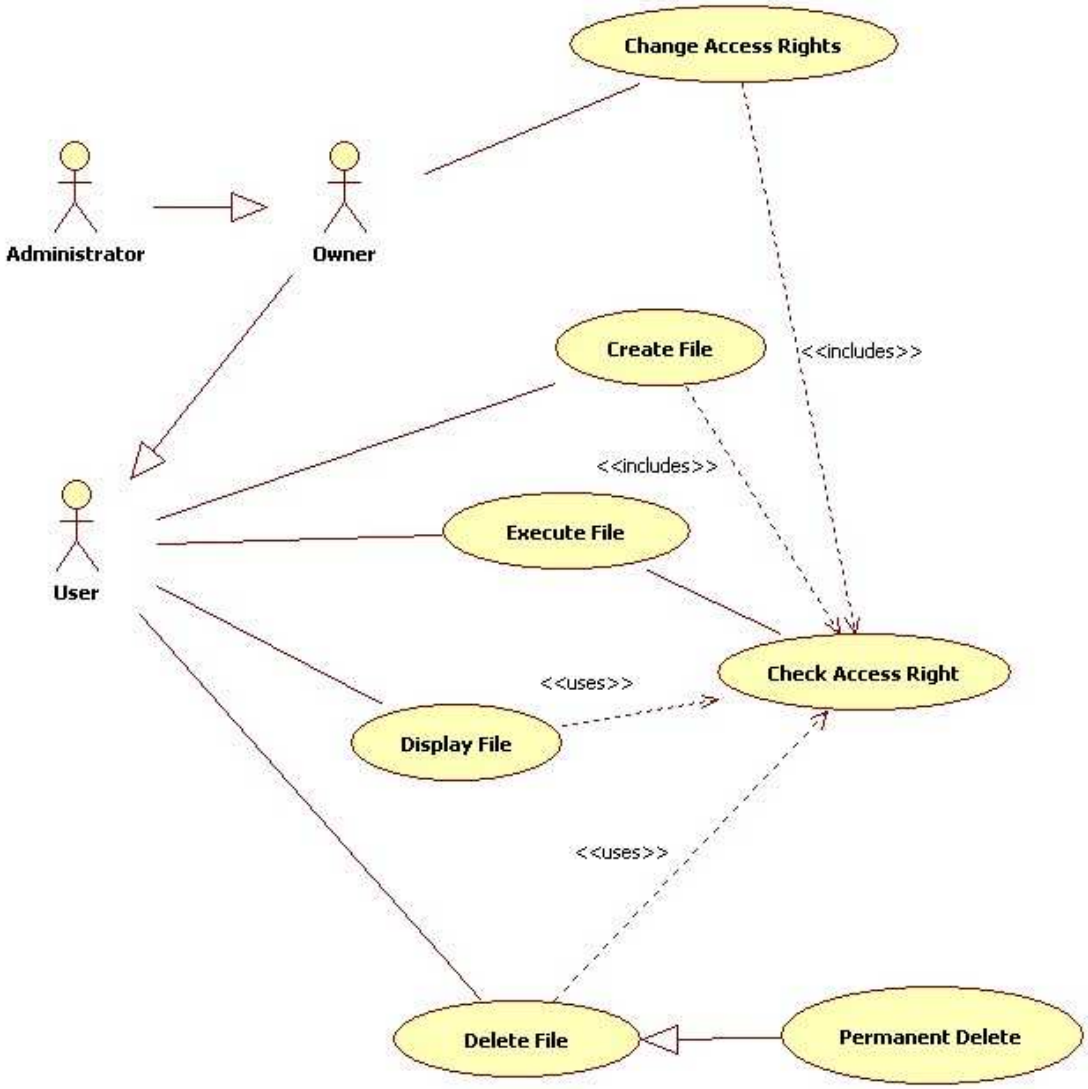


Figure 2.4: A Use Case Diagram for the File System

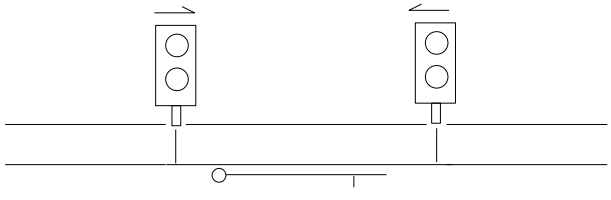


Figure 2.5: A Schematic View of the Railway System

1. If the supervisor decides to change a signal to green, the controller should make sure that
   (a) there is no train coming in the opposite direction towards the signal,
   (b) the crossing is closed and
   (c) the signal in the opposite direction is red.

   If the above checks are successful, then the signal will be changed or otherwise, the supervisor will be notified of the impossibility.
2. If the supervisor decides to change a signal to red, the controller should make sure that there is no train on the track before the signal. If this is the case, then the signal will be changed or otherwise, the supervisor will be notified of the impossibility.
3. If the supervisor decides to change a crossing to open, the controller should make sure that
   (a) there is no train on the track, where the crossing resides and
   (b) the signals before and after the crossing are red.
4. Closing a crossing is always possible.

1. Draw a use case diagram for the system.
2. Detail each use case using by its possible scenarios.

Bibliography

[1] G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language User Guide, Chapters 16-17. Addison-Wesley. 2nd Edition, 2005.

[2] A. Cockburn, Writing Effective Use Cases. Addison-Wesley, 2001.

[3] M. Fowler. UML Distilled: Brief Guide to the Standard Object Modeling Language, Chapter 9. Addison-Wesley, 3rd Edition, 2004.

Chapter 3

Class Diagrams

3.1 Classes

3.1.1 Basic Notions

Classes are units of encapsulation which decompose the responsibilities (and the state) of the system. Each class represents an abstract notion of state and behavior and can be instantiated into objects. Apart from the state and behavior specified by the class definition, each object has a unique identity.

The definition of a class comprises three main parts:

1. name
2. attributes: parts of state with a given name and a type (domain of its values); attributes may be given an initial value from their domain. Syntax of an attribute is given below:

   Multiplicity is a range of the form $a..b$, where $a \in \mathbb{N}$ (is a natural number) and $b \in \mathbb{N} \cup \{*\}$. The symbol $*$ represents any arbitrary natural number greater than or equal to a. Multiplicity $a..a$ is usually denoted by a and $0..*$ (any number) is denoted by $*$.
3. operations: specifying the behavior of the class with a given name and a list of parameters. Each parameter has a name and a type. Operations may also have a return value of a given type.

3.1.2 Visibility

Information hiding and encapsulation are key concepts in managing the complexity of software specification. They imply that the details of the functionality and the state should be hidden from the outside world, and interfaces should provide a stable access to the functionality that a class provides. This is partially achieved by defining visibility ranges for attributes or operations of a class.

An operation or attribute is called *private* when it cannot be accessed by any method beyond the class boundaries, e.g., by operations of other classes or global statements of the program. Private visibility is denoted by a minus sign before the name of the operation or attribute.

Public visibility means that the an operation or attribute can be accesses publicly. This is denoted by a plus sign before the definition of an operation or attribute. It is against information hiding to define public attributes. Instead, to access attributes, one may define public *set* (respectively, *get*) methods, which check the validity of values before assigning them to private attributes (respectively, report the value of private attributes without leaking any reference to them). We omit the set and get methods from class diagrams (and assume their existence), when they do not contain any particular functionality apart from the usual checks and copies.

An attribute or operation is called *protected*, when it is visible only to the members of the class and those of the classes inheriting from it (i.e., specializing it, see Section 3.3 for the definition of generalization / inheritance).

3.2 Relationships

There are three main types of relationships among classes: dependency, association and generalization.

3.2.1 Dependency

Dependency means that a class needs / uses another class for realizing some of its behavior. Dependency is *not structural* and is a rather weak relation. In case of a dependency, the source class does not include (a reference to) the target class but may use it, for example, as a type of a parameter of one its operations. It is denoted by a directed dashed-line from the source (using) class to the target (used) class. Dependency is also called *uses* relation. In UML dependency is used very broadly and can easily overlap with association, defined next, but in this note, we restrict the use of dependency to the relations that are *not* structural.

Consider the file system described in Section 2.5. We can distinguish "File" as a class which has several different responsibilities, to be translated specified in terms of operations. One particular operation of this class is "display". This operation should display the content of the file on different output devices, e.g., a display monitor or a printer). The relation between the class "File" and the class "Output Device" is a dependency relation. This is depicted schematically in Figure 3.1.

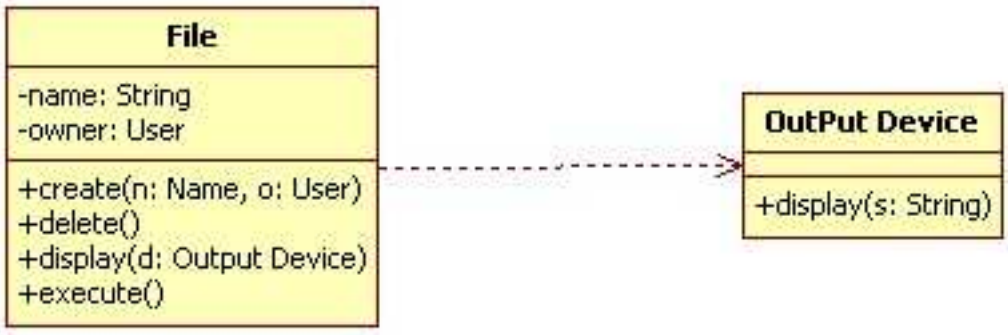


Figure 3.1: A Dependency Relation Between File and Output Device

3.2.2 Association

Association means that objects of one class has (a reference to) objects of another class. Thus, an associations can be seen as an attribute of the source class, of which the type is the target class. Due to their common nature, we use the word *feature* to refer to both attributes and associations. Associations are intrinsically directed, showing which class can *navigate*, i.e., has a reference, to the other class. Associations with the same source and target class are also allowed and represent references to the objects of the same type (e.g., in a linked-list). Associations are represented by solid directed lines. If there is an association in both directions, then the two-way association is represented by a solid line either with arrow-heads at both ends or without any arrow-head at all. The name of the reference is put above the line at the target side; this name is also called the *role*. To denote the number of objects participating in each association, one can put a multiplicity below the line at each side.

For some associations extra information need to be stored, e.g., for a person who is also a member of a library, a member-identifier needs to be stored. Such information and its corresponding operations are gathered in an *association class* and the class is attached to the corresponding association relation.

There are two special kinds of association:

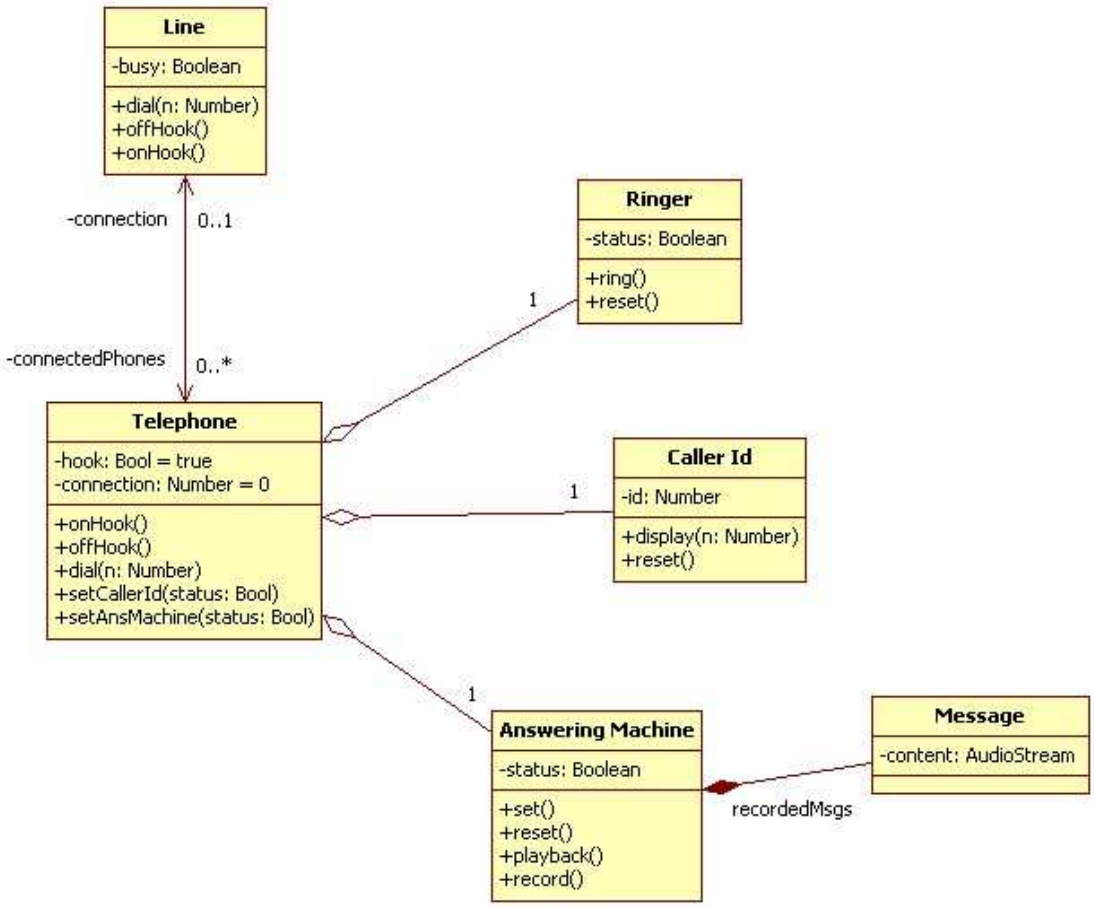


Figure 3.2: Association Relations in the Telephone System

1. aggregation: aggregation represents the whole/part relationship; in aggregation relation the part (the target of the association) may continue to exist even if the whole is destroyed. It is represented just like an association with an empty diamond at the source (whole) side.

2. composition: composition is a stronger type of aggregation in which the whole creates the parts and upon destruction, destroys them. Composition is represented just like an aggregation in which the diamond at the source (whole) side is filled.

Consider the telephone system described in Section 2.2. The telephone set has a number of parts: ringer, answering machine and caller ID display. The answering machine, in turn, may contain a number of recorded messages. The relationship among the telephone set and its parts is an aggregation relation, since the parts are made and can be destroyed independently of the telephone set. The relation between the answering machine and its recorded messages is a composition relation since the messages are created by the answering machine and cannot exist without it. The telephone set is connected to the line, by which it can dial other lines. The line can also signal the telephone set in order to notify an incoming call. The relation between the line and telephone set is thus a two-way association. An excerpt of the class diagram of this system is depicted in Figure 3.2.

3.3 Generalization

Generalization is used when the source of the relation inherits all the state and behavioral specification of the target and specializes (some of) them. The source of the generalization, called sub-class, can replace the target, called super-class, in all its relationships, but not vice-versa. Generalization is also called *inheritance*

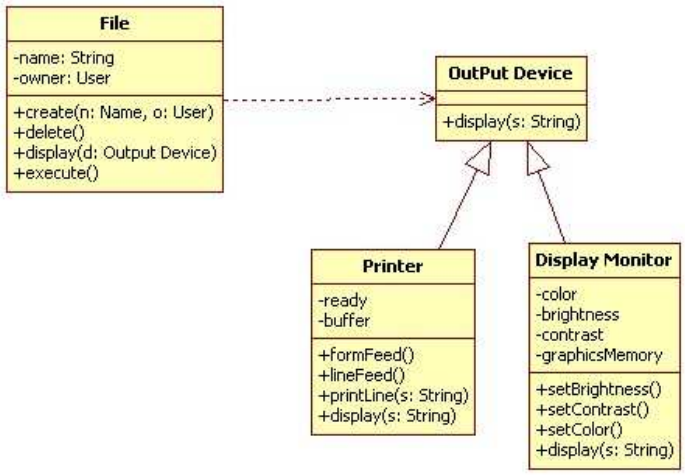


Figure 3.3: Generalization Relations in the File System

or *is-a-kind-of* relation. Generalization is denoted by a directed line, of which the arrow-head is a triangle. Cyclic, hence self-loop, generalization relations is not allowed.

