UNIVERSITY OF OSLO
Department of Informatics

Formal Modeling and Analysis of Distributed Systems in Maude

Lecture Notes
INF3230/INF4230

Peter Csaba Ölveczky

April 27, 2008

Dedicated to the memory of my father Miklós Ölveczky and my grandmother Dr. Osztovics Józsefné ("Babkó")

Contents

Chapter 1

Introduction

The following couple of hundred pages comprise the course material for an introductory course in *formal methods*—aimed at third-year students—at the Department of Informatics at the University of Oslo. This course gives a taste of the use of *Maude* [14] for the *formal modeling* of *distributed systems*, and for the computer-aided *formal analysis* of the resulting models.

The Maude system consists of a high-level programming/specification/modeling language based on the theory of *rewriting logic* [69, 6], and of a high-performance interpreter and model checker for this language.

A *distributed system* is a computer system consisting of subsystems which are "distributed" at various locations and which communicate with each other. Examples are networks of computers, multiprocessor systems, etc. A distributed system is typically *concurrent* in the sense that many of its subsystems may perform computations at the same time. A concurrent system is often *nondeterministic*, which means that different executions with the same "input" can give different results.

1.1 Motivation

Society is becoming increasingly dependent on large and complex software systems whose malfunctioning could have serious consequences. Our cars, bank accounts, aircrafts, power plants, etc., are all controlled to a large extent by computer systems. At the same time, the size and complexity of modern software systems make it almost impossible to avoid errors in their specification and implementation.

One example of a distributed system is the network of branches and ATMs ("minibanker") of a certain bank. Another (?) distributed system is the network of many such banks which allows us to withdraw money in downtown Cairo even though your local bank may not even have a branch there. While the failure of such a system to work as expected only results in an embarrassing (collect!) call to your embassy and/or family, a failure in the computer system controlling an aircraft, a nuclear power plant, a medical device such as a brain scanner, a rocket, etc., could cause the loss of human lives. A recent[1] example could be the "smart"

[1] These notes were first written in December 2001.

bomb which killed some U.S. soldiers instead of Afghan villagers and foot soldiers of (U.S.-founded) Taliban.[2]

Apart from more computing power—which means that computer systems can do more and heavier tasks than before—*distribution* and *reactivity* make systems harder to understand because of different "threads" of computation. A system is not just an input/output system as in the era of punch-cards. We will later see that even fairly small distributed systems can be very hard to understand.

In most of the previous programming courses you may have taken so far, the program development process has been fairly straight-forward. Write the program, compile it, remove compile-time errors, recompile it, etc., then run the program on some test input, fix the error(s) you discover, run the program again, etc. When you have run a couple of tests with the desired outcome, you consider the task solved and proudly show your program to your still struggling neighbor.

For distributed systems it is imperative that the problem is very well understood *before* the system is implemented. You do *not* want to implement your aircraft controller system directly on the aircraft and risk that one Airbus A380 crashes for each small programming error.

To understand a complicated system as much as possible, and as early as possible, we need a *model* (or a *prototype*) of the system. The model is a description of the system which can be used to reason about the system and the consequences of different design choices. In many ways, the computer system development challenge nowadays consists of developing the "high-level" design, i.e., the model. When the task at hand is well understood, the actual implementation in, say, a real aircraft is "just" programming and hardware engineering. Or consider the development of a new digital circuit with a million gates. Once we have a complete description of the system and are certain that it works as expected, the job is essentially done. The rest is "just" a matter of putting the gates together in the described way.

In another example underscoring the importance of developing correct system *specifications*, it turned out that of the 197 critical defects identified during integration and system testing of the Voyager and Galileo spacecraft, only three were due to coding errors [91, 63]. Not only are defects most likely to be introduced in the early stage of software development; it is also much cheaper to correct errors early in the software development process.

What do we need to develop and analyze models/specifications/prototypes of non-trivial distributed systems?

- First of all, we need a suitable *modeling formalism*. It should be as "natural" and "intuitive" as possible. An aircraft controller is difficult to understand, so the modeling formalism should *facilitate* and not *complicate* the modeling and analysis effort. Furthermore, the modeling formalism should allow us to *abstract* from unnecessary technicalities and focus on the problem at hand at the *appropriate level of abstraction*. For a communication protocol, we may be interested in what the protocol should do when a packet is lost, but we are probably not interested in exactly *how* a packet is delivered. In the model we abstract from all the details of how a message is really transmitted, and only consider the two cases "the message is delivered" and "the message is lost."

[2] Although not much is publicly known about this incident, it is speculated that it was caused by a computer problem.

Another aspect is that concurrent systems are notoriously difficult to *program*, and that the goal is to have language support that allows you to program concurrent systems as were they "sequential." Likewise it could be desirable to *model* concurrent systems without explicitly having to deal with concurrency.

The naturalness/ease-of-modeling requirement seems to indicate that the traditional programming languages like C, Java, Basic, Pascal, etc., may not be the most well-suited for modeling distributed systems. New domains require new and more powerful tools. Maybe the situation can be compared to, say, construction. In the beginning of human "civilization," the materials for building houses were, say, rock and bricks, and the tools were primitive. Everyone could make his/her own simple house/hut. Furthermore, more or less *any* building task can—given enough time and manpower—be accomplished using these simple materials and tools, as witnessed by masterpieces such as the Bayon of Angkor Thom, the Great Pyramids, Angkor Wat, and the Temple of Ramses in Abu Simbel. However, for today's tasks, these techniques are not *feasible*. Instead, sophisticated (domain-specific) tools and techniques are used to build, say, the Golden Gate Bridge, oil platforms, and the Empire State Building in a reasonable amount of time (and to *move* the Temple in Abu Simbel!). However, not everyone can use the modern engineering techniques and tools without heavy and specialized knowledge of e.g. mathematics.

In the same way, everyone can learn the basics of Java and Basic in short time and can write all kinds of toy programs they need using these languages. Furthermore, by the Church-Turing Thesis, *any* programming task can—given enough time and manpower—be programmed in Basic. However, for today's complicated tasks, such as simulating a nuclear blast, it may not be *feasible* to use Basic. There is a need for new and more powerful (domain-specific) languages and tools to make the development of modern distributed systems feasible. These new and more powerful and suitable tools may require some mathematical insight and may therefore not easily be used by *everyone* without some training.

- Another criterion for the successful development of distributed systems is that it should be possible not only to *model* the system, but also to *analyze/reason* about the model. ("Will the ATM swallow the tourist's credit card if some data packet is lost between Cairo and Urbana?") A first line of analysis could be to read and try to understand the model "manually," but just staring hard at even small pieces of code usually does not convince us that our programs are correct. We like to *execute* our programs. Similarly, we would like to *execute* (or *prototype*) our model to *simulate* the system in various ways.[3] Obviously, the more scenarios we can simulate, the more confident we become in the correctness of the model. In many cases, this is sufficient. (You can become extremely rich by selling computer programs which crash at least once a day for no reason.)

- Although extensive simulation can remove most errors, it cannot *guarantee* the absence of errors because we usually cannot simulate *all* possible behaviors. Furthermore, the unexpected cases which one may have ignored while *modeling* the system are probably

[3]It may be worth mentioning already here that the computer *simulation* of a distributed system *could* be distributed but does not *have to* be distributed. Indeed, it may be easier not to have to worry about simulating on a distributed platform. In this course, our models will be executed on single-processor machines.

also easy to forget when *simulating* the system. In [54], Bjørn Kirkerud wrote that modern fighter jets are more or less completely flown by computers, and that the first time an F-16 crossed the equator, the computer made the plane turn itself upside-down and continued flying upside-down until the pilot intervened. The error was caused by the programmers forgetting that there was something south of Latitude 0.[4] Obviously, this case was then also ignored in the computer simulations. Among other famous examples in computer science folklore are the crash of the rocket Ariane 5 in 1996 which was caused by a programming failure,[5] and the fault in the SRT division in Intel's Pentium processor ("The Pentium Bug") [89].

We therefore need more extensive analysis capabilities in addition to being able to simulate *one* possible behavior of the system from *one* initial state. We could for instance be interested in exploring somehow *all possible* behaviors from *one* initial state, or from *many* different initial states.

Finally, some systems, like nuclear power plant controllers, may require *guarantees* that they are correct for *all possible* behaviors starting from *any* initial state. Providing such guarantees can be quite hard and time-consuming and should therefore only be undertaken on crucial parts of a program, and only after extensive simulation and other kinds of "lighter" analysis has removed most errors.

I guess that not everyone thinking about taking this course intends to go out and design a fighter jet controller, a nuclear power plant controller, a nuclear bomb simulator, a circuit with 500 000 gates, a new worldwide network of ATMs, or even a "smart" bomb right after finishing this course. Nevertheless, there are enough challenges even in small distributed software systems, which can be surprisingly difficult to understand. Later in the course we will see examples of communication and security protocols specified in less than one page of code, but whose flaws took long time to uncover.

1.2 Why Formal Methods?

By *formal methods* we mean languages, techniques, and tools which are based on mathematical logic.

The original motivation behind the use of formal methods in software development was the desire to prove programs correct. Testing and simulation can, as argued above, never guarantee the absence of errors. While that is usually good enough for your homework sorting algorithm or your ubiquitous operating system, it may not be optimal for safety-critical systems such as nuclear power plants. To guarantee the correctness of a program, one would first have to tell what properties the program should satisfy ("The controller should shut down the power plant no more than x seconds after the temperature reaches y degrees."). These *correctness requirements* can be given as logical formulas. Mathematical techniques (such as Hoare logic [17, 47]) can be used to *prove* that the program is correct w.r.t. its requirements.

[4] Some people suggest that this well-known story is just an "urban legend," but it illustrates the point even if it were so.

[5] A software error occurred when a 64 bit `float` was to be converted into a 16 bits signed `int`; the back-up computer system had the exact same problem!

(This process is called *program verification*.) However, just like errors can sneak into programs, the proofs themselves could be wrong, implying that a "guaranteed" correct program may not be correct after all. Therefore, the program verification techniques were formalized as sets of logical rules, and computer programs, so-called *proof checkers*, were developed to *check* whether a *given* proof is correct according to the logical rules. The next step was to use *theorem provers* not just to *check* a "hand made" proof, but to actually *prove* the program correct. However, most interesting properties of computer programs are *undecidable*. This means that there is no computer program which can always prove/decide whether a given program satisfies a given property. In practice, humans must assist the theorem provers in quite sophisticated ways. The human(s) conducting the proofs must therefore have knowledge of logic and of the theorem prover, as well as a deep understanding of the program being analyzed.

Program verification has turned out to be somewhat problematic in practice. It is fairly hard to verify (automatically or "by hand") even quite small programs, a fact which decreases the confidence in the possibility of proving really large and complex programs correct. For this perceived futility of *program verification*, and the perception that *formal methods* is the same as program verification, formal methods have been maligned and regarded as useless in parts of the computer science community.

While the most optimistic early expectations of program verification may not have been fulfilled, research in program verification has yielded many important "side effects," e.g., in the development of programming languages and programming methodology. Furthermore, the usefulness of program verification, in particular for developing safety-critical systems applications, is gradually being realized in industry (see e.g. [46]). For instance, SRI's PVS [84] system could have detected the "Pentium bug" during routine formal verification [89]. Moreover, it is unlikely that standard testing methods would have identified the defect [86]. Other industrial formal verification projects using PVS discovered subtle defects in Rockwell's AAAMP5 microprocessor [92] and the possibility of firing a failed jet in a space shuttle [90]. Some U.S. government agencies now *require* the use of formal methods for validation and specification of safety-critical system components [46].

Model checking [9] is program verification of *finite-state* systems[6] and has been called an "amazing success story." In contrast to "ordinary" program verification, model checking is decidable: Just enter the program and the correctness criteria, and the model checker will automatically tell you whether your program is correct. While pleasant, it does not sound too impressive (in a finite-state system we could just list all states and check all possible behaviors) until we learn that, thanks to sophisticated techniques such as BDD's, model checkers today can handle systems with more than 10^{120} states. Model checkers have been successfully applied to advanced semiconductor and process designs, and to telecommunication and cache coherence protocols such as the IEEE Futurebus+ protocol. Many processor manufacturers now have their own model checkers.

Program verification is now just one part of "formal methods." I believe that formal methods are becoming crucial for the development of current and future software systems as these systems become more complex due to distribution, mobility, real-time behavior, etc. Indeed,

[6] A finite-state system is a system where there are only a finite number of possible program states. For instance, a program where each variable ranges over a finite domain (as opposed to an infinite domain such as the integers) is a finite-state system.

one of the main contributions of formal methods may be the "high-level" modeling, with its *abstraction* from details, that formal methods support, which make it practically possible to model and understand/analyze complex systems. For example, a sophisticated communication protocol is difficult enough to understand and reason about even without having to model all the details of how a single message is actually transmitted. To focus on the design aspects of the protocol, we want, as mentioned above, to abstract from these details and model message transmission abstractly by two operations "message transmitted" and "message lost."

Executable formal languages are powerful yet mathematically well-defined programming languages which provide a high level of abstraction from machine-level details. Such languages may be considered a further refinement in the chain of increasingly abstract (or "high-level") programming languages starting with programming by providing electric currents to transistors, through binary machine languages, assembler programming languages, to languages such as C, and on to, say, Java. In each such "refinement" step the programming language gets further removed from the workings of the computer—thereby we gradually lose control of the machine resources, possibly leading to slower and more memory-consuming program executions—, making it feasible to develop increasingly large and sophisticated programs.

To understand what a program or a specification does, we must know the *meaning* of the programming language constructs. The meaning is called the *semantics*. The semantics of an imperative programming language is essentially given by how it manipulates the computer memory. The reasoning about such programs has the form of following the values of the memory locations, usually by following variables and pointers. This is fairly low-level reasoning which may work for your homework assignment but is probably less suitable for reasoning about complex distributed systems. The semantics of a formal specification is usually given as a well-known mathematical construct such as an *algebra*, a *category*, or a function. It should be much easier to reason about such "high-level" mathematical models than about how the computer memory is updated.

Indeed, the basis of formal methods is to allow us to construct a well-defined *mathematical model* of the system we want to analyze. Once we have such a model, we can use "high-level" mathematical techniques to *reason* stringently about the model. In addition, since the meaning of the model is mathematically well-defined, one may have tools which can analyze the model in various ways.

While I have compared formal methods to imperative languages, there are also "informal" and "semi-formal" specification formalisms such as UML [36] and SDL [51] . Since they are informal and therefore without mathematical semantics, one cannot reason about specifications written in these formalisms, because their meaning is unclear or at least not really defined. Furthermore, without concrete semantics one cannot build tools to analyze such specifications.

To summarize, formal methods provide powerful yet easy-to-understand and well-defined specification constructs to support abstract, high-level, and rigorous modeling and analysis of complex software systems. Furthermore, since the meaning of the model is well-defined, computer tools can provide *automated* analysis facilities in various forms such as execution/prototyping/simulation, theorem proving, and model checking and other kinds of state space exploration.

1.3 Maude

Maude is a formal declarative programming language based on the mathematical theory of *rewriting logic* [69, 6]. Maude and rewriting logic were both developed by José Meseguer and his group at the Computer Science Laboratory at SRI International. (Meseguer is currently working at the University of Illinois at Urbana-Champaign.) Maude is a state-of-the-art formal tool in the fields of *algebraic specification* [98] and modeling of concurrent systems.

The Maude language specifies rewriting logic theories. Data types are defined algebraically by *equations* and the dynamic behavior of a system is defined by *rewrite rules* which describe how a part of the state can change in one step. Maude supports object-oriented programming, including multiple inheritance and asynchronous communication through message passing, in a natural way.

The Maude interpreter executes *equational* Maude programs by starting with a given "initial expression," and applying the equations "from left to right" until no equation can be applied, thereby computing the *normal form* (or "value") of the expression. The interpreter executes *rewrite* programs by "arbitrarily" applying rewrite rules (also "from left to right") on the given initial expression/state, either until no rule can be applied or until a user-given upper bound on the number of rewrites has been reached. (The equations are applied to reduce each intermediate state to its normal form before a rewrite rule is applied.) Note that in this way, the interpreter executes only one of possibly many different behaviors from the initial state.

The Maude team has also focused on performance; the current version of the interpreter can perform millions of rewrites per second, making Maude competitive with general-purpose programming languages in terms of efficiency.

A rewrite theory is often nondeterministic and could exhibit many different behaviors. Maude's *rewrite* command can be used to execute *one* of these behaviors from a given initial state. To analyze *all* possible behaviors *from a given initial state* one can use Maude's high-performance *search* capabilities and/or Maude's built-in *model checker* [32], which is comparable in efficiency with specialized state-of-the-art model checkers. While more powerful than rewriting, search will not terminate if the desired state cannot be reached *and* there is an infinite number of states that can be reached, and model checking rarely works when an infinite number of states can be reached.

A Maude specification can be further analyzed using Maude's reflective and *meta-programming* features [14, 11]. A Maude program can be represented as an "ordinary" term at the "meta-level," where other Maude programs can manipulate this term/program just as they can manipulate other terms such as "2+3". (Don't worry, you are not supposed to understand this right now!) This gives rise to many new meta-programming capabilities, including the possibility that a Maude program (at the meta-level!) defines different *strategies* guiding the execution of another Maude program. In this way Maude provides a flexible execution strategy language where the user can define all the strategies he needs for analyzing his specification. Apart from the flexibility achieved by not being stuck with a hard-wired fixed set of execution strategies, this is a very logically "clean" approach: the execution strategies are also Maude specifications consisting of equations and rewrite rules, and are not "extra-logical" entities.

Finally, Maude provides built-in socket support for network programming, so that we can not only *model* and *simulate and analyze* a distributed system, but actually *program* such systems

in Maude.

> □ The Maude developers live as they preach when it comes to using Maude as a high-level *prototyping language*, in which a *prototype* of a complex software system can be quickly developed, analyzed, and experienced before the heavy "end" product is built. The Maude engine is, as mentioned, a very sophisticated tool which uses all kinds of techniques to improve performance and ease of specification.
>
> Extensive support for object-oriented specification is not yet built-in. Instead, a prototype extension *of* Maude, called Full Maude [28, 13], has been specified *in* Maude. This prototype could be fairly quickly built, analyzed, and modified once people started using it. Now, after much experience has been gained about how to handle object-oriented specification in Maude, will efficient support for such specification be built into the "core" Maude engine using sophisticated C++ and assembly code for optimal performance. The same is the case with parameterized programming and strategies. Let people write their own strategies using Maude's meta-programming features, and only after learning what are the useful and frequently occurring strategies, will optimized versions of them be built into the Maude engine. □

So far, the main applications areas of Maude have been:

- specification and analysis of various security and network communication protocols (see e.g. [19, 18, 82, 59, 44, 83]);
- execution environment in which other formalisms are executed and analyzed (Full Maude [13], Real-Time Maude [80, 81], structural operational semantics [5], the PLAN programming language for active networks [93], ...);
- prototyping of languages (Maude's own extension to deal with strategies, objects, real-time systems, mobile computing [26], ...).
- modeling of cell biology to simulate and analyze biological reactions [31, 30];
- actual programs developed at NASA to determine the position of objects in space;
- discovering many previously unknown address bar and status bar spoofing attacks that can potentially be used in phishing attacks in web browsers [8];
- finding several bugs in embedded software used by major car makers; and
- efficiently analyzing Java programs [35].

The paper [66] provides a more thorough bibliography and roadmap of the use of Maude around the world.

1.3.1 More on Maude and Downloading Maude

The web page `http://maude.cs.uiuc.edu` contains Maude documentation, papers, and (pointers to) almost everything you want to know about the system. The Maude system is available free of charge, from the above web page, for various UNIX/Linux platforms, as well as for Mac (and Windows under Cygwin, see `http://maude.cs.uiuc.edu/download/windows.html`).

Maude is installed at the department computer cluster in Oslo and can be started by giving the command `maude`.

1.4 Why Choose Maude as a Formal Tool?

Why have we chosen Maude as a formal tool for distributed systems? The tool's main advantages are its intuitive high-level modeling formalism and its support for a wide range of validation techniques (including prototyping, state space exploration, temporal logic model checking, and application-specific analysis strategies). Maude occupies a middle ground between model checkers and theorem provers. While model checkers automatically can verify a specification, they can usually do so only on finite-state systems modeled using some restrictive formalism such as, e.g., finite automata[7]. Many sophisticated systems cannot be conveniently modeled in such restricted formalisms. On the other hand, theorem provers have sufficient modeling power but program verification is, as mentioned, difficult and costly and should not be undertaken before "lighter" validation techniques such as prototyping and search have uncovered most errors.

Maude's specification language is quite intuitive. The *static* parts of a system are described by *equations*, which we all know from mathematics ($(x+y)^2 = x^2+2xy+y^2$), physics ($E = mc^2$), or computer science (`fac(n) = if n>1 then n*fac(n-1) else 1`). The *dynamic* parts of a system are described by *rewrite rules*, which are essentially "one-way equations." That's all! There are no tricky constructs for concurrency or communication. This is in contrast to most other specification formalisms for concurrent systems which combine, say, equational specification for the static parts of a system, and an unrelated formalism such as process algebra or Petri nets for describing the dynamic behavior.

Other reasons why Maude may deserve a closer look include:

- Object orientation is an important paradigm which is not well supported by many formal methods.
- The high performance and robustness as well as the fairly intuitive syntax of the tool.
- Very interesting meta-programming capabilities, which also allow for a wide range of simulation and analysis techniques.
- The tool is continuously extended and improved by very good research groups.
- There is a fair amount of experience in using the tool for larger modeling tasks.

Finally, it must be emphasized that Maude is intended to *complement* and not necessarily *replace* other formal methods. Theorem provers should be used for the verification of safety-critical systems after "lighter" validation techniques have uncovered most errors.

1.5 My Own Experience

My largest "practical" Maude application is the sophisticated AER/NCA suite of protocols for multicast in active networks [82]. The AER/NCA protocol suite [53] was at that time

[7] Maude's model checker can be used also for *infinite-state* systems as long as only a finite set of states can be reached from the initial state.

being developed at TASC Inc. and at the University of Massachusetts at Amherst, and deals with the problem of combining *reliability* (each receiver in the multicast group must have received all packets) and *scalability* for multicast in networks where links may lose packets. The AER/NCA protocol suite uses *active networks* where some routers may be able to cache data and execute programs.

The protocol is very complex since it tries to minimize the number of packets sent around to ensure not only reliability, but also efficiency and "TCP-friendliness" by e.g.: dynamically adjusting the sending rate based on the loss rate and at (dynamically selected) "representative" receivers; suppressing any repair of lost packets if a receiver or router believes the packet is "being repaired"; and estimating how long the receiver should wait after a repair request to resend the repair request.

The protocol developers specified the protocol suite using informal UML-like use cases. The informal specification had problems, such as ambiguities and unstated assumptions, some of which were not even clear to the developers. To test their specification, the protocol developers had to implement it on simulation tools and testbeds, in itself a fairly laborious task.

The range of validation techniques provided by Maude were very useful when analyzing this complex protocol suite. Prototyping was used to easily get a feel for the specification—quite often before I knew exactly what properties I needed to check—and to remove errors as early and cheaply as possible. Further simulation of all possible behaviors using Maude's[8] search and model checking facilities gave more information about the specification and uncovered further faults in the protocol. In this way, Maude allowed us to find important errors and inconsistencies in the Maude specification derived from the original use-case informal specification. Remarkably, the tool found *all* errors discovered independently by the protocol developers themselves using test beds and network simulation tools; in addition, our tool uncovered *additional* significant design flaws that were not found by the developers using their traditional tools. It was particularly encouraging that the Maude specification was easily understandable by network protocol developers with no previous experience with formal methods, and that we could cooperate on the development of the protocol suite based on the Maude specification.

1.6 Overview of the Course

One goal of this course is to gently introduce formal methods and a state-of-the-art modeling formalism and analysis tool, and to show different ways in which the tool can be used to analyze distributed systems.

Another goal is to illustrate how mathematical techniques can be used to reason formally about properties of programs. In a couple of weeks you should be able to confidently say "I can *prove* that my program does not contain infinite loops" instead of just saying "I have tried my program on four test cases and it didn't loop, so I *hope* it will terminate also for other inputs."

[8]The protocol—being a real-time protocol—was actually modeled and analyzed using Real-Time Maude [80], an extension of Maude supporting the specification and analysis of *real-time* systems. In particular, Real-Time Maude provides *time-bounded* search and model checking, which makes it possible to analyze also systems with an infinite reachable state space.

A third goal is just to learn something about distributed systems, various forms of communication and their high-level modeling, what properties are of interest in distributed systems, and what are some typical sources of problems.

Finally, you will learn a clean and powerful general-purpose programming language with support for object-oriented programming. Maude's declarative style of programming, its simplicity, and its high-performance interpreter and compiler may make it better suited for certain programming tasks than other languages like Java.

1.6.1 Contents of the Course

As mentioned, a distributed system consists of two parts: The data types defining the state space (even a distributed system uses integers, lists, and other data types) are defined by *equational specifications* in Maude. The dynamic behavior of a system is defined by *rewrite rules* in Maude. This division is also reflected in this course where Part I treats equational specifications and Part II treats rewrite specifications.

Part I: The Static Parts of a System—Data Types

The beginning of Part I deals with the basics of equational specification in Maude: sorts, function declarations, terms, function definition using equations. In Chapter 2 we write and execute many-sorted, order-sorted, and membership equational specifications in Maude. The data types to be specified are the usual ones: natural numbers, integers, lists, binary trees, and multisets. We will define the usual functions on these data types, such as the ubiquitous *quick-sort* function on lists.

This course is supposed to give an introduction into formal methods and how such methods can be used to reason formally about programs. That happens in Chapter 3, which starts by formally defining how Maude computes with equations. To exemplify how to formally reason about specifications and programs, I have chosen to focus on reasoning about *termination*. This chapter provides some intuition and more concrete techniques to *prove* that your specification does not contain an infinite loop for any input. We study the theoretical basis for the concept of *simplification orderings*, and use the standard path orderings to prove termination. In particular, we look at some small examples, such as different versions of the moderately exciting *coffee bean game*, and show how the given mathematical techniques can be used to easily prove that (a version of) the coffee bean game always terminates. This task is surprisingly tricky without the use of such methods.[9]

In Chapter 3 we will also learn how to verify that specifications are *confluent*, that is, that the result of computing an expression is independent of the order in which Maude chooses to apply the equations.

We are not only interested in whether a specification can have infinite loops, but also in the "meaning" of the specification we have defined. While termination ensures that a Maude computation will always give a result, and confluence ensures that we can only have *one* possible result, these properties do not count for much if we have specified, say, addition in

[9]Indeed, in 2002 I promised a bottle of whiskey to the student who could convincingly state whether a version of the coffee bean game always terminates. Nobody got the whiskey.

such a way that computing $2 + 3$ in Maude gives the result 28. Chapter 4 shows how to use *equational logic* to reason about the meaning of a specification. In particular, we focus on how induction techniques can be used to prove that certain desired properties "follow logically" from a specification.

The mathematical meaning of—i.e. the mathematical model defined by—an equational specification is a (class of) algebra(s). Unfortunately, time will *not* allow us to study the algebraic semantics of equational specifications.

Part II: The Dynamic Parts of a System

Part II starts by explaining how equations and *rewrite rules* can be used to specify all possible concurrent computations in a system. Some very simple examples are used, such as the coffee bean game, a fairly short sorting program, and a simulation of the *Super Bowl*.

The next step is to define rewriting logic formally and to understand this definition. We discuss default execution of Maude programs, and check whether Maude's default interpreter is "smart enough" to win the coffee bean game. Will the 49ers beat the Cowboys in Maude's Super Bowl simulation? When we can model distributed systems there is a need to analyze a system beyond just simulating *one* possible behavior of the system. We use Maude's built-in *search* capabilities to reason about *all* possible behaviors of the system which starts with a given initial state.