A class may be generalized by more than one class (be the source to two generalizations). This phenomenon is called *multiple inheritance*. It is a rather involved concepts and one can also represent a similar design by using two generalizations and one aggregation, thus including the two specialized classes in one class.

Consider the file system example; in this example, there are a number of output devices on which the display command should be able to write. These include a display monitor and a printer. This information is captured by the diagram depicted in Figure 3.3.

Exercise 3.3.1 Consider the specification of the UML-Bookshop given in Exercise 2.5.1. Identify classes, attributes and operations in each use-case and draw the class diagram for this system.

Exercise 3.3.2 Consider the specification of the transport company given in Exercise 2.5.2. Identify classes, attributes and operations in each use-case and draw the class diagram for this system.

Exercise 3.3.3 Consider the embedded system of the railway controller specified in Exercise 2.5.3. Identify classes, attributes and operations in each use-case and draw the class diagram for this system.

Exercise 3.3.4 Consider a building comprising a number of floors in which an elevator system (with a number of elevators) is working. The elevators move among different floors in the building. There are two types of buttons in the system: buttons inside the elevator and a button outside at each floor. A button may be turned on by pushing it or turned off, when the door of the elevator on the corresponding floor is opened. The door of the elevator are closed before it moves from one floor to the other.

1. Identify classes, their attributes and operations in the above system.

2. Draw the class diagram for this system.

Chapter 4

Algebraic Specification

4.1 Introduction

In the previous chapter, class diagrams have been introduced for describing the decomposition of the system on a high-level of abstraction. A clear disadvantage of the class diagrams is that they lack details about the structure and meaning of the state components and the operations on these. Therefore, it is impossible to verify whether the state of the system guarantees certain properties (invariants). In this chapter we will introduce *algebraic specification* for a combined description of state and state changes. Using algebraic specifications it is possible to reason about state and state changes, and thus more confidence in the specification can be obtained.

The purpose of an algebraic specification is to

1. represent mathematical structures and functions over those
2. while abstracting from implementation details such as the size of representations (in memory) and the efficiency of obtaining the outcome of computations
3. as such formalizing computations on data
4. allowing for automation due to a limited set of rules

An algebraic specification achieves these goals by means of defining a number of sorts (data types) together with a collection of functions on them. These functions can usually be divided into two classes:

1. *constructor function symbols*: these are introduced to create elements of the sort or to construct complex elements from simpler ones.
2. *additional function symbols*: these are functions defined in terms of the constructor functions.

If one considers an algebraic specification of the Booleans the constructors can be **true** and **false**. In that case all other connectives, such as $\wedge$ and $\vee$, may be considered to be additional functions. Alternatively, also the combination of **false** and $\neg$ can be considered constructors. In that case **true** may be considered an additional function.

In the context of the description of state and state change one may think of the sort as the set of possible states (not necessarily all of them can occur in practice) and one may think of the functions as being useful for describing the state changes that may occur. In the example of the bookshop that has been used in one of the exercises in the previous chapter, the state of the bookshop consists of a set of books. Each book consists of a title, a type (hardcover or softcover), a price, and an amount of copies that are on stock. A typical operation is the update of the stock with a number of book (copies). Another operation is describing the effect of selling a specific book copy to a customer.

Constructor functions and the additional functions are represented by means of a so-called signature. This is a collection of function names together with their input and output sorts. The previously mentioned functions can be given by the following table.

Constructor function symbols	Additional function symbols
true $:\rightarrow \mathbb{B}$	$\neg : \mathbb{B} \rightarrow \mathbb{B}$
false $:\rightarrow \mathbb{B}$	$\wedge: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
	$\vee: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

The functions on a sort are defined by means of equations defining their properties. For example the definition of the binary function $\wedge$ on the Booleans is achieved by means of the following equations:

$$\textbf{true} \wedge b = b \qquad \textbf{false} \wedge b = \textbf{false}$$

Example 4.1.1 (Natural Numbers) The natural numbers can be represented by an algebraic specification with an infinite number of constructor function symbols as follows:

$$\begin{array}{lcl} 0 & : & \rightarrow \mathbb{N} \\ 1 & : & \rightarrow \mathbb{N} \\ 2 & : & \rightarrow \mathbb{N} \\ \vdots & & \end{array}$$

There are several reasons why this is not convenient. One wants to finitely capture the signature for representation in tools. Also for defining additional function symbols, it is not very convenient to define an infinite number of cases. Consider, for example, the additional function symbol $+$ representing addition of natural numbers. Given the above constructor function symbols, addition can be defined by means of the following equations

$$\begin{array}{llll} 0+0=0 & 1+0=1 & 2+0=2 & \cdots \\ 0+1=1 & 1+1=2 & 2+1=3 & \cdots \\ 0+2=2 & 1+2=3 & 2+2=4 & \cdots \\ \vdots & \vdots & \vdots & \end{array}$$

The question thus is how to *finitely* capture the natural numbers with addition?

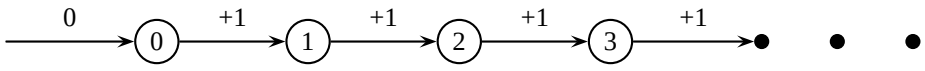


Figure 4.1: The natural numbers.

As the picture in Figure 4.1 illustrates, one can obtain all natural numbers from the following set of constructor function symbols:

$$\begin{array}{lcl} 0 & : & \rightarrow \mathbb{N} \\ S & : & \mathbb{N} \rightarrow \mathbb{N} \end{array}$$

The unary function symbol S, for successor function, represents addition of 1 to a natural number Here the natural number i is represented by the i-times repeated application of function symbol S to symbol 0:

$$S(S(...S(0)))$$

Now it is not hard to define additional well-known functions on natural numbers

$$\begin{array}{lcl} + & : & \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \\ \times & : & \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \end{array}$$

using equations

$$\begin{array}{lcllcl} 0+n & = & n & 0 \cdot n & = & 0 \\ S(m)+n & = & S(m+n) & S(m) \cdot n & = & n + (m \cdot n) \end{array}$$

Observe that in the above equations m and n are variables that range over the sort $\mathbb{N}$. Also note that there is an equation for each constructor function symbol in one of the arguments of the additional function symbols. Such definitions are often called inductive definitions.

In the examples of the Booleans and the natural numbers all sorts involved for the domain and range types of the functions have been of the same sort. In general this is not required. One may define an operation with different domain and range sorts. For example an equality function eq on natural numbers requires two natural numbers as arguments and defines a Boolean. Other examples are an additional function symbol $<\colon \mathbb{N} \times \mathbb{N} \to \mathbb{B}$ representing the relation 'less than' on natural numbers and the predicate $odd : \mathbb{N} \to \mathbb{B}$ indicating whether a natural number is odd.

Equations for these additional function symbols are the following:

$$\begin{array}{llllllll} eq(0,0) & = & \textbf{true} & 0 < 0 & = & \textbf{false} & odd(0) & = & \textbf{false} \\ eq(0,S(m)) & = & \textbf{false} & 0 < S(m) & = & \textbf{true} & odd(S(m)) & = & \neg odd(m) \\ eq(S(m),S(n)) & = & eq(m,n) & S(m) < S(n) & = & m < n & & & \end{array}$$

Exercise 4.1.1 Give another finite set of constructor function symbols for representing the natural numbers. Is it the case that each natural number has a unique representation in terms of these constructor function symbols? Can you think of an additional function symbol where the equations are simpler due to the chosen constructor function symbols.

4.2 Formal definition of algebraic specifications

Definition 4.2.1 (Signature) A signature is a triple (S, Σ_c, Σ_a) where

- S is a collection of sort names
- Σ_c is a set of constructor function symbols with their associated arities
- Σ_a is a set of additional function symbols with their associated arities

Example 4.2.2 The arities of some of the function symbols already discussed are as follows

- **true** has arity $\to \mathbb{B}$ (constant or nullary function symbol)
- S has arity $\mathbb{N} \to \mathbb{N}$ (unary function symbol, prefix notation)
- $+$ has arity $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ (binary function symbol, infix notation)
- $<$ has arity $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$ (binary function symbol, infix notation)

Example 4.2.3 Formally, the previously mentioned sort of the Booleans has a signature consisting of the set of sort names $S = \{\mathbb{B}\}$ and the set of function symbols $\Sigma = \{$**true** $:\to \mathbb{B}$, **false** $:\to \mathbb{B}, \neg : \mathbb{B} \to \mathbb{B}, \wedge\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}, \vee\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}\}$. In this example there is only one sort defined.

Example 4.2.4 A formal statement of the signature of Booleans and Natural numbers is achieved as follows:

- sort names: $S = \{\mathbb{B}, \mathbb{N}\}$
- constructor function symbols (with arity):

$$\Sigma_c = \left\{ \begin{array}{ll} \textbf{true} :\to \mathbb{B}, & 0 :\to \mathbb{N}, \\ \textbf{false} :\to \mathbb{B}, & S : \mathbb{N} \to \mathbb{N} \end{array} \right\}$$

- additional function symbols (with arity):

$$\Sigma_a = \left\{ \begin{array}{l} \neg : \mathbb{B} \to \mathbb{B}, \\ \wedge\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}, \\ \vee\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}, \\ \\ + : \mathbb{N} \times \mathbb{N} \to \mathbb{N}, \\ \times : \mathbb{N} \times \mathbb{N} \to \mathbb{N}, \\ \\ \leq\colon \mathbb{N} \times \mathbb{N} \to \mathbb{B} \end{array} \right\}$$

Using a signature, and a set of variables, terms van be constructed. We assume that for every sort name $s \in S$ a set of variables V_s is given. Usually the type of a variable is clear from its use.

Definition 4.2.5 (Terms) The sets of terms of sorts $s \in S$ over signature (S, Σ_c, Σ_a), denoted $\mathcal{T}_s$ are defined inductively as follows:

1. $x \in \mathcal{T}_s$ for any $x \in V_s$
2. $f(t_1, \cdots, t_n) \in \mathcal{T}_s$ for any $f : s_1 \times \cdots \times s_n \rightarrow s \in \Sigma_c \cup \Sigma_a$ and $t_i \in \mathcal{T}_{s_i}$ (for $i = 1, \cdots, n$)

A term in which no variables occur is called a closed term.

Note that constructor function symbols and additional function symbols are treated the same way in the above definition!

Example 4.2.6 For $x \in V_{\mathbb{N}}$ and $b \in V_{\mathbb{B}}$

- $S(x)$ is a term of sort $\mathbb{N}$
- $b \wedge (S(x) \leq 0)$ is a term of sort $\mathbb{B}$
- $b \leq S(0)$ is not a term

It should be noted that due to the nature of the definition of terms, terms are always finite.

In an algebraic specification, sometimes, there are many ways of writing the same (whatever that may mean). For example, $S(S(0)) + S(0)$ and $S(S(S(0)))$ both denote the natural number 3, and $S(x) \leq 0$ and $\neg$**true** $\wedge\ b$ both denote the boolean value **false**.

Definition 4.2.7 (Equations) An equation over signature (S, Σ_c, Σ_a) is a pair (t, t') of terms of the same sort. Usually we write $t = t'$ for an equation (t, t').

Example 4.2.8 The following are equations over the signature of the Booleans:

true $\wedge\ b = b$	$b \wedge \neg b =$ **false**
false $\wedge\ b =$ **false**	$b \Rightarrow \neg b =$ **false**

Definition 4.2.9 (Algebraic Specification) An algebraic specification (Sig, E) consists of a signature *Sig* and a set of equations E over this signature.

Example 4.2.10 (Equations for $\mathbb{B}$)

$\neg$**true** $=$ **false**	**true** $\wedge\ b = b$	**true** $\vee\ b =$ **true**
$\neg$**false** $=$ **true**	**false** $\wedge\ b =$ **false**	**false** $\vee\ b = b$

Example 4.2.11 (Equations for $\mathbb{N}$)

$0 + x = x$	$0 \times x = 0$
$S(x) + y = S(x + y)$	$S(x) \times y = y + (x \times y)$

Some questions that arise when writing down equations are “How do we know that we have included only correct equations?” and “How do we know we included all relevant equations?”.

Besides the equations of an algebraic specification there are also some more general reasoning rules that allow to derive interesting equalities between terms.

Definition 4.2.12 (Substitution) A substitution σ is a mapping from variables to terms. The application of a substitution σ to a term t, denoted by $\sigma(t)$, is defined inductively as follows

1. $\sigma(x) = \sigma(x)$ for any $x \in V_s$

2. $\sigma(f(t_1,\cdots,t_n)) = f(\sigma(t_1),\cdots,\sigma(t_n))$ for any $f : s_1 \times \cdots \times s_n \to s \in \Sigma_c \cup \Sigma_a$ and $t_i \in \mathcal{T}_{s_i}$ (for $i = 1,\cdots,n$)

In the first clause the first occurrence of σ is the mapping from terms to terms and the second occurrence is the mapping from variables to terms.

Example 4.2.13 For σ given by $\sigma(x) = S(S(S(y)))$, $\sigma(y) = S(0)$, and $\sigma(b) = \neg b'$:

- $\sigma(x \leq S(y)) = \sigma(x) \leq \sigma(S(y)) = S(S(S(y))) \leq S(\sigma(y)) = S(S(S(y))) \leq S(S(0))$
- $\sigma(b \wedge \textbf{false}) = \sigma(b) \wedge \textbf{false} = \neg b' \wedge \textbf{false}$

Definition 4.2.14 (Derivability) Let $T = (Sig, E)$ be an algebraic specification. Two terms t and t' are derivably equal, notation $T \vdash t = t'$, is the smallest relation that satisfies

(ax) $T \vdash t = t'$ for any t and t' such that $t = t' \in E$

(sub) $T \vdash \sigma(t) = \sigma(t')$ for any substitution σ and any t and t' such that $T \vdash t = t'$

(ref) $T \vdash t = t$ for any t

(sym) $T \vdash t' = t$ for any t and t' such that $T \vdash t = t'$

(trans) $T \vdash t = t''$ for any t and t'' such that $T \vdash t = t'$ and $T \vdash t' = t''$ for some t'

(c) $T \vdash f(t_1,\cdots,t'_n) = f(t'_1,\cdots,t_n)$ for any n-ary f and t_i and t'_i such that $T \vdash t_i = t'_i$

Example 4.2.15 Prove $N \vdash S(0) + (0 + S(S(0))) = S(S(S(0)))$ w.r.t. the algebraic specification of natural numbers shown before. The proof is visualized in the following proof tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{0+x=x}\,(\text{ax})}{0+S(S(0))=S(S(0))}\,(\text{sub})}{S(0)+(0+S(S(0)))=S(0)+S(S(0))}\,(c) \quad \dfrac{}{S(x)+y=S(x+y)}\,(\text{ax})}{S(0)+(0+S(S(0)))=S(0+S(S(0)))}\,(\text{sub}) \quad \dfrac{\dfrac{}{0+x=x}\,(\text{ax})}{0+S(S(0))=S(S(0))}\,(\text{sub})}{S(0)+(0+S(S(0)))=S(0+S(S(0)))}\,(\text{trans}) \quad \dfrac{}{S(0+S(S(0)))=S(S(S(0)))}\,(c)$$
$$\dfrac{}{S(0)+(0+S(S(0)))=S(S(S(0)))}\,(\text{trans})$$

Usually, we prefer the following much shorter linear representation:

$$\begin{array}{rcl} S(0)+(0+S(S(0))) & = & S(0)+S(S(0)) \\ & = & S(0+S(S(0))) \\ & = & S(S(S(0))) \end{array}$$

With the algebraic specification for the Booleans shown before, can we prove $B \vdash \neg\neg b = b$? Similarly, with the algebraic specification of the natural numbers shown before, can we prove $N \vdash x + 0 = x$?