The chapter thereafter introduces object-oriented modeling of distributed systems. The state of such a system is represented as a multiset of objects and messages. We specify a simple *population* system and the well-known *dining philosophers problem*. In the latter problem there are five philosophers who spend their lives alternating between thinking and eating delicious dumplings. Unfortunately, there are only five chopsticks on the table, one for each philosopher. A philosopher must therefore get her own chopstick *and* borrow the chopstick from her neighbor to the left whenever she is hungry. After eating, she must release the chopsticks (so others can use them) and start to think, until she becomes hungry again. Will any philosopher starve to death due to lack of available chopsticks? Could they *all* starve to death? These, and many other questions, will be answered in this course.

The next task is the high-level modeling of networks and of different forms of communications between the nodes in these networks: "synchronous" communication, asynchronous communication through message passing (with or without links), unicast/multicast/broadcast, etc. We also consider link failures, link creations, and lossy links. These forms and variations of communication will be exemplified by various examples. In particular, we will model and execute the *alternating bit* and *sliding window* communication protocols, which are used to ensure reliable and "correctly-ordered" transmission of a sequence of data packets from a sender to a receiver through an unreliable transmission medium. We will then model and execute the *two-phase commit* protocol for distributed database systems.

Chapter 10 introduces the notions of *invariance* and *liveness* and investigate how the Maude tool can help in the analysis of whether a specification satisfies its requirements. We can therefore use Maude to determine whether the 49ers will *always* beat the Cowboys in the

Super Bowl[10]; whether J.R. and Sue Ellen could possibly separate even though neither of them wants it; whether one—or all—philosopher(s) could starve to death due to lack of two chopsticks; and whether it is possible that a receiver in the alternating bit protocol receives packets out of order.

We are by then ready for some larger case studies. We will use Maude to model and analyze a well-known security protocol (the *Needham-Schroeder public-key authentication protocol* [75]), whose goal is to let Alice and Bob establish a communication between them so that Alice can be sure she's communicating with Bob and not with the malicious intruder Walker. Is the well-known security protocol up to this task, or can Maude show that Walker can impersonate Bob? Part II of this course will tell.

1.7 Prerequisites

For a course in formal methods, this course contains embarrassingly little mathematics, most of which comes quite early in Chapter 3. Apart from some theoretical reasoning about termination in Part I, the course focuses on the *modeling* of systems, and various ways of analyzing systems by executing their models. In that sense, the course may be more like a programming course which uses a mathematics-based programming language.

While it is an advantage to know some logic and the basics of functions and relations (as provided e.g. by an introductory course in discrete mathematics), one should be able to survive this course without much mathematical background. It is desirable that you are confident with simple recursive definitions such as

```
fac(n) = if n>1 then n*fac(n-1) else 1 fi
```

and recursive tree traversal functions, etc. In Oslo, most students should have picked up some "recursive" thinking as hardworking INF2220/INF1020, INF2810 or INF3110 students.

1.8 What to Read and How to Read

The main task in this course is the modeling and analysis—mostly by executing the models in different ways in Maude—of data types and distributed systems. If you look at these course notes, they contain a fair amount of formal definitions of entities like terms, the reduction relation, variable substitutions, etc. While some of these definitions may look non-trivial or even (unnecessarily?) complicated for persons with limited formal methods experience, they most often just state the fairly obvious thing. Your intuition of a notion may well be correct. The formal definitions are mostly there to help you in case you are not sure about something, and to provide a firm foundation upon which a theory can be solidly built. Once you have a good overall understanding you will realize that most details are fairly straight-forward. The Maude technicalities are there to help you to write and test Maude specifications.

[10]The careful reader has of course already noticed that the 49ers and the Cowboys—playing in the same conference—can never meet in the Super Bowl. Unfortunately, this author cannot find a worthy opponent to the 49ers (their recent 5-11 record notwithstanding!) in the AFC.

Important. Some parts of the text are outside the required scope of this course. They may require deeper knowledge or interest, are just added for "fun," or may be needed to really understand some phenomenon. In particular, all footnotes and all

□ indented sections written in grotesque fonts, and delimited by a pair of boxes □

are *not* required reading in this course. Neither is this introduction.

1.9 Supplementary Reading

Although I hope that this compendium is reasonably self-contained, one can always benefit from reading other material as well. The most important supplementary reading material are the Maude book [14] and the Maude manual [13], which describe the Maude system in much more detail than these lecture notes. The Maude primer [67] is a nice, easy, and useful read. It is also helpful to consult the file `prelude.maude` which describes the modules that are automatically imported by Maude upon startup. It is currently available at the CS Department in Oslo as the file `/store/opt/maude-2.3/bin/prelude.maude`.

Part I of the course is concerned with the equational specification of abstract data types. We focus on the operational aspects (i.e. "term rewriting") where Baader and Nipkow's book [2] and (for Norwegian students) Kirkerud's lecture notes [55] are good sources. The recent book [94] treats term rewriting in great depth. We are less concerned with semantic issues of equational specifications, where Wirsing's [98] is a good source and [60] is the standard textbook. We emphasize in this course termination aspects of term rewriting systems, a subject where I found Dershowitz' [21] to be a good introduction.

Part II of the course is concerned with rewriting logic specifications. The theoretical foundations are described in the paper [69], although the parts of that paper which describe the semantics may be hard to read for a beginner. The references [73, 71] are overviews which are easier to read. The paper [12] gives an overview of the Maude project and presents the plans for the coming releases of Maude. The rewriting logic roadmap [66] and this compendium provide further references.

Acknowledgments

I would like to thank Olaf Owe for many helpful comments on earlier versions of these notes and for his friendly support for this project. I am very grateful to José Meseguer for numerous reasons, including some early discussions on this course. Special thanks to my mother Cecilia, to Elisabeth Lien, and to Imtiaz Ahmad and Maiken Blomli—true friends in need and in deed—for lots of support and encouragement while working on these notes. I also thank some INF220 and INF3230 students and TAs, as well as Christophe Malvasio and Alberto Verdejo, for pointing out errors and typos in previous versions of these notes. Finally, I gratefully acknowledge Dag Langmyhr's kind help with LaTeX issues.

Part I

Equational Specifications and Their Analysis

Chapter 2

Equational Specification in Maude

This chapter describes how to specify data types in Maude.

We all know data types such as the integers, various kinds of stacks, lists, ring-buffers, etc., and how to program them in languages like Java. So why should we bother with defining data types in Maude? Apart from being the basis for specifying complex distributed systems in a manageable and abstract way, *declarative* languages such as Maude have some advantages over imperative languages such as Java, including the following:

- It is hopefully easier to understand Maude specifications. Declarative languages do not have pointers/aliasing/side effects which make imperative programs very hard to understand.
- Declarative programs are easier to specify and modify. The constructs are more "powerful," making it easier to specify complicated tasks, and to modify programs, as there are no side effects, etc.
- Specification *is* programming. When I taught sorting algorithms (in Java) in Bergen, a student asked why there is no way of making the computer itself come up with the sorting programs, so that the programmer wouldn't have to worry about all the intricate details of quick-sort, insertion sort, etc. In a way, declarative programming achieves this goal. You *specify* what quick-sort means, and you have the quick-sort program for free. (Therefore, we will make no distinction between a *specification* and a (declarative) *program*, and will use these words interchangeably.)
- Flexibility. In imperative programming languages you have some built-in types, such as the integers, the booleans, characters, and pointers, and then must use classes, structs, etc., to define other types, and you are then often forced to manipulate complex data types indirectly through pointers. In declarative languages, we specify data types "directly."
- You can reason (mathematically) about declarative Maude programs, since such a program specifies a mathematical object. This implies on the one hand that a specification has a clear mathematical meaning (*semantics*). In imperative programs, the meaning of a program is usually given at a low level, by how the program changes the values of the

memory cells in the CPU. Clearly, it is more tricky to reason at such a low level, and it often fairly difficult to know what some constructs *really* do. Furthermore, since a specification is a high-level mathematical object, it can be reasoned about mathematically quite easily by following well-known mathematical rules. For example, one can *prove* properties of programs such as "the power plant will never overheat," "each message will eventually be read by each receiver," and "quick-sort returns a sorted list for *any* input list." Properties like these can never be claimed by just *testing* a program, no matter how extensive the testing (we cannot test quick-sort for all possible lists).

In imperative programs you manipulate the store quite directly through assembly-like "low-level" instructions. The advantage of declarative programs is that you don't have to mess around with such low-level details, but this also means that you lose control over the memory management and of the execution. Declarative programs may therefore use more *space* (memory) and *time* for executing a program than a fine-tuned imperative program. Maude has tried to minimize this disadvantage by a very sophisticated implementation which uses all the tricks in the books—and some more—to save time and space. The Maude interpreter can perform millions of rewrites per second.

2.1 Many-Sorted Equational Specifications

This section defines *many-sorted equational specifications* (see e.g. [98]) and gives some examples of such specifications. In short, a many-sorted equational specification consists of a set of *sorts* (or *sort symbols*), where each sort roughly corresponds to a data type, a set of *function symbols* (also called *operators*)—some of which are used to define the "values" of the data types, and others which are "ordinary" functions on these values—, and equations defining the functions.

In Maude, an equational specification is a *functional module* which is introduced with the following syntax:

```
fmod MODULENAME is
  BODY
endfm
```

where *MODULENAME* is the name of the module being introduced, and *BODY* is a *set* of declarations of sorts, function symbols, variables, and equations. A *comment* starts with `***` or `---` and lasts until the end of the line, or it starts with `***(` or `---(` and lasts until the first matching occurrence of '`)`'.

2.1.1 A First Example: Addition of Natural Numbers in Maude

To get started, we show a first example of a Maude module and how to execute it in Maude. A Maude specification of the data type of natural numbers and an addition function '`+`' may look as follows:

```
fmod NAT-ADD is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .

  vars M N : Nat .

  *** Define the addition function recursively:
  eq 0 + M = M .
  eq s(M) + N = s(M + N) .
endfm
```

We have defined a functional Maude module called `NAT-ADD`. This module has a sort `Nat`. The *expressions* (or *terms*) of this sort `Nat` are `0`, `s(o)`, `s(s(0))`, ..., `0 + 0`, `s(0) + 0`, `s(0) + s(s(0))`, `(s(0) + s(0)) + s(s(0))`, ...

The function symbols `0` and `s` are declared to be *constructors* (`ctor`), and the constructor terms `0`, `s(0)`, `s(s(0))`, `s(s(s(0)))`, ... are the *values* of `Nat`, and intuitively represent, respectively, the numbers 0, 1, 2, 3, ...

The declaration

```
  op _+_ : Nat Nat -> Nat .
```

declares a function symbol `+` which takes two terms of sort `Nat` as arguments, and "returns" a `Nat`-value. The underscore ('`_`') tells where the arguments should be placed in "mix-fix" notation. If there are no underscores (as is the case for `s`), then the function symbol must be used in "pre-fix" notation. If `+` were declared

```
  op + : Nat Nat -> Nat .
```

instead, the terms of sort `Nat` would be `0`,..., `+(0,0)`, `+(s(0), s(s(0)))`, `+(0, +(s(0), 0))`, and `+(s(0), +(s(s(0)), s(0)))`, etc.

After declaring two variables `M` and `N` of sort `Nat`, the module defines the function `+` recursively by giving two equations which says how to compute the addition of any pair of natural numbers.

Getting Started: Running your First Maude Program

Maude uses functional modules to compute the "value" of a given expression. This is done by using the equations "from left to right" in order to "replace equals for equals" until no equation can be used.

To start the Maude system, just give the UNIX command `maude`. Maude will reply with something like

```
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 2.3 built: Feb 14 2007 17:53:50
            Copyright 1997-2007 SRI International
                   Wed Jan 16 12:37:13 2008
Maude>
```

and is ready to handle your command. You now need to enter the module `NAT-ADD` into Maude. This can be done either by typing it directly on Maude's command line (not recommended) or by writing the module in some file, say `nat-add.maude` in the same directory, and then let Maude read this file by using the `in` command:[1]

```
Maude> in nat-add.maude
```

Exercise 1 *Write the above specification in a file, start Maude, and let Maude read the file with the specification.*

If you typed everything as well as I did, you should have gotten an acknowledgment from Maude saying it has read and stored the module `NAT-ADD`:

```
==========================================
fmod NAT-ADD
Maude>
```

If you got some error message(s) you should be aware of the following:

- Maude is case-sensitive. The sorts `Nat` and `nat` are not the same.
- Each declaration should be ended by a space followed by a period ('`.`'). However, there should *not* be a period after comments (these are terminated by end-of-line) or `endfm`.
- For infix symbols such as `+` there should be a space before and after `+` when it is used. The equation should be written `eq 0 + M = M .`, not `eq 0+M = M .`
- There should be no space between '`_`' and '`+`' in the declaration of `+`.

If Maude did not accept your input, change it and try "`in nat-add.maude`" again. To exit Maude, give the command `q` or `quit`. If something is wrong with your input and Maude does not give a Maude prompt, type ctrl-C or ctrl-D.

Once Maude has accepted our specification, we can use Maude to compute the "value" of an expression. If we have forgotten how much is 2+3, Maude can compute 2+3 for us through its `red` (or `reduce`) command as follows:

```
Maude> red s(s(0)) + s(s(s(0))) .
```

[1] The command `load nat-add` does the same thing, but does not print the list of modules entered.

(Note the trailing period.) Maude answers with

```
reduce in NAT-ADD : s(s(0)) + s(s(s(0))) .
rewrites: 3 in 1ms cpu (0ms real) (3000 rewrites/second)
result Nat: s(s(s(s(s(0)))))
```

and the prompt. The last line gives the result `s(s(s(s(s(0)))))` (representing 5) and states that this result has sort `Nat`. If you did not get such a nice result, make sure that your command ended with a space and a period and that there was space around `+`.

Exercise 2 *Use Maude's* `red` *command to compute*

- *2+4 (which is represented by the expression* `s(s(0)) + s(s(s(s(0))))` *in our specification)*
- *0+0*
- *(2+3)+4*

Exercise 3 *Add a multi-line comment to the module* `NAT-ADD` *by using* `***(` *and* `)` *and execute your specification again.*

2.1.2 Importing Previously Defined Modules

An `including`, `extending`, or `protecting`[2] declaration may be used in a module to include (everything except the variable declarations of) modules which are already defined and entered into Maude. For example, if we have already introduced the modules `A` and `B`, then a module `AandB` which imports both `A` and `B` may be written

```
fmod AandB is
  including A .
  including B .
  ...
endfm
```

2.1.3 Sorts

In algebraic specifications—just as in most imperative languages—we use *sorts* to distinguish different kinds of values, such as integers, characters, the Boolean values, and so on. In Maude sorts are declared by the keyword `sort` or `sorts` as in

```
  sort Int .
  sorts Nat Boolean List .
```

where in the latter line we have declared three sorts.

[2]The Maude book/manual describes the difference between `including`, `extending`, and `protecting`. For the moment, the Maude interpreter treats all three in the same way. To be on the mathematically safe side, you can use `including` when in doubt of which one to use.

2.1.4 Function Symbols, Signatures, and Ground Terms

The sorts are just names and do not contain any associated values such as "2" or "5". We use *function symbols* (or *operator symbols*) to define the values (or the "elements") of each sort, *and* to define functions on the domains. In Maude, a declaration of a function symbol has the form

`op` f `:` s_1 `...` s_n `->` s `.`

for $n \geq 0$, where f is the introduced function symbol, and s_1, ..., s_n, and s are sorts. The list[3] $s_1 \ldots s_n$ is called the *arity*, and s the *value sort* (or *coarity*) of f. We may also declare two or more function symbols with the same arity and value sort in one declarations:

`ops` f g h `:` s_1 `...` s_n `->` s `.`

We will use the terms "function symbol", "function", "operator symbol," "operator," and "operation" interchangeably.

Example 1 The module `NAT-ADD` above declares three function symbols. The function symbol `0` has the empty list as its arity and `Nat` as its value sort, the function `s` has arity `Nat` and value sort `Nat`, and the symbol `+` has arity `Nat Nat` and value sort `Nat`. ♠

According to the definition above of function symbol declarations, the arity of a function symbol may be the empty word/list (i.e., $n = 0$), in which case the (corresponding) function is called a *constant*. (Can you think of why such a function is a constant?)

☐ Should we restrict what kind of function symbol declarations are allowed? For example, should we be able to write[4]

```
sorts Int Boolean .
op a : -> Int .
op a : -> Boolean .
```

Quite often such "overloading" is disallowed, since if we just see an `a` we do not know *which* `a` it is, the `Int` or the `Boolean`. However, in Maude, such overloading is allowed. If the sort of `a` cannot be inferred from the context in which it appears, we can write `(a).Int` or `(a).Boolean` to make clear which `a` is used. Allowing overloading may seem unnecessary, but can be quite convenient such as e.g. when the constant `0` is both a `Bit` value, a Boolean value, and a natural number:

```
sorts Bit Boolean Nat .
ops 0 1 : -> Bit .
ops 0 1 : -> Boolean .
op  0 : -> Nat .
```

[3] $s_1 \ldots s_n$ is actually a *word* over the set of sorts.

[4] Even though I do not write the module inside which these declarations appear, all Maude declarations must occur inside a module.

□

The following definition just says that a signature consists of a set of sorts and a set of function symbol declarations:

Definition 1 (Signature) *A* many-sorted signature (S, Σ) *consists of a set S, whose elements are called* sorts*, and an $S^* \times S$-sorted family $\{\Sigma_{w,s} \mid w \in S^*, s \in S\}$ of function symbols. ($\Sigma_{w,s}$ is the set of function symbols with arity w and value sort s.) We often write $f : w \to s \in \Sigma$ for $f \in \Sigma_{w,s}$. If w is the empty word, then f is often called a* constant *(of sort s).*

Example 2 The many-sorted signature $(S_{\texttt{NAT-ADD}}, \Sigma_{\texttt{NAT-ADD}})$ defined by the module `NAT-ADD` has $S_{\texttt{NAT-ADD}} = \{\texttt{Nat}\}$, and has $\Sigma_{\texttt{NAT-ADD}} = \{\Sigma_{\texttt{NAT-ADD}_{w,\texttt{Nat}}} \mid w \in \{\texttt{Nat}\}^*\}$ where $\Sigma_{\texttt{NAT-ADD}_{\epsilon,\texttt{Nat}}} = \{\texttt{0}\}$, $\Sigma_{\texttt{NAT-ADD}_{\texttt{Nat},\texttt{Nat}}} = \{\texttt{s}\}$, $\Sigma_{\texttt{NAT-ADD}_{\texttt{Nat Nat},\texttt{Nat}}} = \{\texttt{+}\}$, and $\Sigma_{\texttt{NAT-ADD}_{w,\texttt{Nat}}} = \emptyset$ for all other w's. (The empty word is denoted by ϵ.) The only constant in this signature is `0`. ♠

The *ground terms* define the "values" we can talk about. Essentially, a ground term is built by constants and other function symbols in a "sort-correct" way:

Definition 2 (Ground terms) *Given a many-sorted signature (S, Σ), we can define the S-sorted set $\mathcal{T}_\Sigma = \{\mathcal{T}_{\Sigma,s} \mid s \in S\}$ of* ground terms *inductively by the following conditions:*

1. *$\Sigma_{\epsilon,s} \subseteq \mathcal{T}_{\Sigma,s}$; that is, every constant of sort s is a ground term of sort s.*
2. *If $f \in \Sigma_{s_1 \ldots s_n, s}$, and $t_1 \in \mathcal{T}_{\Sigma,s_1}$, ..., $t_n \in \mathcal{T}_{\Sigma,s_n}$, and $n \geq 1$, then $f(t_1, \ldots, t_n) \in \mathcal{T}_{\Sigma,s}$. That is, a function symbol "applied" to ground terms of the appropriate sorts gives another ground term.*
3. *In addition, each set $\mathcal{T}_{\Sigma,s}$ is the* smallest *set satisfying the above conditions. That is, only "things" which can be built from constants and the application of function symbols to ground terms of the right sorts are ground terms.*

□ **Notation.** I will sometimes use `type-writer` font and write '`,`', '`(`', and '`)`' instead of ',', '(', and ')'. So that e.g. a term `f(a,b)` will also written `f(a,b)`. □

Example 3 The set $\mathcal{T}_{\Sigma_{\texttt{NAT-ADD}}} = \{\mathcal{T}_{\Sigma_{\texttt{NAT-ADD},\texttt{Nat}}}\}$ of ground terms of sort `Nat` contains the ground terms `0`, `s(0)`, `s(s(0))`, ..., `0 + 0`, `s(0) + 0`, `s(0) + (s(0) + 0)`, ... ♠

Example 4 Given the signature

```
sorts s s' .
ops a b : -> s .
op f : s -> s' .
op g : s s' -> s .
```

Then, `a` and `b` and `g(a,f(b))` and `g(g(a,f(b)),f(a))`, etc., are some ground terms of sort `s`; and `f(a)` and `f(b)` and `f(g(a, f(b)))` are some ground terms of sort `s'`. Neither `a` nor `b` nor `f(a,b)` nor `q(,,...)` is a ground term of sort `s'`. ♠

Exercise 4 *In the signature given in Example 4,*

1. *explain why the terms* `a` *and* `b` *and* `g(a,f(b))` *and* `g(g(a,f(b)),f(a))` *are ground terms of sort* `s`*;*
2. *explain why neither* `f(a)` *nor* `q(,,..)(` *is a ground term of sort* `s`*;*
3. *is* `f(f(a))` *a ground term of sort* `s` *or sort* `s'`*? Explain!*

□ One may sometimes see the following definition of ground terms, which treats constants as ordinary function symbols:

Definition 3 (Ground terms) *Given a many-sorted signature* (S, Σ)*, the* S*-sorted set* $\mathcal{T}_\Sigma = \{\mathcal{T}_{\Sigma,s} \mid s \in S\}$ *of* ground terms *could also be defined inductively by the following conditions:*

1. *If* $f \in \Sigma_{s_1 \ldots s_n, s}$*, and* $t_1 \in \mathcal{T}_{\Sigma,s_1}$*, . . . ,* $t_n \in \mathcal{T}_{\Sigma,s_n}$*, for* $n \geq 0$*, then* $f(t_1, \ldots, t_n) \in \mathcal{T}_{\Sigma,s}$*.*
2. *Each set* $\mathcal{T}_{\Sigma,s}$ *is the* smallest *set satisfying the above condition.*

In addition, we write c *instead of* $c()$ *for any constant symbol* c*.*

Note that this definition applies to all $n \geq 0$, and hence also includes constants, while in our previous definition, it only applied to all $n \geq 1$ and the constant case was treated separately.

We will in this course sometimes assume this "equivalent" definition of ground terms. Whenever you see a definition which talks about all terms of the form $f(t_1, \ldots, t_n)$ for $n \geq 0$, this includes all the constants! □

2.1.5 Examples of Signatures

The Boolean Values.

Let's define a data type of the Boolean values ("true" and "false"). First, we need to define the sort

```
sort Boolean .
```

This is currently an "empty" sort with no values. Obviously, we want its values to be `true` and `false` (or, `1` and `0`; or `0` and `1`). Furthermore, the constants `true` and `false` intuitively "build" the domain of the sort `Boolean`, and are called *constructors* of the sort `Boolean`. In Maude, this information can be given in the specification by adding an *attribute* `ctor` to the function symbol declaration:

```
ops true false : -> Boolean [ctor] .
```

Ground terms built by using only constructors are called *constructor terms*.

□ The `ctor` information serves as a kind of comment and does not influence the Maude computations. However, tools connected to Maude can use this information, for instance when proving inductive theorems about specifications (see Section 4.2). □

Now we have declared the domain/values/elements of the Booleans. The specification is still not too interesting. It is natural to equip the data type with some non-constructor functions. For example, we may want a negation function `not`, a conjunction function `and`, and a disjunction function `or`:

```
op not : Boolean -> Boolean .
ops and or : Boolean Boolean -> Boolean .
```

Some ground terms of sort `Boolean` are now `true` and `false` and `not(true)` and `and(true, true)` and `or(and(true, false), false)`.

Instead of writing `and`(t, u) we could be interested in writing the more natural t `and` u. Maude supports such *mix-fix* notation by allowing underscores ('_') in function symbol declarations. Each underscore corresponds to one argument place, and the number of underscores and the number of sorts in the arity must be the same. There should be no space around '_' in the function symbol declarations. The signature of the Booleans[5] can be given in the following module:

```
fmod BOOLEAN-SIGN1 is
  sort Boolean .
  ops true false : -> Boolean [ctor] .
  op not_ : Boolean -> Boolean .
  ops _and_ _or_ : Boolean Boolean -> Boolean .
endfm
```

Ground terms are now for example `true`, `false`, `not true`, `not false`, `true and false`, `true and true`, `(not true) and false`, `not (true or false)`, and `_and_(false, false)` (that is, prefix notation can still be used, but only with underscores!).

□ One may still need to add parentheses to tell the parser how to parse terms: the term `(not true) and false` is different from `not (true and false)`. In first-order logic there is a *precedence* between the function symbols, where e.g. negation *binds stronger* than conjunction, so that $\neg x \wedge y$ is read $(\neg x) \wedge y$. We can tell the Maude parser to impose a similar precedence on the function symbols by giving a `prec` n attribute in the function symbol declaration, where n is a natural number (including 0). The *lower* the precedence of an operator, the tighter it binds. The Booleans with the standard precedence may look like

```
fmod BOOLEAN-SIGN-PREC1 is
  sort Boolean .
  ops true false : -> Boolean [ctor] .
  op not_ : Boolean -> Boolean [prec 53] .
  op _and_ : Boolean Boolean -> Boolean [prec 55] .
  op _or_ : Boolean Boolean -> Boolean [prec 59] .
endfm
```

[5] You should give the Maude command `set protect BOOL off .` before entering your version of the Booleans. This turns off Maude's version of the Booleans.

The precedence numbers are "arbitrary." What matters is their relationship: instead of 53, 55, and 59 we could have chosen 1, 2, and 3. A term `true and not true or false` is now read `(true and (not true)) or false`. (Why?)

There are a lot of other parsing issues. Should `true and false and true` be read `(true and false) and true` or as `true and (false and true)`? To read how Maude treats these and other parsing issues, the interested reader is referred to [13, Chapter 3].

Exercise 5

1. *Parse (i.e., insert parentheses to show "how the term is read")*

   `true and not false or not false and (true or true)`

2. *Parse* `true and not true or false` *in a signature in which* `not` *has precedence 18,* `and` *has precedence 5, and* `or` *has precedence 1.*

□

Natural Numbers

Let us now define a signature for the data type of natural numbers (which includes 0). As usual we need to declare constructors to represent the desired domain (0, 1, 2, ...), and declare some non-constructor functions such as `+` and `*`. The first problem is how to represent the natural numbers. Clearly, having a constant for each number as in

```
sort Nat .
ops 0 1 2 3 4 5 ... 1000 : -> Nat [ctor] .
```

is both cumbersome to write (you have to explicitly write 1001 numbers), does not represent all natural numbers, and has other undesired properties. Instead, the usual way to represent natural numbers is to use the constructors

```
fmod NATURALS-SIGN is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  ...
endfm
```

where `0` represents the number 0, and `s`(n) represents the *successor* of n, that is, $n + 1$. So, the numbers 0, 1, 2, 3, ... are represented by, respectively, the constructor terms `0`, `s(0)`, `s(s(0))`, `s(s(s(0)))`, ...[6]

Other function symbols of interest could be

```
ops _+_ _*_ : Nat Nat -> Nat .
```

[6] As we will see in Section 2.7.2, Maude contains for efficiency and ease-of-specification purposes built-in data types (implemented in C++) for the natural numbers and the integers, where the term `2008` can be used instead of `s(s(...s(0)...))`, and where the functions are efficiently implemented.