Exercise 4.2.1 (Booleans) Consider the following algebraic specification B of the Booleans

Constructor function symbols	**Additional function symbols**	**Equations**
true $:\to \mathbb{B}$	$\neg : \mathbb{B} \to \mathbb{B}$	(B1) $\neg$**true** $=$ **false**
false $:\to \mathbb{B}$	$\wedge$: $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$	(B2) $\neg$**false** $=$ **true**
	$\vee$: $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$	(B3) **true** $\wedge\, b = b$
		(B4) **false** $\wedge\, b =$ **false**
		(B5) **true** $\vee\, b =$ **true**
		(B6) **false** $\vee\, b = b$

Extend the given algebraic specification of the Booleans with the well-known additional function symbols $\oplus$ (exclusive or), $\Rightarrow$ (implication) and $\Leftrightarrow$ (bi-implication).

Exercise 4.2.2 Consider the following algebraic specification N of the natural numbers

Constructor function symbols	Additional function symbols	Equations
$0 :\rightarrow \mathbb{N}$	$+ : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$	(N1) $0 + x = x$
$S : \mathbb{N} \rightarrow \mathbb{N}$	$\times : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$	(N2) $S(x) + y = S(x + y)$
		(N3) $0 \times x = 0$
		(N4) $S(x) \times y = y + (x \times y)$

1. Add both a type declaration and a set of defining equations for the well-known factorial function: $m!$.
2. Add both a type declaration and a set of defining equations for the well-known function of exponentiation: m^n.
3. Define $\leq$, $<$, $>$, and $\geq$ on natural numbers. Of course you may assume that the algebraic specification of the Booleans is given.

Exercise 4.2.3 Give an algebraic specification of a sort $\mathbb{C}$ for representing Cartesian coordinates. You may assume that only natural numbers are to be used for representing coordinates. Also define functions for obtaining the first and second component of a coordinate. Finally, define an equality function on coordinates.

Exercise 4.2.4 (Queens) Give an abstract data type for representing the Eight Queens problem. It should allow the specification of a predicate $valid$ that indicates whether a placement of queens on a chess board is safe (i.e., no single queen is attacking another queen).

Exercise 4.2.5 (Arrays) Assume the existence of a sort $\mathbb{E}$. An array is indexed by natural numbers and has a lower and an upper bound. By means of the constructor function $create$ an array can be created: $create(l, u)$ creates an array indexed by natural numbers from l upto and including u.

$$create : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{A}$$

It is assumed that the values of the array are undefined upon creation. By means of the constructor function $assign$ a value of sort $\mathbb{E}$ can be assigned to the array at a certain position.

$$assign : \mathbb{A} \times \mathbb{N} \times \mathbb{E} \rightarrow \mathbb{A}$$

1. Define an additional function $eval$ for obtaining the value of the array at a certain index position.
2. Define a function eq on arrays that identifies terms representing the same (mathematical) array. Stated differently, two arrays are considered equal when they have the same value for all explicitly assigned indices.

4.3 If-then-else

Many times, as we will demonstrate later in this chapter, it is convenient to base the outcome of a certain operation on a condition. To facilitate description of such cases, for any sort $\mathbb{S}$ we introduce an additional function symbol if _ then _ else _ : $\mathbb{B} \times \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{S}$ with equations

if **true** then s else $s' = s$
if **false** then s else $s' = s'$

To facilitate reasoning in general and with if-then-else in particular we introduce a proof rule for reasoning with case distinctions.

Definition 4.3.1 (Derivability)

- $T \vdash t = t'$ for any t and t' such that $T \cup \{b = \textbf{true}\} \vdash t = t'$ and $T \cup \{b = \textbf{false}\} \vdash t = t'$

Using case distinction, one can prove the following properties of if-then-else:

if b then s else $s = s$
if b then s else $s' =$ if $\neg b$ then s' else s

Exercise 4.3.1 Prove

if b then s else $s = s$
if b then s else $s' =$ if $\neg b$ then s' else s
if b then (if c then s else t) else $t =$ if $b \wedge c$ then s else t

Exercise 4.3.2 Consider the extended algebraic specification B of the Booleans from Exercise 4.2.1. Formulate some equations that you expect to hold and prove these. An example of such an equation is $b \wedge (b' \vee b'') = (b \wedge b') \vee (b \wedge b'')$.

Exercise 4.3.3 (Cursor) Give an algebraic specification for placing a cursor on a screen using the sort of coordinates $\mathbb{C}$ from Exercise 4.2.3. Associate with the cursor an iconic representation (of some undetailed sort $\mathbb{BM}$). Include functions for moving the cursor up and down and left and right. Add a function for changing the icon associated with the cursor. Also define a function for obtaining the position of the cursor.

4.4 Constructor induction

We ended the previous section with questions relating to the suitability of the given algebraic specifications for the Booleans and the natural numbers for deriving expected properties. In this section we will finally start using the distinction between constructor function symbols and additional function symbols. We will use the assumption that every closed term is equal to a term that consists of constructors only. As a consequence we can formulate an additional proof rule, called *constructor induction*.

In the following definition, and elsewhere, we use the notation $t[x := t']$ for substitution of term t' for variable x in term t.

Definition 4.4.1 (Derivability) If, for each constructor constant c of the sort of variable x

$$T \vdash t[x := c] = t'[x := c]$$

and

$$T \vdash t[x := f(x_1, \cdots, x_n)] = t'[x := f(x_1, \cdots, x_n)]$$

follows from

$$T \vdash t[x := x_i] = t'[x := x_i]$$

for all $i = 1, \cdots, n$ with x_i of the same sort as x for each n-ary constructor function f of the sort of variable x then

$$T \vdash t = t'$$

Example 4.4.2 (Induction on $\mathbb{B}$) The constructor induction principle from the previous definition instantiated for sort $\mathbb{B}$ becomes: if

$$B \vdash t[x := \textbf{true}] = t'[x := \textbf{true}]$$

and

$$B \vdash t[x := \textbf{false}] = t'[x := \textbf{false}]$$

then

$$B \vdash t = t'$$

As an example we prove $B \vdash \neg\neg b = b$ using this constructor induction.

1. $B \vdash (\neg\neg b)[b := \textbf{true}]$
   $= \neg\neg\textbf{true}$
   $= \neg\textbf{false}$
   $= \textbf{true}$
   $= b[b := \textbf{true}]$

2. $B \vdash (\neg\neg b)[b := \textbf{false}]$
   $= \neg\neg\textbf{false}$
   $= \neg\textbf{true}$
   $= \textbf{false}$
   $= b[b := \textbf{false}]$

Therefore, by induction, $B \vdash \neg\neg b = b$

Example 4.4.3 (Induction on $\mathbb{N}$) The constructor induction principle from the previous definition instantiated for sort $\mathbb{N}$ becomes: if

$$N \vdash t[x := 0] = t'[x := 0]$$

and for all n,

$$N \vdash t[x := S(n)] = t'[x := S(n)]$$

follows from

$$N \vdash t[x := n] = t'[x := n]$$

then

$$N \vdash t = t'$$

As an example we prove $N \vdash x + 0 = x$ using this constructor induction.

1. $N \vdash (x + 0)[x := 0]$
   $= 0 + 0$
   $= 0$
   $= x[x := 0]$

2. Assume $N \vdash (x + 0)[x := n] = x[x := n]$ or $N \vdash n + 0 = n$
   Then $N \vdash (x + 0)[x := S(n)]$
   $= S(n) + 0$
   $= S(n + 0)$
   $= S(n)$
   $= x[x := S(n)]$

Therefore, by induction, $N \vdash x + 0 = x$

As another example, we prove $N \vdash x + S(y) = S(x + y)$:

1. $N \vdash (x + S(y))[x := 0]$
   $= 0 + S(y)$
   $= S(y)$
   $= S(0 + y)$
   $= (S(x + y))[x := 0]$

2. Assume $N \vdash (x + S(y))[x := n] = S(x + y)[x := n]$ or $N \vdash n + S(y) = S(n + y)$
   Then $N \vdash (x + S(y))[x := S(n)]$
   $= S(n) + S(y)$
   $= S(n + S(y))$
   $= S(S(n + y))$
   $= S(S(n) + y)$
   $= (S(x + y))[x := S(n)]$

Therefore, by induction, $N \vdash x + S(y) = S(x + y)$

Finally, we prove $N \vdash x + y = y + x$:

1. $$\begin{array}{lcl} N & \vdash & (x+y)[x := 0] \\ & = & 0 + y \\ & = & y \\ & = & y + 0 \\ & = & (y+x)[x := 0] \end{array}$$

2. Assume $N \vdash (x + y)[x := n] = (y + x)[x := n]$ or $N \vdash n + y = y + n$

   Then $$\begin{array}{lcl} N & \vdash & (x+y)[x := S(n)] \\ & = & S(n) + y \\ & = & S(n+y) \\ & = & S(y+n) \\ & = & y + S(n) \\ & = & (y+x)[x := S(n)] \end{array}$$

   Therefore, by induction, $N \vdash x + y = y + x$

In most examples in these notes, sorts are used in which the constructor function symbols have at most one argument of the range sort. In Exercises 4.5.3 and 4.5.6, algebraic specifications are used for which a constructor function symbol is needed that has two arguments of the sort of the range sort.

Exercise 4.4.1 (Natural numbers) Consider the algebraic specification N of the natural numbers as given in Exercise 4.2.2. Prove that $N \vdash (m + n) \cdot o = m \cdot o + n \cdot o$ for any natural numbers m, n, and o.

Exercise 4.4.2 (Modulo) Give an algebraic specification for representing natural numbers modulo a given positive natural number n. It should contain functions for addition, multiplication, and exponentiation.

4.5 Example Algebraic specifications

In this section some examples of mathematical structures that occur frequently in specifying software are presented.

4.5.1 Multisets

Constructor function symbols

$$\begin{array}{lcll} \varnothing & : & & \to \ \mathbb{M} \\ add & : & \mathbb{E} \times \mathbb{M} & \to \ \mathbb{M} \end{array}$$

where $\mathbb{E}$ is the sort from which the elements of the multiset are drawn. Note that multisets defined this way are not represented uniquely since $add(e, add(f, m))$ and $add(f, add(e, m))$ represent the same multisets.

We introduce equations to achieve that such multiset representations are derivably equal:

$$add(e, add(f, m)) = add(f, add(e, m))$$

Additional function symbols

- $\cup : \mathbb{M} \times \mathbb{M} \to \mathbb{M}$

  $$\varnothing \cup m = m$$

  $$add(e, m) \cup m' = add(e, (m \cup m'))$$

- $\in: \mathbb{E} \times \mathbb{M} \to \mathbb{B}$

  $$e \in \varnothing = \textbf{false}$$

  $$e \in add(e', m) = eq(e, e') \vee (e \in m)$$

  where eq represents equality on sort $\mathbb{E}$

- $\# : \mathbb{M} \to \mathbb{N}$

  $$\#(\varnothing) = 0$$

  $$\#(add(e, m)) = S(\#(m))$$

We prove $M \vdash m \cup (m' \cup m'') = (m \cup m') \cup m''$ using constructor induction:

1. $$\begin{array}{lll} M & \vdash & (m \cup (m' \cup m''))[m := \varnothing] \\ & = & \varnothing \cup (m' \cup m'') \\ & = & m' \cup m'' \\ & = & (\varnothing \cup m') \cup m'' \\ & = & ((m \cup m') \cup m'')[m := \varnothing] \end{array}$$

2. Assume $M \vdash n \cup (m' \cup m'') = (n \cup m') \cup m''$

   $$\begin{array}{lll} \text{Then } M & \vdash & (m \cup (m' \cup m''))[m := add(e, n)] \\ & = & add(e, n) \cup (m' \cup m'') \\ & = & add(e, n \cup (m' \cup m'')) \\ & = & add(e, (n \cup m') \cup m'') \\ & = & add(e, (n \cup m')) \cup m'' \\ & = & (add(e, n) \cup m') \cup m'' \\ & = & ((m \cup m') \cup m'')[m := add(e, n)] \end{array}$$

Therefore, by induction, $M \vdash m \cup (m' \cup m'') = (m \cup m') \cup m''$.

4.5.2 Stacks

Assume a sort $\mathbb{E}$.

Constructor function symbols	Additional function symbols	Equations
$[] :\to \mathbb{S}$	$pop : \mathbb{S} \to \mathbb{S}$	$pop(push(e, q)) = q$
$push : \mathbb{E} \times \mathbb{S} \to \mathbb{S}$	$top : \mathbb{S} \to \mathbb{E}$	$top(push(e, q)) = e$

One should observe that the additional function symbols pop and top are not defined for application on the empty stack $[]$. We assume that undefinedness means that value is not known, i.e., there is a value for $pop([])$, but we cannot rely on knowing this value.

4.5.3 Mathematical sets

Assume a sort $\mathbb{E}$.

Constructor function symbols

$$\begin{array}{lcll} \varnothing & : & & \to \mathbb{S} \\ add & : & \mathbb{E} \times \mathbb{S} & \to \mathbb{S} \end{array}$$

Equations

$add(e, add(e', s)) = add(e', add(e, s))$
$add(e, add(e, s)) = add(e, s)$

Additional function symbols

If you want to define an additional function symbol representing the removal of a specific element from a set, at a certain point it becomes necessary to compare this element to be removed with elements in the set and to have different equations based on this comparison.

$$\begin{array}{lcll} \setminus & : & \mathbb{S} \times \mathbb{S} & \rightarrow & \mathbb{S} \\ del & : & \mathbb{E} \times \mathbb{S} & \rightarrow & \mathbb{S} \end{array}$$

$s \setminus \varnothing = s$
$s \setminus add(e, s') = del(e, s) \setminus s'$

$del(e, \varnothing) = \varnothing$
$del(e, add(e', s)) =$ if $eq(e, e')$
then $del(e, s)$
else $add(e', del(e, s))$

Example 4.5.1 As an example, we give the proof of $e \in del(e, s) =$ **false** where $\in$: $\mathbb{E} \times \mathbb{S} \rightarrow \mathbb{B}$ is define by means of the equations

$e \in \varnothing =$ **false** $\qquad e \in add(e', s) = eq(e, e') \vee e \in s$

.