Exercise 6

1. *Declare other function symbols you think naturally belongs to a data type of natural numbers.*
2. *Can you modify the above signature so that parentheses-less expressions involving* `+` *and* `*` *are parsed correctly, i.e.,* `a * b + c * d` *is parsed* `(a * b) + (c * d)` *for constructor terms* a, b, c, d*?*
3. *Show that each natural number corresponds to exactly one ground term constructed by* `0` *and* `s`*, and vice versa.*

Lists

How do we represent lists of, say, natural numbers, using many-sorted equational specification? The straightforward solution is to start with a constructor `nil` for the empty list:

```
fmod LIST-NAT1-SIGN is
  including NATURALS-SIGN .
  sort List .
  op nil : -> List [ctor] .
```

and then use a constructor `app` which appends an element to the end of a list:

```
  op app : List Nat -> List [ctor] .
```

In this case, a list "1 2 3" is represented by the constructor term

```
  app(app(app(nil, s(0)), s(s(0))), s(s(s(0)))).
```

Using mix-fix notation, we may for example use a function symbol

```
  op _++_ : List Nat -> List [ctor] .
```

instead of `app`. Then, the list "1 2 3" can be written `nil ++ s(0) ++ s(s(0)) ++ s(s(s(0)))` which is probably more readable. Indeed, we can make it more like the mathematical notation by using mix-fix *empty* syntax. That is, instead of `_++_` we can use the function symbol `__`:

```
  op __ : List Nat -> List [ctor] .
```

Then, the list "1 2 3" can be represented by the term `nil s(0) s(s(0)) s(s(s(0)))`.

Exercise 7 *Using the above constructors* `nil` *and* `__`*, show that parentheses are not needed. That is, expressions such as* `nil s(0) s(s(0)) s(s(s(0)))` *can only be parsed in one way. Show it on the above term.*

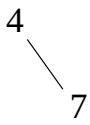

Figure 2.1: A (small) binary tree.

Some non-constructor functions on lists could be

```
op length : List -> Nat .           *** Number of element in a list
op concat : List List -> List .     *** Concatenate two lists
op insertFront : Nat List -> List . *** Insert element first in the list
ops first last : List -> Nat .      *** first/last element in a list
op rest : List -> List .            *** The list with first element removed
op empty? : List -> Boolean .       *** Is the list empty?
op reverse : List -> List .         *** Reverse the list
```

Exercise 8

1. *What other functions do you think a data type of lists of natural numbers should contain?*
2. *We have not yet* defined*, only* declared *the functions* `length`*,* `concat`*,* `first`*, ... What do you think the values of the following expressions should be?*
   - `length(nil s(0) s(s(0)) s(s(s(0))))`
   - `first(nil s(0) s(s(0)) s(s(s(0))))`
   - `empty?(nil s(0) s(s(0)) s(s(s(0))))`
   - `concat(nil s(0) s(s(0)), nil 0 s(s(s(0))))`
   - `reverse(nil s(0) s(s(0)) s(s(s(0))))`

Binary Trees

A binary tree where the nodes are (labeled with) natural numbers can be represented by the following constructors:

```
sort BinTree .
op niltree : -> BinTree [ctor] .
op btree : BinTree Nat BinTree -> BinTree [ctor] .
```

where `btree`(t, n, t') represents the tree with root labeled n which has t as its left subtree and t' as its right subtree. For example, the tree in Fig. 2.1 can be represented by the term

```
btree(niltree, s(s(s(s(0)))), btree(niltree, s(s(s(s(s(s(s(0))))))), niltree))
```

Exercise 9 *Represent the tree given in Fig. 2.2 using the constructors given above.*

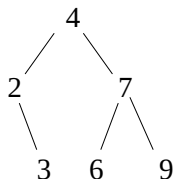

Figure 2.2: Another binary tree.

Other function symbols in this data type could include

```
ops preorder inorder postorder : BinTree -> List .
```

which are intended to list the elements of a tree in the order the elements are encountered in, respectively, preorder, inorder, and postorder traversal of the tree. Other function symbols could be

```
ops size weight : BinTree -> Nat .
op isSearchTree : BinTree -> Boolean .
op reverse : BinTree -> BinTree .
```

(Of course, all the data types for Booleans, natural numbers, and lists must be imported into a module containing the above declarations.)

Exercise 10 *Declare other function symbols which would be natural to include in a data type of binary trees.*

Sets

We all know what a set of natural numbers is. How do we represent one in a many-sorted specification?

The standard approach could be to construct all sets from the empty set, and some constructor such as `_;_` or `_,_` or `_U_` or . . . (pick your favorite symbol). In this example, we use `_;_` instead of the often used `__` to avoid confusion with lists:

```
sort Set .
op empty : -> Set [ctor] .
op _;_ : Set Nat -> Set [ctor] .
```

A set $\{2, 4, 5\}$ can be represented by the constructor term

```
empty ; s(s(0)) ; s(s(s(s(0)))) ; s(s(s(s(s(0)))))
```

In contrast to the previous examples, where there was a one-to-one correspondence between the "values" in the domain we specify and the set of constructor terms, the set $\{2, 4, 5\}$ is the same as the set $\{4, 2, 2, 5\}$ and can therefore also be represented by the constructor term

```
empty ; s(s(s(s(0)))) ; s(s(0)) ; s(s(0)) ; s(s(s(s(0))))
```

Exercise 11 *Declare some function symbols you think are useful in the data type* `Set`*.*

Multisets

A *multiset* over a set S is a "set" of S-elements where the number of occurrences of each element matters. That is, while the *sets* $\{a, b\}$ and $\{a, a, b\}$ are the same, the *multisets*[7] $\{a, b\}$ and $\{a, a, b\}$ are different.

> □ Mathematically, a multiset m over a set S is a function $m : S \to \mathbf{N}$ where $m(s)$ denotes the multiplicity (the number of occurrences) of the element s in m. A *finite multiset* is a multiset m whose *support* $\{s \mid s \in S \wedge m(s) > 0\}$ is a finite set. □

Multisets (of e.g. natural numbers) can, like sets, be constructed from the empty multiset, call it `none`, and some "add"-operator:

```
sort Multiset .
op none : -> Multiset [ctor] .
op __ : Multiset Nat -> Multiset [ctor] .
```

(If you don't want to overuse the convenient `__` operator for multiset union, you may of course use other convenient symbols such as `_;_` or `_&_` instead.)

Exercise 12 *Declare a many-sorted signature for the data type of stacks of natural numbers.*

2.1.6 Variables, Terms, and Equations

We have presented the many-sorted signatures of various data types. Still, these specifications are not very exciting. For example, `s(0) + s(0)` and `s(s(0))` are two different ground terms with no relationship between them. Obviously, it would be more interesting to "compute the value of" `s(0) + s(0)`. In this section we show how to *define* the non-constructor functions using (unconditional and conditional) equations.

First, we need the notion of *variables*. Essentially, we have variables of different sorts:

> □
>
> **Definition 4 (Variables)** *Given a many-sorted signature (S, Σ), a variable set X is an S-sorted family $X = \{X_s \mid s \in S\}$ of pairwise disjoint sets (that is, no variable has two different sorts: $s \neq s' \Longrightarrow X_s \cap X_{s'} = \emptyset$), also disjoint from Σ (that is, nothing can be both a variable and a function symbol). We will often write $x : s$ for $x \in X_s$.*
>
> □

In Maude, the keywords `var` and `vars` are used to declare variables as illustrated below:

[7] We use the same brackets/braces to denote a multiset as we use for sets. Hopefully this is not too confusing.

```
var X : Boolean .
vars M N : Nat .
```

Actually, variables need not be explicitly declared. Instead, variables of the form *var*:*sort* can be used directly, so that the two code segments

```
vars M N : Nat .
eq 0 + M = M .
eq s(M) + N = s(M + N) .
```

and

```
eq 0 + M:Nat = M:Nat .
eq s(M:Nat) + N:Nat = s(M:Nat + N:Nat) .
```

are equivalent.

We have previously defined the notion of *ground* term. ("Non-ground") terms can contain variables of appropriate sorts. More formally: The set $\mathcal{T}_\Sigma(X)$ of *terms* in a signature (S, Σ) w.r.t. a set of variables X are all the "things" that can be built in a sort-consistent way from constants, variables, and the application of function symbols:

Definition 5 (Terms) *Given a many-sorted signature (S, Σ) and a variable set $X = \{X_s \mid s \in S\}$, the S-sorted set of terms $\mathcal{T}_\Sigma(X) = \{\mathcal{T}_{\Sigma,s}(X) \mid s \in S\}$ is defined inductively by the following conditions:*

1. *$X_s \subseteq \mathcal{T}_{\Sigma,s}(X)$ for $s \in S$; that is, a variable of sort s is also a term of sort s.*
2. *$\Sigma_{\epsilon,s} \subseteq \mathcal{T}_{\Sigma,s}(X)$ for $s \in S$; that is, a constant of sort s is also a term of sort s.*
3. *$f(t_1, \ldots, t_n) \in \mathcal{T}_{\Sigma,s}(X)$ if $f \in \Sigma_{s_1 \ldots s_n, s}$ and $t_i \in \mathcal{T}_{\Sigma,s_i}(X)$ for each $1 \leq i \leq n$.*
4. *$\mathcal{T}_\Sigma(X)$ is the smallest S-sorted set satisfying the above conditions.*

Now we are ready to define the functions declared in the signature. These functions are defined recursively by equations and conditional equations:

Definition 6 (Equations) *Given a many-sorted signature (S, Σ), a (Σ-)* equation *is a triple (X, t, t'), written $(\forall X)\ t = t'$, where X is an S-sorted variable set disjoint from Σ, and t and t' are terms of the same sort; i.e., $t, t' \in \mathcal{T}_{\Sigma,s}(X)$ for some $s \in S$.*

A conditional (Σ-) equation *is a $2(n+1)+1$-tuple $(X, u_1, v_1, \ldots, u_n, v_n, t, t')$ for $n \geq 1$, written*

$$(\forall X)\ u_1 = v_1 \wedge \ldots \wedge u_n = v_n \Longrightarrow t = t',$$

such that there are sorts $s_1, \ldots, s_n, s$ in S with $t, t' \in \mathcal{T}_{\Sigma,s}(X)$ and $u_i, v_i \in \mathcal{T}_{\Sigma,s_i}(X)$ for each $i \in \{1, \ldots, n\}$.

Definition 7 (Many-sorted equational specifications) *A* many-sorted equational specification *is a tuple* (S, Σ, E) *where* (S, Σ) *is a many-sorted signature and* E *is a set of* Σ*-equations and conditional* Σ*-equations.*

In Maude, equations are written with syntax

```
eq t = t' .
```

and conditional equations are written with syntax

```
ceq t = t' if u1 = v1 /\ ... /\ u1 = v1 .
```

Example 5 As an example of a many-sorted specification (with only one sort!) we recall our Maude specification of the natural numbers with addition:

```
fmod NAT-ADD is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .

  vars M N : Nat .

  eq 0 + M = M .
  eq s(M) + N = s(M + N) .
endfm
```

♠

The *meaning* of equations and how they are used will be precisely explained in Chapter 3. Informally, the "mathematical" meaning of an equation $(\forall X)\ t = t'$ is that t and t' should be considered equivalent for all "values" of the variables X. For example, the equation $(\forall \texttt{M} : \texttt{Nat})\ \texttt{0 + M} = \texttt{M}$ means that $\texttt{0 + 0} = \texttt{0}$, $\texttt{0 + s(0)} = \texttt{s(0)}$, and so on. Similarly, $(\forall X)\ u_1 = v_1 \wedge \ldots \wedge u_n = v_n \Longrightarrow t = t'$ means that if $u_1 = v_1$ and $\ldots$ and $u_n = v_n$ for some values of the variables in X, then t should be equal to t' for those same values of the variables.

The *operational* meaning describes how Maude computes with equations. In Maude, the `red` command is used to compute the "value" of a ground term. For example, after introducing the above module `NAT-ADD` to the Maude system, we can ask Maude to compute the "value" of a ground term such as e.g., `s(s(0 + s(0))) + 0` by giving the command

```
red s(s(0 + s(0))) + 0 .
```

Then, the following happens in the Maude system:

1. The system checks whether *some* equation can be applied *somewhere* in the term. That is, it checks whether the *lefthand* side of an equation "matches" (or "fits") the term somewhere. It then applies the equation by "replacing equal by equal," that is, by transforming the "fitting part" into the corresponding righthand side. In the example above, the equation `0 + M = M` could be applied on the term `s(s(0 + s(0))) + 0`, reducing it to `s(s(s(0))) + 0`. If more than one equation can be applied, and/or if an equation can be applied in more than one place in a term, then the system chooses *(pseudo-) arbitrarily* what equation to apply and where to apply it. For example, in addition to the previous application, the equation `s(M) + N = s(M + N)` could be applied on `s(s(0 + s(0))) + 0`, giving `s(s(0 + s(0)) + 0)`.

2. The above process is iterated on the resulting term as long as there is some equation which can be applied.

3. When no equation can be applied anywhere, the "current" term is output as the result.

Example 6 We see that `s(s(0 + s(0))) + 0` could reduce to `s(s(s(0))) + 0` in the module `NAT-ADD`. In the next step, only the equation `s(M) + N = s(M + N)` can be applied, giving `s(s(s(0)) + 0)`. In the next step, only this same equation can be applied, giving `s(s(s(0) + 0))`. In the next step, this same equation can be applied, giving `s(s(s(0 + 0)))`. Now, only the equation `0 + M = M` can be applied, giving the term `s(s(s(0)))`. Now, no more equation can be applied, and therefore the result is `s(s(s(0)))`, which is a constructor term, and which is the result output by Maude:

```
result Nat : s(s(s(0)))
```

The sequence $\texttt{s(s(0 + s(0))) + 0} \rightsquigarrow \texttt{s(s(s(0))) + 0} \rightsquigarrow \texttt{s(s(s(0)) + 0)} \rightsquigarrow \ldots \rightsquigarrow \texttt{s(s(s(0)))}$ is called a *derivation*, a *computation*, or a *reduction sequence*. ♠

Exercise 13

1. *Retrace the derivation given in Example 6. That is, show exactly how and where the equations are applied in each step.*

2. *Show a derivation from* `s(s(0 + s(0))) + 0` *if the equation* `s(M) + N = s(M + N)` *is applied in the first step.*

2.2 Operational Properties of Specifications

As described above, Maude computes with equational specifications by taking a ground term and "arbitrarily" applying equations from left to right until no equation can be applied. There are intuitively some desired properties the equational specification should satisfy, so that this form of computation is meaningful. How to mathematically state these properties and how to formally check/prove them is treated in Chapter 3. This section therefore just intends to explain these properties intuitively.

2.2.1 Termination (No Infinite Looping)

For any ground term we want its computation to *terminate*, that is, we don't want any infinite derivations. For example, in the module `NAT-ADD`, no matter how the equations to apply are chosen, each computation would always end up with a term to which no equation applies.

Exercise 14 *Argue informally that there can be no infinite computation in the module* `NAT-ADD`*.*

However, in a specification

```
sort s .
ops a b : -> s .
eq a = b .
eq b = a .
```

the system would reduce `a` to `b` using the first equation, and then `b` would be reduced to `a` using the second equation, and then `a` would again be reduced to `b` using the first equation, and so on, giving an infinite computation

$$\texttt{a} \rightsquigarrow \texttt{b} \rightsquigarrow \texttt{a} \rightsquigarrow \texttt{b} \rightsquigarrow \cdots$$

starting from `a`. Similarly, if we add the equation

```
eq M + N = N + M .
```

(where `M` and `N` are variables) to our specification of the natural numbers, then we could get infinite computations.

Exercise 15 *Show an example of an infinite computation in the module* `NAT-ADD` *extended with the above commutativity equation.*

A specification is called *terminating* if it does not allow any infinite computation.

2.2.2 Uniqueness of the "Result"

In a terminating system, we would like to get the *same* result, no matter *how* we apply the equations (since we are computing the values of function applications, which should be deterministic). For example, any computation of `s(s(0 + s(0))) + 0` should always end with the result `s(s(s(0)))`, not `s(s(s(0))) + 0` or `s(0)` or anything else. (Since we have no control over the application of equations, it would be unsatisfactory if the result of computing a term would depend on how the Maude systems internally chooses which equations to apply.)

Example 7 In the following terminating specification

```
sort s .
ops a b c : -> s .
eq a = b .
eq a = c .
```

the term `a` does *not* have a unique result, since it can be computed to both `b` and `c`. ♠

A result of a computation of a term t is called a *normal form* of t. If it is in addition unique, then this unique normal form is written $t!$. For example, the normal form of `s(s(0 + s(0))) + 0` is `s(s(s(0)))` in the module `NAT-ADD`.

2.2.3 The Result Should be a Constructor Term

As we have seen, the set of function symbols is divided into *constructors*, which define the set of "values" of a data type (such as $\{\texttt{0}, \texttt{s}\}$ defining the set of natural numbers as terms `0`, `s(0)`, `s(s(0))`, ...) and non-constructor, or *defined*, function symbols such as `+`. As we want to compute the value of a term, it is obviously desirable that *each ground term is reducible to some constructor term.* For example, if we forget the equation `0 + M = M` from the specification of `NAT-ADD`, then `s(s(0 + s(0))) + 0` reduces to `s(s((0 + s(0)) + 0))`, which cannot be further reduced, and which is not the result we really wanted.

An important part of this criterion is that a defined function is "defined" on *all* constructor ground terms. For instance, for natural numbers, n_1 `+` n_2 is defined for all values/constructor ground terms n_1 and n_2, since n_1 (and n_2 as well for that matter) either should be of the form `0` or `s`(n) for some n. In the first case, the equation `0 + M = M` will apply, and in the second case `s(M) + N = s(M +N)` can be applied.

Functions are often defined by having one equation for each constructor, although sometimes we need fewer or more equations:

```
op double : Nat -> Nat .
var N : Nat .
eq double(N) = N + N .
```

Obviously, the above equation covers all arguments of `double`. A functions `minusTwo` which removes two from any number greater than one can most easily be defined by three equations:

```
op minusTwo : Nat -> Nat .
var N : Nat .
eq minusTwo(0) = 0 .
eq minusTwo(s(0)) = 0 .
eq minusTwo(s(s(N))) = N .
```

For any constructor ground term n, some equation can be applied on `minusTwo`(n). Always make sure that your functions are defined for all constructor ground terms!

2.3 Examples of Many-Sorted Specifications

In this section we present the many-sorted specifications of some common data types whose signatures were given above. In particular, we should try to ensure that our specifications have all the nice properties mentioned above. The specifications should be terminating and the result of a Maude computation should be a unique constructor term.

2.3.1 The Natural Numbers with Multiplication

The following module `NAT-MULT` specifies the natural numbers with addition and multiplication operations. If you don't mind parentheses, you don't need to worry about operator precedence in this course.

> □ I have chosen to add a precedence for + and * so that an expression like n_1 + n_2 * n_3 is read as n_1 + (n_2 * n_3). As mentioned, this is achieved by letting * have a *lower* precedence number than +. (The actual values don't matter: I could have chosen 2 and 1 instead of 33 and 31.) □

```
fmod NAT-MULT is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat [prec 33] .
  op _*_ : Nat Nat -> Nat [prec 31] .

  vars M N : Nat .

  eq 0 + M = M .
  eq s(M) + N = s(M + N) .
  eq 0 * M = 0 .
  eq s(M) * N = N + M * N .
endfm
```

Exercise 16

1. *Test the above specification* `NAT-MULT` *in Maude by computing the value of the term* `s(s(s(0))) * s(s(0)) + s(0) * s(s(s(s(0))))` *using Maude's* `red` *command.*

2. *(Important! Try to get this one right. A simple straight-forward solution uses three equations to define* `monus`*.) Define a Maude module* `NAT-MONUS` *which imports the module* `NAT-MULT` *and defines a* `monus` *operation, so that* m `monus` n *is* $m - n$ *when* $m \geq n$ *and is 0 otherwise. Test your specification in Maude.*

2.3.2 The Boolean Values

The data type of Boolean values and functions can likewise be specified as follows:

```
fmod BOOLEAN is
  sort Boolean .
  ops true false : -> Boolean [ctor] .
  op not_ : Boolean -> Boolean [prec 53] .
  op _and_ : Boolean Boolean -> Boolean [prec 55] .
  op _or_ : Boolean Boolean -> Boolean [prec 59] .
  op _implies_ : Boolean Boolean -> Boolean [prec 61] .

  var X : Boolean .
  eq true and X = X .
  eq false and X = false .
  ...
endfm
```

Exercise 17 *Since there are also built-in Boolean values in Maude, you should give the Maude command* `set protect BOOL off .` *before entering the specifications below into Maude.*

1. *Continue the specification of the module* `BOOLEAN` *by defining the functions* `not`*,* `or`*, and* `implies`*. (Remember that* `not true` *is* `false` *and vice versa.* x `or` y *is* `true` *if at least one of* x, y *is* `true`*, and* x `implies` y *is* `false` *only when* x *is* `true` *and* y *is* `false`*.)*
2. *Test your specification using Maude. What is e.g.,* `true or (not true) and true`*?*
3. *Declare and define an operator* `if_then_else_fi` *(without using* `not`*,* `and`*, etc.). Then you should be able to redefine* `and`*,* `not`*,* `or`*, and* `implies` *in one equation each using* `if_then_else_fi`*. Do that and test your new specification in Maude.*
4. *Declare and define a function* `equals` *which denotes Boolean equality.*

2.3.3 The Natural Numbers Revisited

We now define the module `NAT1` by extending the module `NAT-MONUS` (which in turn is an extension of `NAT-MULT` with a `monus` function) to include some comparison operators, including `==` for equality, and functions for finding the maximum and minimum of two numbers[8]:

```
fmod NAT1 is
  protecting NAT-MONUS .
  protecting BOOLEAN .

  ops _<=_ _<_ _>=_ _>_ _==_ : Nat Nat -> Boolean .
  ops max min : Nat Nat -> Nat .    *** The largest/smallest of two numbers
```

[8]The definitions of `max` and `min` are not too readable. You may want to define an `if-then-else-fi` operator on the natural numbers, and then define `max` and `min` naturally.

```
  vars M N : Nat .
  eq 0 < s(M) = true .
  eq M < 0 = false .
  eq s(M) < s(N) = M < N .
  eq s(M) == 0 = false .
  eq 0 == s(M) = false .
  eq 0 == 0 = true .
  eq s(M) == s(N) = M == N .
  eq max(M, N) = (M monus N) + N .
  eq min(M, N) = (M + N) monus max(M, N) .
  ...
endfm
```

Equality may of course also be defined in terms of inequality:

```
  eq M == N = not ((M < N) or (N < M)) .
```

I don't know which solution is more elegant. The second solution seems less efficient since it computes two inequalities.

Exercise 18 *Define the other comparison operators in the above module* `NAT1`*. Save your specification as you will need it later.*

Exercise 19 *Define in your module* `NAT1` *an operator*

```
  op diff : Nat Nat -> Nat .
```

which computes the difference *between two natural numbers. (The difference between 2 and 8 is 6, and the difference between 4 and 1 is 3.)*

2.3.4 Lists

Recall the signature of the data type of lists of natural numbers in Section 2.1.5. In particular, the list constructors are `nil` for the empty list, and the append operator `__` which appends a natural number to a list. The functions on lists should therefore be defined on these two constructors, as in

```
fmod LIST-NAT1 is
  including LIST-NAT1-SIGN .

  vars N N' : Nat .
  vars L L' : List .

  eq length(nil) = 0 .
```

```
  eq length(L N) = s(length(L)) .
  eq concat(L, nil) = L .
  eq concat(L, L' N) = concat(L, L') N .
  eq first(nil) = 0 .                  *** Default/error value
  eq first(nil N) = N .
  eq first(L N N') = first(L N) .
  ...
endfm
```

Exercise 20 *Define the other functions on the data type* `List` *whose signature* `LIST-NAT1-SIGN` *is given in Section 2.1.5.*

Exercise 21 *Lists of natural numbers can be compared* lexicographically*, like the ordering in a dictionary or a phone book. A list l is greater than a list l' if there is a number i such that*

- *the ith element of l exists, and it is greater than the ith element in l' or the ith element in l' does not exists; and*
- *for all $j < i$, the jth element of l is the same as the jth element of l'.*

In short, l is greater than l' if both lists are the same until either l' stops or until an element in l is greater than the corresponding element in l'. For example, the list "4 5 6" is greater than both "3 4 5 6 7", "4 5", and "4 5 2 10".

1. *(Slightly tricky?) Show (by an example) that there is an* infinite *sequence*

   $$l_0 > l_1 > l_2 > l_3 > \dots$$

   *of lists $l_0, l_1, l_2, l_3, \dots$ such that l_i is greater than l_{i+1} for all i. (*Hint: *Sometimes the length of the lists must increase when the list gets "smaller" for this to work.)*

2. *Try to informally and intuitively explain why there is* no *infinite sequence*

   $$l_0 > l_1 > l_2 > l_3 > \dots$$

   of lists $l_0, l_1, l_2, l_3, \dots$ of the same length *such that l_i is greater than l_{i+1} for all i.*

3. *Define in Maude a function*

   ```
   op _greaterThan_ : List List -> Boolean .
   ```

   which compares two lists lexicographically, and test your definition by running some test examples in Maude.

2.3.5 Binary Trees

In this section we define the following data type of binary trees of natural numbers:

```
fmod BINTREE-NAT1 is
  protecting BOOLEAN .
  protecting NAT1 .
  protecting LIST-NAT1 .

  sort BinTree .
  op niltree : -> BinTree [ctor] .
  op bintree : BinTree Nat BinTree -> BinTree [ctor] .
  ops preorder inorder postorder : BinTree -> List .  *** Traverse the tree
  ops size weight : BinTree -> Nat .
  op isSearchTree : BinTree -> Boolean .
  op reverse : BinTree -> BinTree .

  vars BT BT' : BinTree .
  vars N N' : Nat .
  eq preorder(niltree) = nil .
  eq preorder(bintree(BT, N, BT')) =
        insertFront(N,          *** Root first, then left and right subtrees:
                    concat(preorder(BT), preorder(BT'))) .
  eq size(niltree) = 0 .
  eq size(bintree(BT, N, BT')) = s(size(BT) + size(BT')) .
  ...
endfm
```

Exercise 22 *Define the other functions in the module* `BINTREE-NAT1`*. The functions* `inorder` *and* `postorder` *should return a list of the elements of the tree encountered when the tree is traversed in an inorder (resp. postorder) manner.* `weight` *gives the sum of the elements in the tree, and* `isSearchTree` *returns* `true` *if and only if the tree is a binary* search *tree; that is, an inorder traversal ("from left to right") encounters the elements in increasing (or at least non-decreasing) order). The function* `reverse` *should reverse the tree, i.e. "flip it" around its vertical axis. Test your specification using Maude. (Don't forget to import your previous specifications of the Booleans, the natural numbers, and lists.)*

2.3.6 Multisets

Remember that a multiset is a "set" where an element may occur more than once.

> □ A multiset (over some ordered set) A is greater than a multiset B if and only if you can obtain B from A by removing $n \geq 1$ elements from A, and replacing each removed element with a (possibly empty) multiset of smaller elements. □

For multisets over totally ordered sets like the natural numbers, the multiset with the largest element is also the largest multiset. If both multisets have the same greatest element, then

remove this greatest element *once* from each multiset and compare the remaining multisets. Any non-empty multiset is greater than the empty multiset.

Example 8

- The multiset $\{2, 2, 1\}$ is greater than $\{1, 2\}$ since we can in the first step remove one occurrence of the greatest element "2" from each multiset; we are then left with $\{2, 1\}$ and $\{1\}$ where the former is greater since it has the largest element (2).[9]
- The multiset $\{3, 8\}$ is greater than the multiset $\{7, 6, 5, 7, 5, 5, 6, 0, 1\}$, because it has the largest element 8. Similarly, the multiset $\{28099, 3, 8\}$ is greater than the multiset $\{28099, 7, 6, 5, 7, 5, 5, 6, 0, 1\}$.