By constructor induction on s:

- $$\begin{array}{lcl} S & \vdash & (e \in del(e, s))[s := \varnothing] \\ & = & e \in del(e, \varnothing) \\ & = & e \in \varnothing \\ & = & \textbf{false} \\ & = & \textbf{false}[s := \varnothing] \end{array}$$

- Assume $S \vdash (e \in del(e, s))[s := s'] =$ **false**$[s := s']$

  – Use equation $eq(e, e') =$ **true**

  $$\begin{array}{lcl} S \cup \{eq(e, e') = \textbf{true}\} & \vdash & (e \in del(e, s))[s := add(e', s')] \\ & = & e \in del(e, add(e', s')) \\ & = & e \in \text{ if } eq(e, e') \\ & & \quad \text{then } del(e, s') \\ & & \quad \text{else } add(e', del(e, s')) \\ & = & e \in \text{ if } \textbf{true} \\ & & \quad \text{then } del(e, s') \\ & & \quad \text{else } add(e', del(e, s')) \\ & = & e \in del(e, s') \\ & = & \textbf{false}[s := add(e', s')] \end{array}$$

  – Use equation $eq(e, e') =$ **false**

$$
\begin{array}{lll}
S \cup \{eq(e,e') = \textbf{false}\} & \vdash & (e \in del(e,s))[s := add(e',s')] \\
& = & e \in del(e, add(e',s')) \\
& = & e \in \begin{array}{l}\text{if } eq(e,e') \\ \quad \text{then } del(e,s') \\ \quad \text{else } add(e', del(e,s'))\end{array} \\
& = & e \in \begin{array}{l}\text{if } \textbf{false} \\ \quad \text{then } del(e,s') \\ \quad \text{else } add(e', del(e,s'))\end{array} \\
& = & e \in add(e', del(e,s')) \\
& = & eq(e,e') \vee (e \in del(e,s')) \\
& = & \textbf{false} \vee (e \in del(e,s')) \\
& = & \textbf{false} \vee \textbf{false} \\
& = & \textbf{false} \\
& = & \textbf{false}[s := add(e',s')]
\end{array}
$$

Exercise 4.5.1 (Stacks) Consider the following algebraic specification S for representing stacks with elements from a given (yet unidentified) sort $\mathbb{E}$ with a function symbol eq representing equality.

Constructor function symbols	**Additional function symbols**	**Equations**
$[] :\rightarrow \mathbb{S}$	$pop : \mathbb{S} \rightarrow \mathbb{S}$	(S1) $pop(push(e,q)) = q$
$push : \mathbb{E} \times \mathbb{S} \rightarrow \mathbb{S}$	$top : \mathbb{S} \rightarrow \mathbb{E}$	(S2) $top(push(e,q)) = e$

1. Extend this algebraic specification with a function $\mathrm{Empty?}$ for determining whether or not a stack is empty.
2. Define an equality function on stacks.
3. Formulate the induction principle for proving properties about stacks.
4. Prove that equality is reflexive, i.e., $S \vdash eq(s,s) = \textbf{true}$.
5. Can you prove that $push(e,s) = []$ is not derivable in S?[1]

Exercise 4.5.2 (Sets) Consider the following algebraic specification S for representing sets with elements from a given (yet unidentified) sort $\mathbb{E}$ with a function symbol eq representing equality.

Constructor func. symb.	**Additional func. symb.**	**Equations**
$\varnothing :\rightarrow \mathbb{S}$	$\in: \mathbb{E} \times \mathbb{S} \rightarrow \mathbb{B}$	(S1) $add(e, add(e',s)) = add(e', add(e,s))$
$add : \mathbb{E} \times \mathbb{S} \rightarrow \mathbb{S}$		(S2) $add(e, add(e,s)) = add(e,s)$
		(S3) $e \in \varnothing = \textbf{false}$
		(S4) $e \in add(e',s) = eq(e,e') \vee (e \in s)$

1. Extend the algebraic specification for sets with a function $\cup$ representing the union of sets.
2. Formulate the constructor induction principle for proving properties about sets.
3. Prove that $\varnothing$ is a unit of $\cup$, i.e., $S \vdash s \cup \varnothing = s$ and $S \vdash \varnothing \cup s = s$.
4. Prove $S \vdash add(e,s) \cup s' = add(e, s \cup s')$ and $S \vdash s \cup add(e,s') = add(e, s \cup s')$.
5. Prove idempotency of $\cup$, i.e., $S \vdash s \cup s = s$.
6. Prove commutativity of $\cup$, i.e., $S \vdash s \cup s' = s' \cup s$.
7. Prove associativity of $\cup$, i.e., $S \vdash s \cup (s' \cup s'') = (s \cup s') \cup s''$

Exercise 4.5.3 (Sets)

[1] This exercise may be somewhat beyond the expected level for this course. Nevertheless!

1. Give an algebraic specification $S2$ of sets using the following constructor function symbols:

$$\begin{array}{lcll} \varnothing & : & & \to \mathbb{S}, \\ \{\_\} & : & \mathbb{E} & \to \mathbb{S}, \\ \cup & : & \mathbb{S} \times \mathbb{S} & \to \mathbb{S}. \end{array}$$

2. Can you characterize $\{\_\}$ in terms of the constructor function symbols of the algebraic specification of sets from the previous exercise and the constructor function symbol add from that algebraic specification S in terms of the constructor function symbols of $S2$?

3. Prove that these algebraic specifications capture the same notion of sets.

Exercise 4.5.4 Extend the algebraic specification of the multisets and the mathematical sets over elements from a given sort $\mathbb{E}$ with an additional function symbol $setify : \mathbb{M} \to \mathbb{S}$ that turns a multiset into a set in such a way that all elements that are in the multiset are in the corresponding set and vice versa. Formalise this statement and prove it.

Exercise 4.5.5 (Lists)

1. Give an algebraic specification L for representing lists (over elements of sort $\mathbb{E}$) with constructor function symbols $[]$ for the empty list and $\lhd$ for adding an element of sort $\mathbb{E}$ to the back of a list.

2. Define additional function symbol $\rhd$ for adding an element to the front of a list.

3. Define additional function symbols $first$ and $last$ for obtaining the first and last elements from the list, respectively.

4. Define additional function symbols $tail$ for the list without its first element, rev for reversing a list, and $\bowtie$ for concatenating two lists.

5. Formulate the induction principle for proving properties about lists.

6. Prove that $L \vdash rev(rev(l)) = l$.

7. Prove associativity of concatenation.

Exercise 4.5.6 Consider the algebraic specification of lists from Exercise 4.5.5. Assume that for the sort of elements of the list $\mathbb{E}$ an ordering function $\leq$ is given.

1. Define a function rem that removes all occurrences of an element from a list.

2. Define a predicate $ordered?$ that indicates whether a list is ordered or not.

3. Specify a function $addord$ which inserts an element in a list in such a way that it maintains the ordering. Prove that this is the case, i.e., formulate this statement and give a proof.

4. Define an operation $remord$ which removes all occurrences of an element from a list that is assumed to be ordered. Prove that

$$rem(e, q) = \text{if } ordered?(q) \text{ then } remord(e, q) \text{ else } rem(e, q)$$

Exercise 4.5.7 (Binary Tree)

1. Give an algebraic specification of binary trees where each node of the binary tree contains an element of sort $\mathbb{N}$.

2. Provide a function $contains$ for checking whether a binary tree contains a certain value.

3. Define an equality function eq on binary trees.

4. Define a function set on binary trees which gives the set of elements that occur in a binary tree.
5. Formulate the induction principle for proving properties about binary trees.
6. Prove that for binary trees the following property holds:

$$BT \vdash contains(e, t) \Leftrightarrow e \in set(t) = \textbf{true}$$

Exercise 4.5.8 (Ordered binary trees) An ordered binary tree is an ordered binary tree for which for all nodes the following property holds: the value associated with the node is larger than the values of the nodes in the left subtree and smaller than or equal to the values of the nodes in the right subtree. We use the sort $\mathbb{BT}$ we have just defined in the previous exercise.

1. Define a function $ordered : \mathbb{BT} \to \mathbb{B}$ that indicates whether a binary tree is ordered.
2. Define a function $add : \mathbb{E} \times \mathbb{BT} \to \mathbb{BT}$ for adding a value to an ordered binary tree maintaining orderedness.
3. Give a more 'efficient' version of the function $contains$, named $ordcontains$, that, when applied to ordered binary trees, is equal to the function $contains$. Prove that it is the same function on ordered binary trees.

Exercise 4.5.9 Write an algebraic specification of a sort $\mathbb{ST}$ with the following functions:

- $emptytable$ representing an empty symbol table
- $enter$ for entering a symbol and its type into a symbol table
- $lookup$ for obtaining the type of a given symbol
- $delete$ for removing a symbol, type pair from a given table based on a given symbol
- $replace$ for replacing the type of a given symbol in a given table

Discuss undefinedness of the functions and when necessary give a precondition that guarantees definedness.

4.6 A note on eq versus $=$

One can define a predicate eq as an additional function symbol to represent some notion of equality. This notion of equality and derivability (=) can differ as the following examples indicate. In the first example eq and $=$ are indeed one and the same. In the second example, eq identifies more terms than $=$. The last example shows a specification where $=$ identifies more terms than eq does. Consequently, one can derive **true** $=$ **false**. Such algebraic specifications are called *inconsistent*.

In each of the examples there is a sort $\mathbb{N}$ with constructor function symbols $0 :\to \mathbb{N}$ and $S : \mathbb{N} \to \mathbb{N}$ and additional function symbol $eq : \mathbb{N} \times \mathbb{N} \to \mathbb{B}$.

Example 4.6.1 In this example there are no equations only involving constructor function symbols, and the additional function symbol eq is defined by the following equations:

$$eq(0, 0) = \textbf{true}$$
$$eq(0, S(m)) = \textbf{false}$$
$$eq(S(m), 0) = \textbf{false}$$
$$eq(S(m), S(n)) = eq(m, n)$$

Note that here it is the case that the notion of equality by means of eq and derivability (=) coincide: $N \vdash eq(s, s') = \textbf{true}$ iff $N \vdash s = s'$.

Example 4.6.2 In this example one more defining equation is given for eq:

$$
\begin{aligned}
&eq(0,0) = \textbf{true}\\
&eq(0,S(m)) = \textbf{false}\\
&eq(S(m),0) = \textbf{false}\\
&eq(S(m),S(n)) = eq(m,n)\\
&eq(S(S(m)),n) = eq(m,n)
\end{aligned}
$$

As a consequence it is no longer the case that $N \vdash eq(s,s') = \textbf{true}$ implies $N \vdash s = s'$, but it is still the case that $N \vdash eq(s,s') = \textbf{true}$ if $N \vdash s = s'$. A counterexample for the first statement is $N \vdash eq(S(S(0)),0) = \textbf{true}$, but $N \nvdash S(S(0)) = 0$.

Example 4.6.3 In this example, the equations for eq are the same as in the first example, but instead an equation is added between constructor function symbols.

$$
\begin{aligned}
&eq(0,0) = \textbf{true}\\
&eq(0,S(m)) = \textbf{false}\\
&eq(S(m),0) = \textbf{false}\\
&eq(S(m),S(n)) = eq(m,n)\\
&S(S(m)) = m
\end{aligned}
$$

Now one can derive

$$N \vdash \textbf{true} = eq(0,0) = eq(0,S(S(0))) = \textbf{false}.$$

Exercise 4.6.1 Given the algebraic specification of Example 4.6.1, show that $N \vdash eq(s,s') = \textbf{true}$ iff $N \vdash s = s'$.

Exercise 4.6.2 Extend the algebraic specification of mathematical sets, natural numbers and Booleans with an additional function symbol $\# : \mathbb{S} \to \mathbb{N}$ that determines the cardinality of a set and an equality function $eq : \mathbb{S} \to \mathbb{B}$. Prove that $eq(s,s') \Rightarrow eq(\#(s),\#(s'))$.

4.7 Semantics of Abstract Data Types

Definition 4.7.1 (Σ-Algebra) Let (S, Σ_c, Σ_a) be a signature. A Σ-algebra is a collection of sets $(A_s)_{s \in S}$ and for each function symbol $f : s_1 \times \cdots \times s_n \to s \in \Sigma_c \cup \Sigma_a$ a function $f_A : A_{s_1} \times \cdots \times A_{s_n} \to A_s$. A mapping from $\Sigma_c \cup \Sigma_a$ to the functions of the algebra is called an interpretation.

Example 4.7.2 (Sort $\mathbb{B}$) The set $\{\textbf{true}, \textbf{false}\}$ with standard operations $\neg$, $\wedge$ and $\vee$ is a Σ-algebra for the sort of the Booleans.

Example 4.7.3 (Sort $\mathbb{N}$) The set $\{0,1,2,\cdots\}$ with interpretation

$$
\begin{array}{lcl}
0 & \mapsto & 0\\
S & \mapsto & \_ + 1\\
+ & \mapsto & +\\
\times & \mapsto & \times
\end{array}
$$

is a Σ-algebra for the sort of the natural numbers.

Alternatively, also the set $\{\textbf{true}, \textbf{false}\}$ with interpretation

$$
\begin{array}{lcl}
0 & \mapsto & \textbf{true}\\
S & \mapsto & \neg\\
+ & \mapsto & \vee\\
\times & \mapsto & \wedge
\end{array}
\qquad \text{or with interpretation} \qquad
\begin{array}{lcl}
0 & \mapsto & \textbf{false}\\
S & \mapsto & \neg\\
+ & \mapsto & \oplus\\
\times & \mapsto & \wedge
\end{array}
$$

are Σ-algebras for the sort $\mathbb{N}$.

Definition 4.7.4 An equation $t = t'$ is valid in algebra A, notation $A \models t = t'$, if and only if $\sigma(t) = \sigma(t')$ for any mapping σ from variables to elements of A.

Example 4.7.5

- **true** $\wedge\ b = b$ is valid;
- $b \Rightarrow \neg b =$ **false** is not valid since for $b =$ **false**, we can not have **true** $=$ **false**;
- $S(S(0)) = 0$ is not valid in algebra of natural numbers, but it is valid in the other two algebras.

Definition 4.7.6 (Model) A $\Sigma_c \cup \Sigma_a$-algebra A is a model of an algebraic specification $T = ((S, \Sigma_c, \Sigma_a), E)$ w.r.t. interpretation ι if and only if every derivable equation $T \vdash t = t'$ is valid in algebra A, $A \models \iota(t) = \iota(t')$

Example 4.7.7 (Booleans) The set $\{$**true**, **false**$\}$ with standard operations $\neg$, $\wedge$ and $\vee$ is a model for the sort of the Booleans.

The equations of $\mathbb{B}$ are all valid in the algebra.

$\neg$**true** $=$ **false**	**true** $\wedge\ b = b$	**true** $\vee\ b =$ **true**
$\neg$**false** $=$ **true**	**false** $\wedge\ b =$ **false**	**false** $\vee\ b = b$

Example 4.7.8 (Models of sort $\mathbb{N}$) The set $\{0, 1, 2, \cdots\}$ with interpretation

$$\begin{array}{lcl} 0 & \mapsto & 0 \\ S & \mapsto & \_ + 1 \\ + & \mapsto & + \\ \times & \mapsto & \times \end{array}$$

is a model as can be seen from the table below

Equation in N	**Equation in model**	**Valid?**
$0 + x = x$	$0 + x = x$	yes
$S(x) + y = S(x + y)$	$(x + 1) + y = (x + y) + 1$	yes
$0 \times x = 0$	$0 \times x = 0$	yes
$S(x) \times y = y + (x \times y)$	$(x + 1) \times y = y + (x \times y)$	yes

The set $\{$**true**, **false**$\}$ with interpretation

$$\begin{array}{lcl} 0 & \mapsto & \textbf{true} \\ S & \mapsto & \neg \\ + & \mapsto & \vee \\ \times & \mapsto & \wedge \end{array}$$

is not a model since there is a derivable equality that is not valid in the model. Actually each of the equations of the algebraic specification is not valid.