♠

Exercise 23

1. *Which of the multisets* $\{5, 3, 3, 2\}$*,* $\{5, 3, 2, 2, 2, 1\}$*, and* $\{5, 4\}$ *is the smallest? The greatest?*
2. *Explain that for any multisets* m *and* m' *of natural numbers, it is either the case that* m *is greater than* m'*, or* m' *is greater than* m*, or* m *and* m' *are the same multiset.*
3. *(Difficult right now? Try it!) Show that for any multiset* m_0 *over the natural numbers, there is no* infinite *sequence*

   $$m_0 > m_1 > m_2 > m_3 > \ldots$$

   of multisets $m_0, m_1, m_2, m_3, \ldots$ *such that each* m_i *is greater than* m_{i+1}*.*
4. *Recall the constructors for multisets of natural numbers in Section 2.1.5 and define a multiset comparison function*

   ```
   op _greaterThan_ : Multiset Multiset -> Boolean .
   ```

   in Maude and test it by running it on some examples.

   Hint: *You may need to define quite a lot of helpful functions to get this one right.*

2.4 Order-Sorted Equational Specifications

The many-sorted world is unnecessarily strict in that sorts are not related in any way. This hardly seems practical. For example, it is natural to have a sort `Nat` for the natural numbers and a sort `Int` for the integers. In the many-sorted world these sorts are totally disjoint, unless we just use the sort `Int` and forget about `Nat`. The latter approach is not very elegant,

[9] Note that $\{2, 1\}$ can be obtained from $\{2, 2, 1\}$ by replacing one occurrence of 2 with no elements.

since some functions, such as the factorial function ("!"), do not take negative numbers as arguments.

To have unrelated sorts `Int` and `Nat` is unsatisfactory as well. Essentially it requires functions used both for natural numbers and integers to be declared and defined twice, and does not allow the use of a natural number instead of an integer, etc. So, for example, the number 4 would have to be represented as e.g., `s(s(s(s(0))))` as a natural number and, e.g., `s'(s'(s'(s'(0'))))` when seen as an integer. (Alternatively, a conversion operator `(nat)_ : Int -> Nat` could be used to take an `Int` value to a `Nat` value.)

Similarly, in the object-oriented world we are used to *subclasses*, where, say, a class `Bird` may be a subclass of a class `Animal`.

Maude supports *order-sorted specifications* (see e.g. [42, 72, 95, 43], and [76] for an introduction in Norwegian), in which a sort may have *subsorts*. Intuitively, a subsort declaration

```
subsort s' < s .
```

means that the (sub)sort `s'` is "included" in the (super)sort `s`. That is, each element of `s'` is also an element of `s`.

Example 9 It is natural to have subsorts such as

```
subsort Nat < Int .
```

and

```
subsort Bird < Animal .
```

since the natural numbers are included in the integers, and all birds are included in the set of all animals. In Maude one may also combine subsort declarations into one declaration using the `subsorts` keyword:

```
subsorts Nat Neg < Int .
```

which states that both `Nat` and `Neg` are subsorts of `Int`. ♠

A subsort declaration does *not* also declare the sorts, so the sorts used above must also have been declared (using the keyword `sort` or `sorts`) as usual.

Formally, in an order-sorted signature, the set of sorts is equipped with a *partial ordering* $\leq$. The subsort relation $\leq$ induces a subsort relation $\leq$ on lists of sorts of the same length where

$$s_1 \ \ldots \ s_n \leq s'_1 \ \ldots \ s'_n$$

holds if and only if $s_i \leq s'_i$ for each $1 \leq i \leq n$.

2.4.1 Function Declarations

When `Nat` is a subsort of `Int`, a function which is declared to take `Int` arguments will also accept `Nat` arguments, since any `Nat` value is also an `Int` value. For example, a function

```
op _+_ : Int Int -> Int .
```

also applies to natural numbers. Still, one could add a declaration

```
op _+_ : Nat Nat -> Nat .
```

to tell the Maude system that the value of m+n has sort `Nat` if both m and n have sort `Nat`. In Maude, such declarations of *subsort overloaded* functions are mostly needed for constructors so that every constructor term gets the appropriate sort.

□ The following example shows that there are some order-sorted specifications where it is not obvious which is the sort of a ground term.

Example 10

```
fmod NO-PREREG is
  sorts s1 s2 s12 u1 u2 .
  subsorts s12 < s1 s2 .
  op a : -> s1 .
  op b : -> s2 .
  op c : -> s12 .
  op f : s1 -> u1 .
  op f : s2 -> u2 .
endfm
```

The sort `s12` is a subsort of the sorts `s1` and `s2`. The sorts `u1` and `u2` are unrelated. The question is: what is the sort of the term `f(c)`? Since `c` is an element of sort `s1`, the term `f(c)` should have sort `u1`, but since `c` is also an element of sort `s2`, the term `f(c)` should have sort `u2`. Such ambiguity is not desired since `u1` and `u2` are totally unrelated. In Maude, this is solved by requiring that each term has a *unique least sort*. In the above case, `f(c)` has sorts `u1` and `u2`, neither of which is smaller than the other, and therefore the term has no *unique* smallest sort, and Maude will print a warning that the specification fails a (*pre-regularity*) check. The specification would be OK if we added

```
  sort u12 .
  subsorts u12 < u1 u2 .
```

and a declaration

```
  op f : s12 -> u12 .
```

since the smallest sort of `f(c)` would be `u12`. ♠

□

2.4.2 Order-Sorted Signatures

An *order-sorted signature* is just a many-sorted signature with an additional partial ordering $\leq$ on the sorts:

Definition 8 (Order-sorted signature) *An order-sorted signature is a triple $(S, \leq, \Sigma)$, where S is a set (of sorts), $\leq$ is a partial ordering on S, and Σ is an $S^* \times S$-sorted family $\{\Sigma_{\omega,s} \mid \omega \in S^*, s \in S\}$.*

As indicated above, some restrictions on order-sorted signatures are needed for them to make sense. To describe these restrictions easily, we first need to define the set of terms of an order-sorted specification.

Terms are defined as expected, namely, if $s' \leq s$, then a term of sort s' is also a term of sort s. (A term of sort `Nat` is also a term of sort `Int`.) Variable sets are defined as for many-sorted systems.

Definition 9 (Terms in order-sorted signatures) *Given an order-sorted signature $(S, \leq, \Sigma)$ and a variable set $X = \{X_s \mid s \in S\}$, the S-sorted set of terms $\mathcal{T}_\Sigma(X) = \{\mathcal{T}_{\Sigma,s}(X) \mid s \in S\}$ is defined by "adding" the following condition*

0. $\mathcal{T}_{\Sigma,s'}(X) \subseteq \mathcal{T}_{\Sigma,s}(X)$ if $s' \leq s$; that is, a term in a subsort s' is also a term of the supersort s.

to Definition 5 which defines the terms in a many-sorted signature.

Again the set of ground terms $\mathcal{T}_\Sigma$ is defined as expected: $\mathcal{T}_\Sigma = \{\mathcal{T}_{\Sigma,s} \mid \mathcal{T}_{\Sigma,s} = \mathcal{T}_{\Sigma,s}(\emptyset), s \in S\}$.

Definition 10 (Least sort) *A term $t \in \mathcal{T}_{\Sigma,s}(X)$ has a unique* least sort *if the set $\{s \mid t \in \mathcal{T}_{\Sigma,s}(X)\}$ of sorts of t has a unique smallest element w.r.t. $\leq$, in which case this unique least sort of t is denoted $LS(t)$.*

We can now state the requirement on the sensibility of signatures:

Definition 11 (Sensible signatures) *An order-sorted signature $(S, \leq, \Sigma)$ is* sensible *if for any function symbol declaration $f : s_1 \dots s_n \to s \in \Sigma$ with $n \geq 1$, and any sequence $s'_1 \dots s'_n$ with $s'_i \leq s_i$ for all i, the term $f(x_1, \dots, x_n)$, where x_i is a variable of sort s'_i for each i, has a unique least sort.*

□ There are various slightly different "sensibility" criteria in the literature, usually denoted "regularity" or "preregularity". Our notion "sensible" should not be confused with the most flexible criterion for order-sorted signatures, given in [72].

Example 11 We may use `0` for both `false` in a sort `Boolean` and for the value 0 in a sort `Nat`, and may use `+` both for disjunction ("or") on the Booleans and for addition on the natural numbers. The signature

```
fmod OVERLOADED is
  sorts Boolean Nat .
  ops 0 1 : -> Boolean [ctor] .          *** ``false'' and ``true''
  op 0 : -> Nat [ctor] .                 *** The number 0
  op s : Nat -> Nat [ctor] .
  op _+_ : Boolean Boolean -> Boolean .  *** ``or''
  op _+_ : Nat Nat -> Nat .
endfm
```

is sensible. ♠

In sensible signatures, if we know the sorts of the constants in a term, each term has a unique least sort. For example, `0 + 0` has sorts `Nat` and `Boolean`, but the term `(0).Nat + (0).Nat` (we have told the system that we consider the `Nat`-constant `0`) has unique least sort `Nat`. □

Exercise 24 *Explain why the order-sorted signature*

```
fmod INSENSITIVE is
  sorts Nat NeList NeSet .
  subsorts Nat < NeList NeSet .
  op __ : NeList NeList -> NeList .
  op __ : NeSet NeSet -> NeSet .
endfm
```

(which tries to declare the signature of both nonempty lists and nonempty sets of natural numbers using the same constructor) is not sensible. How would you modify the above signature to make it sensible?

Maude will print a warning message if signatures are not sensible. We will therefore require, and assume, that all order-sorted signatures are sensible.

Exercise 25 *Let's consider the following signature:*

```
  sorts s1 s2 s3 s4 .
  subsorts s2 s3 < s4 .
  subsort s2 < s1 .
  op a : -> s3 .
  op b : -> s2 .
  op g : s3 s2 -> s1 .
  op g : s2 s1 -> s2 .
  op g : s1 s1 -> s4 .
```

1. *Draw the "subsort graph" of this signature (that is, draw a some graph for yourself so that can see the subsort relation).*
2. *Is the signature sensible?*
3. *Can you list at least 4 ground terms of sort* `s4`*? Of sort* `s1`*?*

4. *What is the least sort of the terms* `a` *and* `g(b, g(b, g(a,b)))`*?*

5. *Explain why we cannot add a declaration*

```
op g : s4 s4 -> s4 .
```

and still have a sensible signature.

2.4.3 Order-Sorted Equational Specifications

An order-sorted equational specification consists of an order-sorted signature and a set of unconditional and conditional equations, just like in the many-sorted case. The sorts of the terms t and t' in an equation $t = t'$ must be in the same *connected component*[10] of the partially ordered set $(S, \leq)$ of sorts, and analogously for conditional equations [42]. (Intuitively, two sorts s and s' are in the same connected component of $(S, \leq)$ if there is a "path" from s to s' when you draw the partially ordered set $(S, \leq)$ as an undirected graph.)

When we compute with equations by using them from left from right, it is, for various reasons, an advantage that the equations are *sort-decreasing* [41]. That is, for each *instance* of an equation, the least sort of the lefthand side should be greater than or equal to the least sort of the righthand side of the equation.

2.4.4 Examples of Order-Sorted Equational Specifications

This section illustrates the use of order-sortedness with some typical cases.

Inclusion

The main motivation behind order-sortedness is of course inclusion of domains, such as the inclusion of the natural numbers `Nat` in the sort `Int`.

Partiality

We have defined the data type of natural numbers with addition, subtraction, and multiplication. However, we omitted division. The reason is that division is a *partial* function on the natural numbers, since $n/0$ is undefined for any n.

So, what do we do? In many-sorted algebra, we could just avoid to give value to any term `n/0`. This would then be a "junk" term of sort `Nat`, and would lead to more junk such as `n/0 + s(0)`, which is clearly not elegant. In other formalisms one may add an explicit element $\perp$ denoting "undefined", and could then define `n/0` $= \perp$. Maude does not use this option. Instead, in Maude we can use subsorts to avoid having to deal with terms such as `n/0`. For example, we can have a subsort `NzNat` of `Nat`, denoting the nonzero natural numbers 1, 2, 3, ...:

[10] A connected component of $(S, \leq)$ is an equivalence class in the transitive and symmetric closure of $(S, \leq)$.

```
  sorts NzNat Nat .
  subsort NzNat < Nat .
```

The constructors should be declared so that `NzNat` contains all the nonzero positive numbers:

```
  op 0 : -> Nat [ctor] .
  op s : Nat -> NzNat [ctor] .
```

The division operator `/` can then be declared to have only nonzero denominator values as follows:

```
  op _/_ : Nat NzNat -> Nat .
```

Now we don't have to worry about n/`0`, since this is not a term.

Exercise 26 *We have declared* `s` *to go from* `Nat` *to* `NzNat`*. What's wrong with the following declarations? Why can't either of them be used instead?*

```
  op s : Nat -> Nat [ctor] . or
  op s : NzNat -> NzNat [ctor] . or
  op s : NzNat -> Nat [ctor] .
```

Exercise 27 *Define a Maude module* `NAT` *with sorts* `NzNat` *and* `Nat` *as above with all the usual functions for natural numbers (just copy them from the module* `NAT1` *in Section 2.3.3), including the division function* `/`*.* Hint: *You may want to use both* `+`*,* `<=`*, and* `monus` *in the definition of* `/`*. (Remember that conditional equations in Maude are preceded by the keyword* `ceq`*.)*

Constructors for the Integers

It is for many purposes desirable to have constructors so that the constructor ground terms are in a one-to-one correspondence with the domain they span. There is such a one-to-one correspondence for the natural numbers, since each constructor ground term corresponds to exactly natural number, and vice versa.