Equation in N	**Equation in model**	**Valid?**
$0 + x = x$	**true** $\vee\ x =$ **true**	no
$S(x) + y = S(x + y)$	$\neg x \vee y = \neg(x \vee y)$	no
$0 \times x = 0$	**true** $\wedge\ x =$ **true**	no
$S(x) \times y = y + (x \times y)$	$\neg x \wedge y = y \vee (x \wedge y)$	no

On the other hand, the set $\{$**true**, **false**$\}$ with interpretation

$$\begin{array}{lcl} 0 & \mapsto & \textbf{false} \\ S & \mapsto & \neg \\ + & \mapsto & \oplus \\ \times & \mapsto & \wedge \end{array}$$

is a model:

Equation in N	Equation in model	Valid?
$0 + x = x$	$\mathbf{false} \oplus x = x$	yes
$S(x) + y = S(x + y)$	$\neg x \oplus y = \neg(x \oplus y)$	yes
$0 \times x = 0$	$\mathbf{false} \wedge x = \mathbf{false}$	yes
$S(x) \times y = y + (x \times y)$	$\neg x \wedge y = y \oplus (x \wedge y)$	yes

In general, models as defined thus far, can easily contain more elements that intended. In the light of our constructor induction principle presented before, it is necessary to only allow models that guarantee that any element from the algebra can be represented by constructor function symbols only. Furthermore, we require that all equalities between closed constructor terms are derivable from the algebraic specification.

Definition 4.7.9 (Constructor model) A $\Sigma_c \cup \Sigma_a$-algebra A is a constructor model of an algebraic specification $T = ((S, \Sigma_c, \Sigma_a), E)$ w.r.t. interpretation ι if and only

Model A is a model of $((S, \Sigma_c, \Sigma_a), E)$

No junk For any element a from A there exists a closed term s over Σ_c such that $A \models \iota(s) = a$

No confusion For any closed terms t and t' over Σ_c: $A \models \iota(t) = \iota(t')$ implies $T \vdash t = t'$

Consider the following alternative algebraic specification for the Booleans:

Constructor function symbols	Additional function symbols	Equations
$\mathbf{true} :\rightarrow \mathbb{B}$	$\mathbf{false} :\rightarrow \mathbb{B}$	$\neg\neg b = b$
$\neg : \mathbb{B} \rightarrow \mathbb{B}$	$\wedge: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$	$\mathbf{false} = \neg\mathbf{true}$
	$\vee: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$	$\mathbf{true} \wedge b = b$
		$\mathbf{false} \wedge b = \mathbf{false}$
		$b \vee b' = \neg(\neg b \wedge \neg b')$

Is this an algebraic specification of the Booleans? Let's first consider this question by studying the constructor model of this algebraic specification.

1. Is it a model? All equations are valid, so it is!
2. Does the algebra have junk? No, **true** is represented by **true** and **false** by $\neg$**true**.
3. Does the algebra have confusion? All valid equations between constructor terms are derivable.

Another way of answering the question whether this is an algebraic specification of the Booleans is to study whether it allows for the same derivations as another algebraic specification for the Booleans (that we consider adequate): Is this algebraic specification "equivalent" to the standard one given before?

1. Signatures contain the same symbols
2. All equations from AB are derivable from B

$$\begin{array}{|lcl|}\hline \neg\neg b & = & b \\ \mathbf{false} & = & \neg\mathbf{true} \\ \mathbf{true} \wedge b & = & b \\ \mathbf{false} \wedge b & = & \mathbf{false} \\ b \vee b' & = & \neg(\neg b \wedge \neg b') \\ \hline \end{array} \vdash \boxed{\begin{array}{l} \neg\mathbf{true} = \mathbf{false} \\ \neg\mathbf{false} = \mathbf{true} \\ \mathbf{true} \wedge b = b \\ \mathbf{false} \wedge b = \mathbf{false} \\ \mathbf{true} \vee b = \mathbf{true} \\ \mathbf{false} \vee b = b \end{array}}$$

3. and vice versa

$$\boxed{\begin{array}{l} \neg\mathbf{true} = \mathbf{false} \\ \neg\mathbf{false} = \mathbf{true} \\ \mathbf{true} \wedge b = b \\ \mathbf{false} \wedge b = \mathbf{false} \\ \mathbf{true} \vee b = \mathbf{true} \\ \mathbf{false} \vee b = b \end{array}} \vdash \begin{array}{|lcl|}\hline \neg\neg b & = & b \\ \mathbf{false} & = & \neg\mathbf{true} \\ \mathbf{true} \wedge b & = & b \\ \mathbf{false} \wedge b & = & \mathbf{false} \\ b \vee b' & = & \neg(\neg b \wedge \neg b') \\ \hline \end{array}$$

Exercise 4.7.1 Consider the algebraic specification of the Booleans. Give an algebra with just one element that is a model for this algebraic specification. Show that this algebra is not a constructor model.

Chapter 5

Z

5.1 Introduction

Z is a specification language for structuring mathematical specification of computer systems. It has rigorous set-theoretic foundations and has been used successfully in specifying computer systems, including business information systems. The **Z** notation has been standardized by ISO (International Organization for Standardization) in 2002 under standard ISO/IEC 13568:2002, entitled Information Technology - Z Formal Specification Notation - Syntax, Type System and Semantics.

A specification in **Z** is meant to be descriptive, i.e., to specify *what* the state of the system and its behavior are and not *how* they are to be implemented. It provides a very powerful specification language that allows for very abstract and *non-executable* specifications. Inspired by **Z** a number of light-weight, executable and analyzable specification languages have been introduced, among which Alloy [3] is most notable. A specification in **Z** is *model-based*, i.e., it focuses on the specifying the state of the system and possible transformations thereon. Variants of **Z** have been developed that extend it with behavioral specification [2].

We chose **Z** *not* because we advocate using this particular formalism in software design; rather, we would like to advocate its underlying principles: its descriptive style and its high level of abstraction. In the end, we will not be using and we do not necessarily want to use the full power of **Z**. We would like to communicate the main ideas behind **Z** and show how software structure can be simplified and presented in a rigorous fashion.

In this chapter, we will focus on some basic aspects of the **Z** notation. We shall try to give a general flavor of the specification in **Z** and at times, will leave out the precise mathematical meaning of the **Z** constructs. The interested reader may refer to [5, 7] for a general introduction to **Z** (treating many concepts that are not treated in this chapter) and consult [4] for the mathematical underpinnings of the **Z** constructs.

5.2 The Mathematical Toolkit of Z

5.2.1 Introduction

We mentioned using mathematics as a specification language before in this course. Formal methods and languages are supposed to provide us with a higher-level syntax (with an underlying mathematical meaning) that is suitable and convenient for specifying software and hardware related phenomena. **Z** is no exception; it provides some means for structuring and modularizing specification but otherwise, it is an abstraction layer over set theory. **Z** provides us with a basic language, which we call *"The Mathematical Toolkit of Z"*, for specifying basic and common mathematical objects such as sets, relations and function. In this section, we will review this mathematical toolkit. In Sections 5.3 and 5.4, we introduce constructs that are used to give a suitable structure to these mathematical objects in order to represent concepts of a computer system such as states and operations.

5.2.2 Sets

Types and sets. **Z** notation is based on *typed set theory*. The word *typed* means that in order to build a set one should draw its elements from a particular universe of elements, called *type*. The type of an element x is the largest set s to which it belongs, i.e., $x \in s$. One may use pre-defined type, e.g., set of integers $\mathbb{Z}$, or construct new types using the constructs given below or assume some given types without knowing further about their internal structure.

To declare variables, one should define the set over which they range. For example, variables x and y ranging over natural and integer numbers are written as $x : \mathbb{N}$ and $y : \mathbb{Z}$. The type of both variables is however $\mathbb{Z}$ and thus, the expression $x = y$ is well-typed.

Sets: enumeration. There are different ways to construct a set. One way, called *enumeration*, is to represent it with its (finite) collection of elements. The type *TypeFace* containing elements *normal*, *bold* and *italic* is defined in **Z** as follows.

$$\mathit{TypeFace} == \{\mathit{normal}, \mathit{bold}, \mathit{italic}\}$$

Sets: comprehension. Sets may also be constructed by taking an already defined set and choosing those elements therein which satisfy a certain predicate. This construction is called set *comprehension* and is illustrated by means of the following examples.

$$\mathbb{N} == \{x : \mathbb{Z} \mid x \geq 0 \bullet x\}$$

The set of natural numbers is defined above, as the set of integer that are at least 0.

$$\mathit{PosEven} == \{x : \mathbb{N} \mid x \neq 0 \bullet 2 * x\}$$

The set $PosEven$ is the set of all positive natural numbers. The same can alternatively be defined as follows.

$$\mathit{PosEven2} == \{x : \mathbb{N} \mid (x \neq 0 \land \exists y : \mathbb{N} \bullet x = 2 * y) \bullet x\}$$

When the expression after the dot (•) is the same as the expression before the bar (|), the dot and the expression afterwards can be omitted for brevity. Thus, the set specified above can be written more concisely as follows.

$$\mathit{PosEven2} == \{x : \mathbb{N} \mid x \neq 0 \land \exists y : \mathbb{N} \bullet x = 2 * y\}$$

The *empty set of integers* can be specified as follows.

$$\mathit{EmptyInt} == \{x : \mathbb{Z} \mid \mathit{false} \bullet x\}$$

Note that empty sets of different types are different. Usually, by mis-using notation, one refers generically to the empty set of appropriate type by $\varnothing$. The notation for predicates is explained further in Section 5.2.4. For each $a, b : \mathbb{Z}$, the set of integers between a and b (inclusive) is denoted by $a \mathinner{..} b$ and stands for $\{x : \mathbb{Z} \mid a \leq x \leq b\}$. If $a > b$, then $a \mathinner{..} b$ is the empty set of integers.

Given sets. One may assume the definition of sets of which the exact definition is not relevant or cumbersome to give. Such sets are called *given sets*. Suppose that one needs to use the set of characters in the specification without defining them by enumeration. Then, one can write.

$$[\mathit{Char}]$$

The above statement declares that there is a given set of characters, called *Char* which can be used in the specification.

There are a number of sets whose definitions (either by comprehension or as a given set are assumed and one can use them readily in the specification; these sets include $\mathbb{B}$ for booleans, $\mathbb{N}$ for (non-negative) natural numbers, and $\mathbb{N}_1$ for positive natural numbers.

Sets: operations. Sets can be formed by applying the usual set theoretic notations as given below.

- union $\cup$, intersection $\cap$, difference $\setminus$,
- power set $\mathbb{P}$, finite power set $\mathbb{F}$, the set of of non-empty subsets $\mathbb{P}_1$, the set of finite and non-empty subset $\mathbb{F}_1$, and
- Cartesian product $\times$.

The cardinality of set A (i.e., the number of its elements) is denoted by $\#A$.

One should note that elements of different types are incomparable. For example, consider two elements o and a, respectively, of distinct given types $[Oranges]$ and $[Apples]$. Then expressions $o = a$ and $o \neq a$ are both meaningless since the type of the two elements o and a do not match, that is one cannot compare (sets of) apples and oranges!

5.2.3 Relations and Functions

Relations: how to define. Relations and functions are sets which are used extensively in practice and thus have been provided with dedicated notation. The set of all relations between sets U and V is denoted by $U \leftrightarrow V$, which is in fact an acronym for $\mathbb{P}(U \times V)$. A finite relation, i.e. a relation that contains a finite number of pairs, can also be represented by a finite set of pairs. For example, the following set may represent a phone book relating person names with their telephone numbers.

$$PhoneBook == \{(Alice, 2345), (Alice, 1234), (Bob, 1234), (Sue, 5765)\}$$

The same set may also be represented as follows.

$$PhoneBook == \{Alice \mapsto 2345, Alice \mapsto 1234, Bob \mapsto 1234, Sue \mapsto 5765\}$$

For an n-tuple $t = (a_1, \ldots, a_n)$, and for each positive $i \leq n$ the ith component is denoted by $t.i$. Thus, $(Alice \mapsto 2345).1 = Alice$ and $(Alice \mapsto 2345).2 = 2345$.

Relations: domain, range, inverse and image. The domain and the range of a relation R are denoted by **dom**R and **ran**R, respectively. Assume that a relation R is of type $U \leftrightarrow V$, then the restriction of its domain to a set U' is denoted by $U' \lhd R$ and stands for the following set.

$$\{p : R \mid p.1 \in U'\}$$

For example, consider the relation R given before, $\{Alice\} \lhd R$ gives us all the pairs whose first component is *Alice*, that is.

$$\{Alice\} \lhd PhoneBook = \{Alice \mapsto 2345, Alice \mapsto 1234\}$$

To obtain the set of telephone numbers of *Alice*, one may thus write **dom**$(\{Alice\} \lhd PhoneBook)$. Range restriction of R to a set V' is denoted by $R \rhd V'$ and is defined analogously. For the relation *PhoneBook*, one may want to project on pairs with telephone number 1234, which is given by the following expression.

$$PhoneBook \rhd \{1234\} = \{Alice \mapsto 1234, Bob \mapsto 1234\}$$

One may exclude set U' from the domain of R, denoted by $U' \lhd\!\!\!\!- R$, which is formally defined as the following expression.

$$\{p : R \mid p.1 \notin U'\}$$

Analogously, one may exclude set V' from the range of R, which is denoted by $R \rhd\!\!\!\!- V'$. The inverse of a relation R is denoted by $R^\sim$ and is given by the following definition.

$$R^\sim = \{v \mapsto u : V \times U \mid u \mapsto v \in R\}$$

The image of a set S with respect to a relation $R : U \leftrightarrow V$, denoted by $R(\!| S |\!)$, is the set of all elements to which an element of S is mapped by R. Formally, the image of S with respect to R is defined below.

$$R(\!| S |\!) == \{v : V \mid \exists u : U \bullet u \in S \land u \mapsto v \in R\}$$

For example, in our *PhoneBook* relation, a query concerning the telephone numbers of *Alice* and *Sue* can be written as follows.

$$\mathit{PhoneBook}(\!| \{\mathit{Alice}, \mathit{Sue}\} |\!) = \{1234, 2345, 5765\}$$

Similarly, a query for the name of all people with telephone number 1234 is written as follows.

$$\mathit{PhoneBook}^{\sim}(\!| \{1234\} |\!) = \{\mathit{Alice}, \mathit{Bob}\}$$

Functions: definition and varieties. Functions are special types of relations satisfying the following predicate.

$$\mathit{func}(R) == \forall x : \mathbf{dom} R \bullet \#(\{x\} \lhd R) \leq 1$$

In other words, a (partial) function maps each element of its domain to at most one element of the range. The set of all partial functions from U to V, denoted by $U \pfun V$, is the subset of $U \rel V$ satisfying the above predicate. The set of all total functions from U to V, denoted by $U \fun V$ is the set of all partial functions f such that $\mathbf{dom} f = U$, i.e., each element of U is mapped to exactly one element of the range. There are other notations for different sorts of functions which are summarized in Table 5.1 (taken from [5, Figure 5.2]). To summarize the intuition behind these notations, a function is called injective if it does not map any two elements of the domain to the same element in the range; it is surjective if it maps at least one element to each element of the range (i.e., the mapping covers the range) and it is bijective if it is both injective and surjective. A function is finite when its domain is finite. Figure 5.1 depicts a Venn diagram relating varieties of relations and functions.

Name	Symbol	**dom**f	1-to-1	**ran**f
Total function	$\fun$	$= U$		$\subseteq V$
Partial function	$\pfun$	$\subseteq U$		$\subseteq V$
Total injection	$\finj$	$= U$	Yes	$\subseteq V$
Partial injection	$\pinj$	$\subseteq U$	Yes	$\subseteq V$
Total surjection	$\fsur$	$= U$		$= V$
Partial surjection	$\psur$	$\subseteq U$		$= V$
(Total) Bijection	$\bij$	$= U$	Yes	$= V$
Finite partial function	$\ffun$	$\in (\mathbb{F}\, U)$		$\subseteq V$
Finite partial injection	$\finj$	$\in (\mathbb{F}\, U)$	Yes	$\subseteq V$

Table 5.1: Notations for Varieties of Functions from U to V

To define functions, one may also use the lambda notation. For example, consider the following definition for function $f : \mathbb{Z} \pfun \mathbb{Z}$, which maps each natural number not greater than 5 to its square.

$$f == \lambda n : \mathbb{Z} \mid n \leq 5 \bullet n^2$$

Function f is not a total function, since integers greater than 5 are not included in the domain; it is not injective since it maps -1 and 1 both to 1; it is not surjective either since negative integers are not mapped to.

One may define a total function $tf : \mathbb{Z} \fun \mathbb{Z}$ for square as follows.

$$tf == \lambda n : \mathbb{Z} \mid \mathbf{true} \bullet n^2$$

When the predicate before the dot (**•**) is **true**, it can be omitted. Hence, function tf can also be defined as follows.

$$tf == \lambda n : \mathbb{Z} \bullet n^2$$

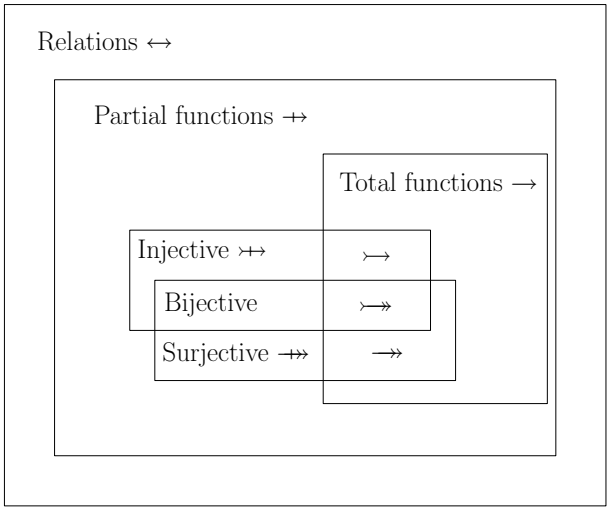


Figure 5.1: Venn Diagram Representing Varieties of Relations and Functions (from[5, Figure 5.3])

Composition and overriding Given two relations $R : U \leftrightarrow V$ and $S : V \leftrightarrow W$, their composition, denoted by $R \fatsemi S$ is defined as follows.

$$R \fatsemi S == \{(u, w) : U \times W \mid \exists v : V \bullet (u, v) \in R \land (v, w) \in S\}$$

For each two functions $f, g : U \pfun V$, overriding f by g, denoted $f \oplus g$ is defined as $(\mathbf{dom} g \domres f) \cup g$. The value of $f \oplus g$ is the same as the value of g, where g is defined and otherwise it is the value of f.

5.2.4 Logical Formulae

A logical formula can be built by means of the following constructions.

- Atomic propositions can be formed by using membership $\in$ or subset $\subseteq$ or strict subset $\subset$ relations. Atomic propositions **true**, **false** $\in \mathbb{B}$ are also included in the language.
- Predicates can be combined using binary connectors for conjunction ($\land$), disjunction ($\lor$), implication ($\Rightarrow$) and bi-implication ($\Leftrightarrow$) and a unary operator for negation ($\neg$).
- Predicates can be formed by using quantifiers to quantify variables of defined types; using this construction, the generic form of a predicate is $\exists x : U \bullet Pred$ for existential quantification, $\forall x : U \bullet Pred$ for universal quantification, and $\exists_1 x : U \bullet Pred$ for unique existential quantification.
- Logical definitions: One may define an element e of a given type by using the statement $e = \mu x : U \mid Pred \bullet t$. Informally, this statement reads: e is the result of replacing x in t with the unique element in U satisfying $Pred$. For example, one may define $Amsterdam = \mu\, country : Countries \mid people\ wear\ clogs\ in(country) \bullet capital(country)$; this is an acronym for $\exists_1 country : Countries \bullet people\ wear\ clogs\ in(country) \land \exists\, country : Countries \mid people\ wear\ clogs\ in(country) \bullet capital(country) = Amsterdam$.

5.2.5 Sequences and Bags

Sequences are useful constructs in specifications. A sequence of elements in U can be specified using a function f from $\mathbb{N}$ to U (note that an element in U may appear zero or more times in a sequence); for such a function, we require that its domain is exactly the set of integers from 1 to the size of the sequence. Thus,

$f(i)$, if defined, denotes the element at the position i of the sequence. The set of all sequences of elements in (a subset of) U is denoted by $\mathbf{seq}U$ and the informal explanation of the constraints on functions in $\mathbf{seq}U$ is formalized as follows.

$$\mathbf{seq}U == \{f : \mathbb{N} \nrightarrow U \mid \mathbf{dom}f = 1..\#f\}$$

Sequences may be denoted by the sequence of elements as illustrated by the following example.

$$line = \langle T, h, i, s, \square, i, s, \square, a, \square, l, i, n, e\rangle$$

The empty sequence is given by $\langle\rangle$ in this notation.

The head of a sequence $s : \mathbf{seq}U$, i.e., $s(1)$, is also denoted by $head\, s$; its tail is denoted by $tail\, s$ and its formal definition is given below.

$$tail(s) = \{(i, u) : \mathbb{N} \times U \mid (i + 1, u) \in s\}$$

The reverse of a sequence is denoted by $rev\, s$. Given two sequences $s, t : \mathbf{seq}U$, their concatenation is denoted by $s \frown t$.

Bags (also called multisets) are collections of elements in which repetition is relevant but order is not relevant. Bags with elements from U are specified by functions $U \nrightarrow \mathbb{N}$ denoting the number of repetitions of each element in U.

Exercise 5.2.1 Give the definition for a function $equiv : (A \leftrightarrow A) \rightarrow \{\mathbf{true}, \mathbf{false}\}$, which given a relation on A determines whether it is an equivalence or not.

Exercise 5.2.2 Give the definition for $A \rightarrowtail\!\!\!\!\!\twoheadrightarrow B$, by only using the concept of relation.

Exercise 5.2.3 Give the generic definitions of the functions for the reverse of a sequence and the concatenation of two sequences.

Exercise 5.2.4 Give the generic formal definition of $squash : (\mathbb{Z} \twoheadrightarrow X) \rightarrow \mathbf{seq}X$, which renumbers the domain of its argument (a partial finite function) to produce a sequence. For example, if $f = \{(-2 \mapsto 1), (0 \mapsto 2), (3 \mapsto 3)\}$, then $squash(f)$ is $\langle 1, 2, 3\rangle$.

Exercise 5.2.5 Give definitions for finite power set $\mathbb{F}$, the set of of non-empty subsets $\mathbb{P}_1$, the set of finite and non-empty subset $\mathbb{F}_1$ in terms of the power set $\mathbb{P}$ and set comprehension.

Exercise 5.2.6 For each of the areas of Figure 5.1, give an example of a function (on natural numbers) that belongs to this area.

5.3 Global Definitions

A **Z** specification comprises the following parts:

Given Sets
Type Definitions
Axiomatic Definitions
State Schemas
Operation Schemas
Initialization Schemas
Schema – Calculus Expressions

In this section, we give an overview of the constructs use in each of the above-given parts.

5.3.1 Free Types

Apart from the constructs given before, one may use free types to define new types. Free types resemble the definition of Abstract Data Types (ADTs) in algebraic specifications. For example, consider the definition of *TypeFace* in Section 5.2.2. One could define the abstract data type to represent type faces using the following axiomatic definition.

$$TypeFace == normal \mid bold \mid italic$$

It defines a type *TypeFace* with three pairwise *distinct* elements, *normal, bold* and *italic*.
The following specification, defines the type of binary trees of natural numbers.

$$BinTreeNat == leaf\langle\langle \mathbb{N} \rangle\rangle \mid Node\langle\langle BinTreeNat, BinTreeNat \rangle\rangle$$

The above definition asserts that a natural number is a leaf of a binary tree (by applying the constructor *leaf* to it) and a binary tree can be constructed by taking two binary trees and composing them using the constructor *Node*.

5.3.2 Axiomatic Definitions

Sometimes it is necessary to define types and variables that will be used throughout the specification. One way to realize this is through axiomatic definitions. For example, suppose that we would like to declare that a line has at most 80 characters, one can define the constant *MaxLine* as follows.

$$MaxLine : \mathbb{N}$$
$$\hline$$
$$MaxLine = 80$$

Suppose that one wants to distinguish a subset of characters as "control characters" and designate a particular character "br" to denote line break. These are achieved by the following axiomatic definition.

$$CtrlChar : \mathbb{P}\, Char$$
$$br : Char$$

Another example of axiomatic definitions concerns the definition of sets. Suppose that we would like to define a file name as a sequence of characters. The following definition realizes this.

$$FileName == \mathbf{seq} Char$$

The following axiomatic definition defines the concept of line by specifying that at the end of each line, there is be a line break and its length should not exceed the specified maximum length. (For simplicity, we did not check whether the last line break is the only line break in the line.)

$$Line : \mathbb{P}\, \mathbf{seq} Char$$
$$\hline$$
$$\forall\, l : Line \bullet l(\#l) = br \land \#l \leq MaxLine$$

If the predicate part of an axiomatic definition is **true**, it can be omitted.

Generic axiomatic definitions can be used to give definitions for arbitrary types. For example, the following axiomatic definition, defines when two sequences of arbitrary elements are permutations of each other. Note that

5.4 Schemas

5.4.1 Introduction

One may use sets, relations and predicates to specify a computer system but for any real application, this approach will be bound for failure. By using the basic mathematics, one will end up with pages of unstructured mathematics which will be difficult, if not impossible, to understand and communicate. To overcome this problem **Z** provides some structuring constructs. In particular, *schemas* are the main building blocks of a **Z** specification. Schemas encapsulate the state of a module and specify relevant operations on them.

5.4.2 State Schemas

A state schema declares the local state of a software module and the invariants on the state. (Invariants are predicates specifying constraints that should always hold, regardless of the state change.) A state schema consists of three parts: name, declaration part and predicate part. The name part gives a name to the aspect of the system (the module) being specified. The declaration part defines a number of variables of certain types. Declarations may be separated by a semicolon or by line breaks. The predicate part defines invariants on these variables. The predicates may be separated by logical connectives or line breaks, which stand for conjunctions. When the predicate part is **true**, it can be omitted.

Consider the following example which defines the notion of a file system. A file has a name and contains a sequence of lines. The predicate part requires that the name of the file does not contain control characters.

FileSystem

$$fs : FileName \pfun \mathbf{seq} Line$$

$$\forall n : FileName \mid n \in \mathbf{dom} fs \bullet \mathbf{ran} n \cap CtrlChar = \varnothing$$

The name of the above schema is *FileSystem*, its only declaration is $fs : FileName \pfun \mathbf{seq} Line$ and its only predicate is $\forall n : FileName \mid n \in \mathbf{dom} fs \bullet \mathbf{ran} n \cap CtrlChar = \varnothing$. If there are more declarations in a schema, they are then separated by semicolon or line break.

In addition to the graphical notation used above, schemas can be represented textually. The same schema can be written textually as $FileSystem == [fs : FileName \pfun \mathbf{seq} Line \mid \forall n : FileName \mid n \in \mathbf{dom} fs \bullet \mathbf{ran} n \cap CtrlChar = \varnothing]$.

To build systems hierarchically, schemas may be included in each other. For example, suppose one wants to define a state schema for an operating system which includes the specification of a file system, then one can write.

OperatingSystem

FileSystem

$\vdots$

$\vdots$

In the above specification, an exact copy of all declarations and predicates of *FileSystem* will be copied to the definition of *OperatingSystem*. When including a schema, common declarations should be consistent, that is, variables with the same name should have the same type. Otherwise, the inclusion is not defined.

One may think of a state schema as a set of all mappings from variables to values that satisfy the predicate part. The above concept is formalized in **Z** in terms of *bindings*, but we skip the details in this chapter.

5.4.3 Operation Schemas

Operation schemas specify possible transformations on state schemas. Before proceeding with the syntax and an examples of operation schemas, we need to treat the notion of decorated variables which are used in operation schemas.

Decorated variables. Variable names may be suffixed by a question mark ?, an exclamation mark ! or a prime $'$ to denote input variables, output variables and after-state variables, respectively. In terms of the underlying mathematical theory, these variables are just ordinary variables with different names; these notations are only used to distinguish these variables based on their intuitive meaning (by using this syntactic convention).

To make primed copies of variables and predicates in a schema, one may just use a primed copy of the schema name. For example, *FileSystem*′, where *FileSystem* is the schema given in the previous section, stands for the following schema.

FileSystem′

$$fs' : FileName \pfun \mathbf{seq} Line$$

$$\forall n : FileName \mid n \in \mathbf{dom} fs' \bullet \mathbf{ran} n \cap CtrlChar = \varnothing$$

An operation schema, similar to a state schema, has a name, a declaration part and a predicate part. In the declaration part, one may include the definition of the state schema on which the operation is defined. To include the definition of the state schema, one usually uses the notation $\Delta Name$, where *Name* denotes the name of the state schema to be imported. The statement $\Delta Name$ stands for $Name;\ Name'$, i.e., a copy and a primed copy of *Name*. In other words, by writing $\Delta Name$ in the declaration part of an operation schema, one includes two copies of the declaration and predicate parts of the schema *Name* into the operation schema: one copy with the original names of the variables, called the *before-state variables*, and one copy with the primed variables denoting the *after-state* variables (the state resulting from applying the operation schema). This way, the predicates of the state schema hold for both before-state and after-state variables. The operation is then specified by defining predicates relating before-state to after-state variables. The following example adds a new file with name *name*? and an empty content to the file system.

NewFile

$$\Delta FileSystem$$
$$name? : FileName$$

$$name? \notin \mathbf{dom} fs$$
$$fs' = fs \cup \{(name?, \langle\rangle)\}$$

An operation schema is supposed to define a relation between valuation of before-state, after-state, input and output variables, if any. If the name already exists, i.e., $name? \in \mathbf{dom} fs$, then the predicate part is false. In this case, the operation schema does not define the relation between pre-state, after-state and input valuations, if the input variable *name*? satisfies $name? \in \mathbf{dom} fs$. We return to this issue, when we define the concept of pre-condition.

One may expand the definition of Δ, resulting in a *normalized schema*. The normalized version of *NewFile* is given below.

NewFile

$$fs : \mathbf{seq} Char \pfun \mathbf{seq} Line$$
$$fs' : \mathbf{seq} Char \pfun \mathbf{seq} Line$$
$$name? : \mathbf{seq} Char$$

$$name? \notin \mathbf{dom} fs$$
$$fs' = fs \cup \{(name?, \langle\rangle)\}$$
$$\forall n : FileName \mid n \in \mathbf{dom} fs \bullet \mathbf{ran} n \cap CtrlChar = \varnothing$$
$$\forall n : FileName \mid n \in \mathbf{dom} fs' \bullet \mathbf{ran} n \cap CtrlChar = \varnothing$$

If the valuation of after-state variables is the same as the valuation of their before-state counterparts, this should be specified explicitly by expressions of the form $x = x'$, where x denotes such variables. Some operation schemas do not change the value of after-state variable at all. To avoid repeating statements of the form $x = x'$, one may include the state schema using the notation $\Xi Name$, where *Name* is the name of the state schema to be included. The following operation schema lists the name of all files present in the file system.

List

$\Xi FileSystem$
$list! : \mathbb{F}\, FileName$

$list! = \mathbf{dom} fs$

Thus, expanding the notation for Ξ, one obtains the following equivalent schema.

List

$fs : FileName \pfun \mathbf{seq} Line$
$fs' : FileName \pfun \mathbf{seq} Line$
$list! : \mathbb{F}\, FileName$

$list! = \mathbf{dom} fs$
$\forall n : FileName \mid n \in \mathbf{dom} fs \bullet$
$\quad \mathbf{ran} n \cap CtrlChar = \varnothing$
$\forall n : FileName \mid n \in \mathbf{dom} fs' \bullet$
$\quad \mathbf{ran} n \cap CtrlChar = \varnothing$
$fs = fs'$

Initialization. There is a special operation schema, named *Init*, which is in charge of initializing the local state. For example, the following schema initializes the file system to a state in which the file system is empty.

Init

$FileSystem'$

$fs' = \varnothing$

Note that there is no before-state for the *Init* schema as it is supposed to define the *initial* state of the system.

Exercise 5.4.1 Define the state schema for a file. It should comprise a name (as specified in Exercise **??**), and a content which is a sequence of lines. A line is a sequence of characters ending with a special character 'br'.

Exercise 5.4.2 Consider the state schema defined in Exercise 5.4.1.

1. Write an operation schema that given a file and a line number, inserts a new line at the specified line number and shifts the rest of the content down.
2. Write an operation schema that searches in a file (specified by its name) for a particular string possibly including a wild card "questionMark" which can match any character. The output is either "found" or "notFound".

Exercise 5.4.3 Consider a bounded sequence of a given set *El*, the maximum size of which is given by constant *n*.

1. Write a state schema for a bounded sequence.
2. Write an operation schema *Add* for adding an element to the head of a sequence.

Exercise 5.4.4 Specify a simplified model of the Irish parliament (Exercise 4 of ERD) as follows. The parliament is composed of T.D.s (teachta dála, member of parliament in Gaelic). Each T.D. can be affiliated with at most one political party. Some T.D.'s might not be affiliated with any part and may thus be independent.

1. Give the state schema for the Dáil.
2. Define the operation schema for *Renounce*, by which an independent T.D. renounces its independence and declares an affiliation with a certain political party.

5.5 Schema calculus

5.5.1 Introduction

The schema calculus of **Z** is a useful tool to extend the specifications and combine them. It facilitates modular construction of specifications. Schemas can be combined using logical connectives (such as conjunction, disjunction and implication), schemas can be negated, and one may use quantification over the declarations in schemas. Moreover, one may compose two operation schemas sequentially. Based on the schema calculus, the concepts of precondition of an operation schema and the initialization theorem (for the *Init* schema) are defined.

5.5.2 Logical Connectives

Consider two normalized schema $Schema_0$ and $Schema_1$ of the following form.

$Schema_0$
$Decl_0$

$Pred_0$

$Schema_1$
$Decl_1$

$Pred_1$

If $Decl_0$ and $Decl_1$ are consistent, i.e., variables appearing both in $Decl_0$ and $Decl_1$ have the same type, then the disjunction of $Schema_0$ and $Schema_1$, denoted by $Schema_0 \lor Schema_1$ is defined as follows.

$Schema_0 \lor Schema_1$
$Decl_0$
$Decl_1$

$Pred_0 \lor Pred_1$

Conjunction and implication between schemas and negation of a schema are defined accordingly by combining the declaration parts (in case of conjunction and implication) and taking conjunction, implication or negation of the predicate parts, respectively.

Consider the schema *NewFile* given before which adds a new file to the system. It can be observed that the schema only defines an after state when there is no file with name "*name*?" already present in the file system. Thus, the after-state of the system is not defined in case such a file already exists. To remedy this, one may define the following schema which reports an error in this case.

$[Msg]$

$error, done : Msg$

FileExists
$\Xi FileSystem$
$name? : FileName$
$report! : Msg$

$name? \in \mathbf{dom} fs$
$report! = error$

The following statement defines *ImprovedNewFile*, which combines *NewFile* with *FileExists*.

$$ImprovedNewFile == NewFile \lor FileExists$$

Spelling out the definition of $\lor$ given above (and some simplification) results in the following schema.

ImprovedNewFile

$\Delta FileSystem$
$name? : \mathbf{seq} Char$
$report! : Msg$

$(name? \notin \mathbf{dom} fs$
$fs' = fs \cup \{(name?, \langle\rangle)\}) \lor$
$(name? \in \mathbf{dom} fs$
$fs' = fs$
$report! = error)$

The after state of the system is now defined for both cases $name? \notin \mathbf{dom} fs$ and $name? \in \mathbf{dom} fs$. Namely, if the input file name is not present in the file system, the file will be added, or otherwise, an error message will be reported. However, for the case $name? \in \mathbf{dom} fs$, the value of *error*! is not defined. It might be that the file is successfully added to the file system while *report*! is given the value *error*, which is counter-intuitive. To improve the situation, one may define the schema *TotalNewFile* as follows.

Done

$report! : Msg$

$report! = done$

$$NewFileRep == NewFile \land Done$$

$$TotalNewFile == NewFileRep \lor FileExists$$

5.5.3 Quantification

One may use quantification over the variables declared in a schema. As a result, the quantified variables will be removed from the declaration of the schema and quantification is moved into the predicate part. This is formalized by the following definitions.

$$\exists Decl_0 \bullet [Decl_1 \mid Pred] == [Decl_1 \setminus Decl_0 \mid \exists Decl_0 \bullet Pred]$$

$$\forall Decl_0 \bullet [Decl_1 \mid Pred] == [Decl_1 \setminus Decl_0 \mid \forall Decl_0 \bullet Pred]$$

The more interesting case of quantification is existential quantification which may be interpreted as *hiding* the quantified variable. Consider the schema *NewFile* and suppose that one needs to specify an operation which adds *some* file to the file system. The following schema realizes this.

$$SomeNewFile == \exists name? : FileName \bullet NewFile$$

The above schema is, by the definition of existential quantification, equal to the following schema.

SomeNewFile

$\Delta FileSystem$

$\exists name? : FileName \bullet$
$name? \notin \mathbf{dom} fs$
$fs' = fs \cup \{(name?, \langle\rangle)\}$

The above schema defines the after-state of the file system by a state in which *some* new empty file is added to the before-state. Existential quantification on schemas is also called *hiding* in the literature.

5.5.4 Composition

Schemas can be composed sequentially. Intuitively, when two operation schemas are to be composed sequentially, the after-state of the first schema is to be matched with the before-state of the second. To do this, the after-state of the first schema (i.e., primed variables) as well as before-state of the second (i.e., normal variables) are both renamed into double-primed variables. Furthermore, this matched in-between state is hidden using existential quantification. The formal definition of schema composition is given below.

$$Schema_0 \fatsemi Schema_1 == \exists State'' \bullet Schema_0[''/'] \land Schema_1[''/],$$

where $State''$ denotes the double-primed variables of both $Schema_0$ and $Schema_1$, $Schema_0[''/']$ denotes substitution of primed variables of $Schema_0$ with double-primed variables (with the same names) and $Schema_0[''/']$ denotes substitution of normal variables of $Schema_1$ with double-primed ones. Consider the operation schema

$$OneFile == Init \fatsemi NewFile,$$

which specifies the addition of one file to the empty file system. According to the definitions of $\fatsemi$, *Init* and *NewFile*, the definition of *OneFile* is equal to the following.
$\exists fs'' : FileName \pfun \mathbf{seq} Line \bullet$

$$\begin{array}{|l} Init[''/'] \\ \hline fs'' : FileName \pfun \mathbf{seq} Line \\ \hline fs'' = \varnothing \\ \hline \end{array} \quad \land \quad \begin{array}{|l} NewFile[''/] \\ \hline fs'' : FileName \pfun \mathbf{seq} Line \\ fs' : FileName \pfun \mathbf{seq} Line \\ name? : FileName \\ \hline name? \notin \mathbf{dom} fs'' \\ fs' = fs'' \cup \{(name?, \langle\rangle)\} \\ \hline \end{array}$$

By applying the definition of $\land$, we obtain.
$\exists fs'' : FileName \pfun \mathbf{seq} Line \bullet$

$$\begin{array}{|l} Init[''/'] \land NewFile[''/] \\ \hline fs'' : FileName \pfun \mathbf{seq} Line \\ fs' : FileName \pfun \mathbf{seq} Line \\ name? : FileName \\ \hline fs'' = \varnothing \\ name? \notin \mathbf{dom} fs'' \\ fs' = fs'' \cup \{(name?, \langle\rangle)\} \\ \hline \end{array}$$

By replacing fs'' by its equal $\varnothing$, we obtain the following schema.

$\exists fs'' : FileName \pfun \mathbf{seq} Line \bullet$

$$\begin{array}{|l} Init[''/'] \land NewFile[''/] \\ \hline fs'' : FileName \pfun \mathbf{seq} Line \\ fs' : FileName \pfun \mathbf{seq} Line \\ name? : FileName \\ \hline fs'' = \varnothing \\ fs' = \{(name?, \langle\rangle)\} \\ \hline \end{array}$$

Finally, by applying the definition of existential quantification (and simplifying the result using the one point rule, see the next section), we get the following intuitive result.

OneNewFile

$fs' : FileName \pfun \mathbf{seq} Line$
$name? : FileName$

$\exists fs'' : FileName \pfun \mathbf{seq} Line \bullet$
$fs'' = \varnothing$
$fs' = \{(name?, \langle\rangle)\}$

=

OneNewFile

$fs' : FileName \pfun \mathbf{seq} Line$
$name? : FileName$

$fs' = \{(name?, \langle\rangle)\}$

5.5.5 Precondition

Intuitively the precondition of an operation schema indicates when the after-state and output of an operation schema is defined. Consider an operation schema *Op* and assume that the collections of after-state variables and output declarations of *Op* are denoted by *AftSt* and *Output*, respectively. Then, the precondition of *Op*, denoted by **pre***Op*, is defined by the schema $\exists AftSt; \ Output \bullet Op$. For example, consider the operation schema *NewFileRep*. Then **pre***NewFileRep* is the following schema.

$\exists fs' : FileName \pfun \mathbf{seq} Line; \ report! : Msg \bullet$

NewFileRep

$\Delta FileSystem$
$name? : FileName$
$report! : Msg$

$report! = done$
$name? \notin \mathbf{dom} fs$
$fs' = fs \cup \{(name?, \langle\rangle)\}$

One can substantially simplify the above schema, using the following general recipe.

1. Expand the primed version of the state schema in the declaration part. The (unprimed) state schema can be kept unexpanded.

2. Apply the definition of existential quantification in the schema calculus and move the existential quantification inside the predicate part of the schema.

3. Apply logical simplification to simplify the predicate part of the schema. The most common used logical rule in this simplification is the following *one point* rule.

   $$\exists x : A \bullet (p \land x = t) = t \in A \land p[t/x],$$

   where t does not contain x.

4. Redundant predicates, i.e., those already included in the (unprimed) state schema can be omitted.

For example, consider the definition of **pre***NewFileRep*. By expanding the definition of $FileSystem'$, we obtain.

$\exists fs' : FileName \pfun \mathbf{seq} Line; \ report! : Msg \bullet$

NewFileRep

$$
\begin{array}{l}
FileSystem \\
fs' : FileName \pfun \mathbf{seq} Line \\
name? : FileName \\
report! : Msg \\
\hline
\forall n : FileName \mid n \in \mathbf{dom} fs' \bullet \\
\quad \mathbf{ran} n \cap CtrlChar = \varnothing \\
report! = done \\
name? \notin \mathbf{dom} fs \\
fs' = fs \cup \{(name?, \langle\rangle)\}
\end{array}
$$

By applying the definition of $\exists$, we get the following schema.

pre*NewFileRep*

$$
\begin{array}{l}
FileSystem \\
name? : FileName \\
\hline
\exists fs' : FileName \pfun \mathbf{seq} Line;\ report! : Msg \bullet \\
\quad \forall n : FileName \mid n \in \mathbf{dom} fs' \bullet \\
\qquad \mathbf{ran} n \cap CtrlChar = \varnothing \\
\quad report! = done \ \ name? \notin \mathbf{dom} fs \\
\quad fs' = fs \cup \{(name?, \langle\rangle)\}
\end{array}
$$

We can apply the one point rule on $\exists fs' : FileName \pfun \mathbf{seq} Line$ and thus, we obtain:

pre*NewFileRep*

$$
\begin{array}{l}
FileSystem \\
name? : FileName \\
\hline
\exists report! : Msg \bullet \\
\quad fs \cup \{(name?, \langle\rangle)\} \in FileName \pfun \mathbf{seq} Line \land \\
\quad \forall n : FileName \mid n \in \mathbf{dom} fs \cup \{(name?, \langle\rangle)\} \bullet \\
\qquad \mathbf{ran} n \cap CtrlChar = \varnothing \\
\quad report! = done \\
\quad name? \notin \mathbf{dom} fs
\end{array}
$$

from *FileSystem* we know that $fs \in FileName \pfun \mathbf{seq} Line$ and we have that $name? \in FileName$ and $\langle\rangle \in \mathbf{seq} Line$, thus, $fs \cup \{(name?, \langle\rangle)\} \in FileName \pfun \mathbf{seq} Line = true$.

pre*NewFileRep*

$$
\begin{array}{l}
FileSystem \\
name? : FileName \\
\hline
\exists report! : Msg \bullet \\
\quad \forall n : FileName \mid n \in \mathbf{dom} fs \cup \{(name?, \langle\rangle)\} \bullet \\
\qquad \mathbf{ran} n \cap CtrlChar = \varnothing \\
\quad report! = done \\
\quad name? \notin \mathbf{dom} fs
\end{array}
$$

By applying on point rule on $\exists report! : Msg$, we get the following schema.

pre*NewFileRep*

$$
\begin{array}{l}
\mathit{FileSystem} \\
\mathit{name}? : \mathit{FileName} \\
\hline
\mathit{done} \in \mathit{Msg} \\
\forall\, n : \mathit{FileName} \mid n \in \mathbf{dom} fs \cup \{(\mathit{name}?, \langle\rangle)\} \bullet \\
\quad \mathbf{ran} n \cap \mathit{CtrlChar} = \varnothing \\
\quad \mathit{name}? \notin \mathbf{dom} fs
\end{array}
$$

From the axiomatic definitions, we know that $\mathit{done} \in \mathit{Msg}$. Furthermore, from *FileSystem*, we have $\forall\, \mathit{name} : \mathit{FileName} \mid \mathit{name} \in \mathbf{dom} fs \bullet \mathbf{ran}\mathit{name} \cap \mathit{CtrlChar} = \varnothing$, thus, *NewFileRep* can be further simplified into the following.

pre*NewFileRep*

$$
\begin{array}{l}
\mathit{FileSystem} \\
\mathit{name}? : \mathit{FileName} \\
\hline
\mathit{done} \in \mathit{Msg} \land \mathbf{ran}\mathit{name}? \cap \mathit{CtrlChar} = \varnothing \\
\mathit{name}? \notin \mathbf{dom} fs
\end{array}
$$

The above result is very intuitive; operation schema *NewFile* adds a new file only when the name of the file does not contain control characters and furthermore, a file with the given name does not exist already.

A schema is called *total* if its precondition is a schema of the form $[\mathit{State};\ \mathit{Inps}? \mid \mathit{true}]$.

5.5.6 Initialization Theorem

Initialization theorem states that the system can always be initialized. Formally speaking, initialization theorem states $\mathbf{pre}\mathit{Init} = [\mid \mathit{true}]$, i.e., schema *Init* is total. A similar simplification recipe, as for precondition, can be applied in order to prove the initialization theorem. Consider the *Init* schema for file systems. Next, we prove the initialization theorem for *Init*.

$\exists\, fs' : \mathit{FileName} \pfun \mathbf{seq}\mathit{Line} \bullet$

Init

$$
\begin{array}{l}
\mathit{FileSystem}' \\
\hline
fs' = \varnothing
\end{array}
$$

= (Expanding the definition *FileSystem*′)

$\exists\, fs' : \mathit{FileName} \pfun \mathbf{seq}\mathit{Line} \bullet$

Init

$$
\begin{array}{l}
fs' : \mathit{FileName} \pfun \mathbf{seq}\mathit{Line} \\
\hline
fs' = \varnothing \\
\forall\, n : \mathit{FileName} \mid n \in \mathbf{dom} fs' \bullet \\
\quad \mathbf{ran} n \cap \mathit{CtrlChar} = \varnothing
\end{array}
$$

= (Definition of $\exists$)

Init

$$
\begin{array}{l}
\hline
\exists\, fs' : \mathit{FileName} \pfun \mathbf{seq}\mathit{Line} \bullet \quad fs' = \varnothing \\
\quad \forall\, n : \mathit{FileName} \mid n \in \mathbf{dom} fs' \bullet \\
\qquad \mathbf{ran} n \cap \mathit{CtrlChar} = \varnothing
\end{array}
$$

= (One point rule)

Init

$$\varnothing \in \mathit{FileName} \pfun \mathbf{seq}\mathit{Line}$$
$$\forall n : \mathit{FileName} \mid n \in \mathbf{dom}\varnothing \bullet$$
$$\mathbf{ran} n \cap \mathit{CtrlChar} = \varnothing$$

= (Def of $\varnothing$)

Init

$$\varnothing \in \mathit{FileName} \pfun \mathbf{seq}\mathit{Line}$$
$$\forall n : \mathit{FileName} \mid \mathit{false} \bullet$$
$$\mathbf{ran} n \cap \mathit{CtrlChar} = \varnothing$$

= (Definition of $\pfun$ and $\forall$)

Init

$$\mathit{true}$$

Exercise 5.5.1 Calculate the precondition of operation schema *Add* from Exercise 5.4.3.

Exercise 5.5.2 Calculate the precondition of operation schema *Renounce* from Exercise 5.4.4.

Exercise 5.5.3 Consider a school management system with the following informal specification.

- The students may enroll in a number of courses;
- after finishing the course, they get a grade between 1 and 10 and a *pass* or *fail* mark; if the grade is greater than 5, then the mark is pass, otherwise it is fail;
- Some courses are prerequisites of others; the prerequisite relation is irreflexive and transitive;
- A student may enroll in a course only if s/he has a pass mark for all the prerequisite courses.

Formalize the above specification in **Z**.

1. Define the given sets, and give the necessary axiomatic definitions, if any.
2. Define the state schema *School*.
3. Define the *Init* schema, specifying the initial state, in which no student is enrolled and has no grades.
4. Define the following operation schemata for:
   (a) enrollment in a course *Enroll*,
   (b) giving a grade and a mark to a student for a course *EnterGrade*,
   (c) adding prerequisite relation *AddPreReq*.
5. Write down and simplify $\mathit{Init} \fatsemi \mathit{Enroll}$.
6. Calculate pre-conditions for
   (a) *Enroll* and
   (b) *AddPreReq*.
7. Is *Enroll* total? If not, then extend it to a total schema (using schema calculus operators) and prove that the extended schema is indeed total.

5.6 Z and Class Diagrams

The main purpose of introducing **Z** in this chapter was to formalize the definition of classes in UML. To this end, the specification structure of **Z** is extended by the concept of classes; the resulting extension is called Object-**Z** [1, 6]. In this chapter, we may slightly deviate from the Object-**Z** style of specification (e.g., in the specification of the initialization schema) to be consistent with the **Z** standard. But most of the concepts remain identical to those presented in [1, 6].

5.6.1 Classes in Z

We start with formalizing classes by giving the following specifications:

1. Given types and axiomatic definitions, if any.

2. State schema, capturing the attributes of the class, their types and their invariants. The state schema does not have a name, because it represents the state of the class.

3. Initialization schema, formalizing the constructor of the class.

4. Operation schemata. In each operation schema, the variables that are changed are listed using the notation $\Delta(v_0, v_1, \ldots)$. If a variable in the state schema is not changed in the operation schema, then it need not be mentioned in the declaration part of the operation schema.

The following is an example of a class specified in Object-**Z**. It encapsulate the specification of a file system given below.

FileSystem

$$MaxLine : \mathbb{N}$$

$$MaxLine = 80$$

$$CtrlChar : \mathbb{P}\, Char$$
$$br : Char$$

$$FileName == \mathbf{seq} Char$$

$$Line : \mathbb{P}\, \mathbf{seq} Char$$

$$\forall l : Line \bullet l(\#l) = br \wedge \#l \leq MaxLine$$

$$fs : FileName \pfun \mathbf{seq} Line$$

$$\forall n : FileName \mid n \in \mathbf{dom} fs \bullet \mathbf{ran} n \cap CtrlChar = \varnothing$$

INIT

$$fs' = \varnothing$$

NewFile

$$\Delta(fs)$$
$$name? : \mathbf{seq} Char$$
$$report! : Msg$$

$$(name? \notin \mathbf{dom} fs \wedge fs' = fs \cup \{(name?, \langle\rangle)\} \wedge report! = done) \vee (name? \in \mathbf{dom} fs \wedge fs' = fs \wedge report! = error)$$

List

$$list! : \mathbb{F}\, FileName$$

$$list! = \mathbf{dom} fs$$

In the definition of a class one may use the definition of other classes as types, thus realizing concepts such as association, aggregation and composition. Sequences or sets can be used to realize multiple associations and the multiplicity can be enforced by a predicate on the size of the sequence / cardinality of the set.

5.6.2 Inheritance

One aspect of inheritance is extending the features or operations of a base class without touching the original features and operations. This can be simply achieved in Object-**Z** by including the base class in the extended class and adding the new features and/or operations. Suppose that you would like to add new operations *Rename* and *Del* to change the file name, and delete a file, respectively. This is achieved in the following specification.

ChFS

FileSystem

Rename

$\Delta(fs)$
oldName? : **seq***Char*
newName? : **seq***Char*
report! : *Msg*

$(oldName? \in \mathbf{dom} fs$
$newName? \notin \mathbf{dom} fs$
$fs' = fs \cup \{(newName?, fs(oldname?))\}$
$report! = done) \lor$
$((oldName? \notin \mathbf{dom} fs \lor newName? \in \mathbf{dom} fs)$
$fs' = fs$
$report! = error)$

Delete

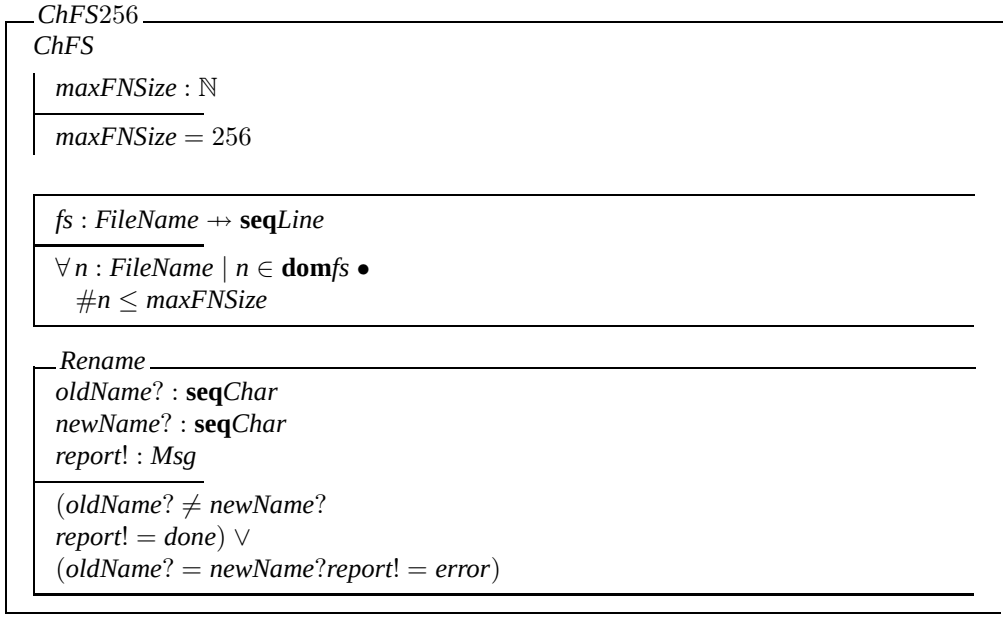
$\Delta(fs)$
name? : **seq***Char*
report! : *Msg*

$(name? \in \mathbf{dom} fs$
$fs' = fs \setminus \{(name?, fs(name?))\}$
$report! = done) \lor$
$(name? \notin \mathbf{dom} fs$
$fs' = fs$
$report! = error)$

Extension with fresh features and operations is not all that is to inheritance. More constraints can also be added to an already existing feature or operation. By default, if the state schema or operation schemata common to the base class are also present in the extended class, for each common schema, the union of the declarations (of the base and extended schema) and conjunction (logical and) of the predicates is taken. For example, suppose that we would like to extend our file system by the following two constraints:

1. file names can be at most 256 characters long and

2. in the rename operation, the old and the new file name may not coincide.



There is even more to inheritance than what we have formalized so far. One may want to modify the operations of a base class or even define them anew. This can also be done using two useful mechanisms: renaming and redefinition. When including a schema, one may rename variable and schema names, using the notation $Class[v_0'/v_0, \ldots, s_0'/s_0, ldots]$, where v_0' and s_0' are the new names for v_0 and s_0, respectively. One may also use the notation $Class[RedefOp_0, \ldots]$. This will mean that when including *Class*, the definition of Op_0 (and possibly other redefined operations) will not be included. The definition of Op_0 in the extended schema will replace the old definition of Op_0 (and will not be conjoined with it). If both renaming and redefinition are applied to a class inclusion, then the renaming is done first and then redefinition is applied. In the following example, we specialize our filesystem $ChFS256$ to a class called $FS + Recovery$ with a new definition of the *Delete* operation: in the new definition, the file is not removed from the file system, but its name is appended with the string ".bak". (If a deleted version of the file already exists, the older version will be permanently removed.) In the class $FS + Recovery$, the old *Delete* operation is called *Remove*.

FS + Recovery

$ChFS[Remove/Delete]$

Delete

$\Delta(fs)$
$name? : \mathbf{seq}Char$
$report! : Msg$

$\exists bn : \mathbf{seq}Char \bullet bn = name? \frown \langle ., b, a, k \rangle$
$(name? \in \mathbf{dom} fs$
$fs' = (fs \setminus \{(bn, fs(bn))\}) \cup \{(bn, fs(name?))\}$
$report! = done) \lor$
$(name? \notin \mathbf{dom} fs$
$fs' = fs$
$report! = error)$

Recover

$\Delta(fs)$
$name? : \mathbf{seq}Char$
$report! : Msg$

$\exists bn : \mathbf{seq}Char \bullet bn = name? \frown \langle ., b, a, k \rangle$
$(name? \notin \mathbf{dom} fs$
$bn? \in \mathbf{dom} fs$
$fs' = fs \cup \{(name?, fs(bn))\}$
$report! = done) \lor$
$((name? \in \mathbf{dom} fs \lor bn? \notin \mathbf{dom} fs)$
$fs' = fs$
$report! = error)$

Exercise 5.6.1 Give an axiomatic definition for the type of *DOSFileNames*. The constraints on this type are as follows.

1. A file name is a sequence of non-control characters;
2. it comprises a name part and an optional extension part separated by the special symbol ‘dot’, which cannot appear elsewhere in the name or the extension parts (if there is no extension then ’dot’ is also omitted);
3. the name part and the extension parts can be, respectively, at most 8 and 3 characters long.

Define an extended (inherited) class from the class *ChFS* that enforces the filenames to be *DOSFileNames*. For each operations, if the user provides a sequence of characters which is not a *DOSFileName*, s/he should get a report stating which constraint has been violated.

Exercise 5.6.2 Consider the class *FS + Recovery*. Extend its definition in such a way that by deleting a file, the older versions of the file that are already deleted will get the extension “.bak 1”, “.bak 2”, etc. Upon recovery, the last version should be recovered and the older version should be shifted from “.bak i” to “.bak i-1”. For simplicity, assume that the set of natural numbers is a subset of characters.

5.6.3 Other Concepts of Class Diagrams

UML comes with a lot of (at times, too much) bells and whistles in its different notational aspects. We believe that what we have described so far, will enable you to have a rigorous understanding of most of the fundamental aspects of class diagrams and thus, we dispense with formalizing the rest of the concepts appearing in UML class diagrams. Some of these concepts are already formalized in Object-**Z** and even

have notational support, others are open to formalization and can be subjects for research activities. The most important aspect of class diagrams, which is used in practice but we did not formalize in this note, is the concept of visibility. Object-**Z** has a standard notation for specifying public interfaces (operations and features) and the way they are treated with respect to inheritance. We refer to [1] for more explanation.

Bibliography

[1] R. Duke and G. Rose. Formal Object-Oriented Specification Using Object-Z. MacMillan, 2000.

[2] C. Fischer. How to Combine Z with Process Algebra. In proceedings of ZUM 98, vol. 1493 of LNCS, pp. 5–23, Springer, 1998.

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

[4] J.M. Spivey. *Understanding Z.* vol. 3 of Cambridge Tracts in Theoretical Computer Science, Cambridge, 1988.

[5] B. Potter, J. Sinclair and D. Till. *An Introduction to Formal Specification and Z.* Prenticec Hall, 1996.

[6] G. Smith. The Object-Z Specification Language. Kluwer Academic Publishers, 2000.

[7] J. Woodcock and J. Davies. *Using Z : specification, refinement, and proof.* Prentice Hall, 1996.