Without subsorts it is fairly tricky to represent the *integers* (i.e., both the positive and negative numbers) naturally so that each integer corresponds to exactly one constructor term, and vice versa. However, it is easy to have this desired one-to-one correspondence also for the integers by using subsorts as follows:

```
fmod INTEGERS is
  protecting BOOLEAN .
  sorts Zero NzNat NzNeg Nat Neg Int .
  subsorts Zero < Nat Neg < Int .
  subsort NzNat < Nat .
  subsort NzNeg < Neg .
```

The sort `Zero` is the intended sort for `0`, `NzNat` and `NzNeg` denote the nonzero natural and negative numbers, respectively, `Nat` and `Neg` all natural, respectively negative numbers, including 0, and `Int` denotes all integers. The sort `NzInt` denoting nonzero integers is added to deal with division:

```
sort NzInt .
subsorts NzNat NzNeg < NzInt < Int .
```

We use the following well-known constructors for the natural numbers:

```
op 0 : -> Zero [ctor] .
op s : Nat -> NzNat [ctor] .
```

How do we build the negative numbers? There are two intuitive options. One is to *negate* a natural number to get a negative number (like `- s(s(0))` to represent -2):

```
op -_ : NzNat -> NzNeg [ctor prec 15] .
```

Exercise 28 *We do not want* `op -_ : Nat -> Neg [ctor] .` *because then one number would have two different representations. Which number?*

The other option is symmetric to the construction of the natural numbers, namely, to use a "predecessor" function `p`, where `p`(x) is the predecessor of x (that is, $x-1$), just as `s`(n) is the successor of n. Such a constructor should be declared

```
op p : Neg -> NzNeg [ctor] .
```

In this case, -2 is represented by `p(p(0))`. I guess that it is a matter of taste *which* of the two suggested constructors one prefers. In either case, it should be possible to see that each constructor term represents exactly one integer, and vice versa.

Exercise 29 *For both sets of constructors suggested, show how we represent the following numbers: -3, 5, 0, -0, -4.*

In the definition below of the data type of integers, I have chosen `-_` to be the constructor for the negative numbers instead of `p`. A suitable set of non-constructor functions in the module `INTEGERS` could be the following, which corresponds closely to that in `NAT`. The difference is that a "normal" minus operator `_-_` replaces `monus`, and that a new function symbol `abs` denotes the absolute value of an integer:

```
ops _+_ _-_ : Int Int -> Int [prec 33] .
op _*_ : Int Int -> Int [prec 31] .
op _/_ : Int NzInt -> Int [prec 31] .
ops _<=_ _<_ _>=_ _>_ _==_ : Int Int -> Boolean .
ops max min : Int Int -> Int .
op abs : Int -> Nat .
```

The following variables are used in my specification of `INTEGERS`:

```
vars M N : Nat .          vars I J : Int .
var NEG : Neg .           var NZNEG : NzNeg .
var NZN NZN' : NzNat .    var NZI : NzInt .
```

First, we define addition on the natural numbers:

```
eq 0 + I = I .
eq s(M) + N = s(M + N) .
```

Then, we define subtraction on the naturals:

```
eq I - 0 = I .
eq 0 - NZN = - NZN .
eq s(M) - s(N) = M - N .
```

Then we can define addition on all integers[11]:

```
eq - NZN + (- NZN') = - (NZN + NZN') .
eq M + (- NZN) = M - NZN .
eq (- NZN) + N = N - NZN .
```

Exercise 30

1. *Argue informally that* + *is "defined" for all integers. That is, any term* t + t' *can be reduced to a constructor term if both* t *and* t' *are constructor terms.*
2. *Argue that each equation is "correct."*

We continue by defining subtraction on all integers:

```
eq 0 - (- NZN) = NZN .
eq (- NZN) - (- NZN') = NZN' - NZN .
eq M - (- NZN) = M + NZN .
eq (- NZN) - N = - (NZN + N) .
```

Finally, we define multiplication:

```
eq 0 * I = 0 .
eq s(N) * I = (N * I) + I .
eq (- NZN) * I = - (NZN * I) .
```

[11]The extra parentheses in the following equations are not needed, due to the precedence on the operators. They are just added for readability.

Exercise 31 *Define the above functions when the predecessor function* `p` *is the constructor instead of* `-_`*. Then, use Maude to test whether your specification is correct.*

Exercise 32 *An attempt to define the comparison function* `<=` *could be*

```
eq NEG <= N = true .
eq N <= NZNEG = false .
eq (- NZN) <= (- NZN') = NZN' <= NZN .
eq s(M) <= s(N) = M <= N .
```

Explain why these equations do not define `<=` *for all pairs of integers. Then add the "missing" equation(s).*

Exercise 33 *Define the functions* `abs`, `min`, *and* `max`.

Exercise 34 *Define the functions* `==`, `<`, `>=`, `>`, *and* `/` *on the integers and test your specification in Maude.*

Elements in a List, a Set, a Multiset, etc.

Recall our constructors for lists of natural numbers:

```
op nil : -> List [ctor] .
op __ : List Nat -> List [ctor] .
```

Lists are then of the form `nil` $n_1 \ldots n_k$. It is possible to instead declare lists by saying that a natural number is also a list (of length one):

```
subsort Nat < List .
```

and then declare the constructors

```
op nil : -> List [ctor] .
op __ : List List -> List [ctor] .
```

Hence, `nil` is a list, `s(s(0))` is a list, `(0 s(s(0))) (s(0) s(s(s(0))))` is a list, and `nil s(s(0))` is a list. The advantage is that lists are more elegantly presented as a sequence of numbers. The disadvantage is that the one-to-one correspondence between constructor terms and "lists" is gone, since `s(s(0))` and `nil s(s(0))` represent the same list, as do `(s(s(0)) 0) s(0)` and `s(s(0)) (0 s(0))`. The next chapter will show some Maude features which "remove" these problems.

Similarly, for sets, multisets, etc., we can also declare

```
subsort Nat < Set .
```

to say that a natural number is also a (singleton) set of natural numbers.

We will quite frequently use this technique in this course.

Explicit "Undefined" Value

Sometimes we may be interested in having an additional "default" (or "error" or "uninitialized") value added to a sort (just like `null` is such a "default" pointer address in languages like C). The quick fix is to have a supersort, say `DefNat` in the case of natural numbers, and define a constant of that sort, such as in the following:

```
sort DefNat .
subsort Nat < DefNat .
op noNat : -> DefNat [ctor] .
```

Hence, the sort `Nat` denotes the natural numbers while its supersort `DefNat` contains all the natural numbers and the value `noNat`.

2.5 Membership Equational Logic Specifications

Maude supports the specification and execution not only of order-sorted specifications but also of *membership equational logic* specifications. Membership equational logic [72, 4] is an elegant extension of order-sorted equational logic which will not be explained in detail in this course. Instead, two prototypical problems will illustrate the benefit of membership equational logic.

Consider first the use of the sort `NzInt` to avoid division by `0`. The result was that `s(0) / 0` is not a term of any sort and we did not need to worry about it. So far so good. But what about the term `s(s(0)) / (s(s(0)) - s(0))`? Have we shot ourselves in the foot with fancy subsorting? Although `s(s(0)) / (s(s(0)) - s(0))` (that is, $2/(2-1)$) seems perfectly sensible and should have value `s(s(0))`, the term `s(s(0)) - s(0)` is not a term of sort `NzInt`, and therefore the term `s(s(0)) / (s(s(0)) - s(0))` is not a well-formed term!

Membership equational logic allows expressions like `s(s(0)) / (s(s(0)) - s(0))` and gives them "the benefit of doubt." Such an expression does not have a sort like `Int` but an *error sort*.[12] The term `s(s(0)) / (s(s(0)) - s(0))` is evaluated by computing wherever possible, so that eventually it is evaluated to `s(s(0)) / s(0)` using the equations for `-`. This term is a well-formed term of sort `Int` and the computation can proceed to give the desired result:[13]

```
Maude> red s(s(0)) / (s(s(0)) - s(0)) .
result NzNat: s(s(0))
```

Similarly, the term `s(0) / (s(0) - s(0))` is also given the benefit of doubt and is reduced to `s(0) / 0`, which does not have a sort and which cannot be further reduced, and is therefore a term which has an "error sort" (which is the *kind* `[Int]`):

```
Maude> red s(0) / (s(0) - s(0)) .
result [Int]: s(0) / 0
```

[12] In membership equational logic, this error sort is the *kind* of the corresponding non-error sort.

[13] We remove some Maude output such as the number of reductions performed.

Membership equational logic tackles the problem of defining subsorts which are difficult or impossible to define by syntactic means in a signature.

Consider for example lists of natural numbers. We may want to have a subsort `OrderedList` for ordered (or sorted) lists. Once we have such a sort, we could define operations such as `merge` which are defined only on ordered lists. The first action is to define the subsort `OrderedList`:

```
sort OrderedList .
subsort OrderedList < List .
```

The next step is to define the "values" of this new sort. Obviously, we can by syntactic means define `nil` to be a constant of sort `OrderedList`. How can we say that a longer (and ordered) list is a term of sort `OrderedList`? The attempts

```
op __ : List Nat -> OrderedList [ctor] .
```

and

```
op __ : OrderedList Nat -> OrderedList [ctor] .
```

are both futile. Obviously, we would like to state something like "`OL N` is an ordered list when `OL` is an ordered list and `N` is greater than or equal to the rightmost element in `OL`." This statement can be modeled in Maude which supports *membership axioms* of the form

`mb` t `:` s `.`

and conditional membership axioms of the form

`cmb` t `:` s `if` *cond* `.`

where *cond* is a conjunction which can have both membership tests of the form $t' : s'$ and, as usual, equations of the form $u = u'$, for terms t, t', u, u' and sorts s, s'. The specification defining the sort `OrderedList` can be given as follows:

```
fmod ORDERED-LIST-NAT1 is
  protecting LIST-NAT1 .
  sort OrderedList .
  subsort OrderedList < List .

  var N : Nat .   var OL : OrderedList .

  mb nil : OrderedList .
  cmb OL N : OrderedList if (last(OL) <= N) = true .
endfm
```

We can test the specification by checking the sorts of some lists:

```
Maude> red nil s(0) s(s(0)) .
result OrderedList: nil s(0) s(s(0))
```

The result is a term of sort `OrderedList`. Reducing the list `nil s(0) 0` gives a term of (least) sort `List` as expected. The sort `OrderedList` could also be defined as follows:

```
var L : List .
cmb L : OrderedList if isSorted(L) = true .
```

It should be noted that membership axioms may easily lead to infinite looping, in particular when the "lefthand side" of a membership axiom is just a variable.

□ Intuitively[14], in membership equational logic, each connected component of the partially ordered set $(S, \leq)$ of sorts has a *kind*. We write `[`s`]` in Maude for the kind of the connected component of the sort s. Note that `[`s_1`]` and `[`s_2`]` denote the same kind when s_1 and s_2 are in the same connected component. The kind `[`s`]` can be seen as the (error) supersort of the connected component to which s belongs, in that all s-terms and "error" terms "of sort s" has this kind. The Maude system automatically adds a declaration

```
op f : [s1] ... [sn] -> [s] .
```

for any declaration

```
op f : s1 ... sn -> s .
```

in the specification. Therefore, a declaration

```
op _/_ : Int NzInt -> Int .
```

means that the Maude specification (implicitly) contains a declaration

```
op _/_ : [Int] [NzInt] -> [Int] .
```

Since `s(0) - s(0)` is a term of sort `Int`, and hence also of kind `[Int]`, also the term `s(0) / (s(0) - s(0))` has kind [Int] due to the implicit declaration above and the fact that `[NzInt] = [Int]`. Since `s(0) / (s(0) - s(0))` is a "well-kinded" term, it can be further reduced to the term `s(0) / 0` of kind `[Int]`. This term cannot be reduced any further, and although well-kinded, it has no *sort*. Terms which do not have sorts, but only a kind, are often understood to be "error" terms.

Kinds are in many ways treated as ordinary sorts in Maude—except for defining sort constraints and testing for membership in a kind (which is totally unnecessary!). We may declare variables such as

```
var INT-OR-ERROR : [Int] .
```

or functions such as

```
op _/_ : Int Int -> [Int] .
```

The latter declaration can also be written

```
op _/_ : Int Int ~> Int .
```

where the arrow `~>` states that `_/_` is a partial function. □

[14] The mathematical definition of membership equational logic (see [72]) is quite different from the exposition given below, which focuses more on the use of kinds and subsorts in Maude.

2.6 Parameterized Modules

We have introduced data types for *lists* of natural numbers, for *sets* of natural numbers, etc. If we want lists of e.g., Booleans, integers, or of lists of natural numbers, it seems that we'd need to define all such modules from the beginning.

Obviously, that is not the case. Modules can be *parameterized* so that we can have generic specifications of lists, such as `LIST{X :: ELEM}`, where `X` is a parameter which range over all modules which "satisfy" the "interface" (called *theory*) `ELEM`. This parametric module can be instantiated to `LIST{Nat}`, `LIST{Bool}`, etc. Maude has very powerful parameterization features. However, we will *not* introduce parameterization in this course to avoid learning too many language constructs. The interested reader should confer e.g., the Maude manual [13].

2.7 Built-in Modules and Functions

This section presents some modules and functions which are built into Maude and which are either automatically imported by any user module, or can be imported by any user module. They are all defined in the file `prelude.maude` which is read when you start Maude. (You can change this file if you feel like redefining the built-in modules or feel like giving commands which should always be executed upon the start of Maude.)

For example, a built-in module `BOOL` containing the Boolean constants and functions is automatically imported in any user-defined module. Furthermore, for efficiency and ease-of-specification purposes Maude has a built-in data type for arbitrary large natural numbers and integers. This allows us to write numbers as "`2008`" instead of the very much more cumbersome `s(s(...s(0)...))`, and which computes `2008 + 1001` as efficiently as C++. Maude also provides built-in data types for strings and floating point numbers.

2.7.1 The Module `BOOL`

The following module `BOOL` defines the *Boolean* values and some useful functions:

```
fmod TRUTH-VALUE is
  sort Bool .
  op true : -> Bool [ctor special (id-hook SystemTrue)] .
  op false : -> Bool [ctor special (id-hook SystemFalse)] .
endfm

fmod TRUTH is
  protecting TRUTH-VALUE .
  op if_then_else_fi : Bool Universal Universal -> Universal
                                                   [poly (2 3 0) special ...] .
  op _==_ : Universal Universal -> Bool [prec 51 poly (1 2) special ...] .
  op _=/=_ : Universal Universal -> Bool [prec 51 poly (1 2) special  ...] .
endfm
```

```
fmod BOOL is
  protecting TRUTH .
  op _and_ : Bool Bool -> Bool [assoc comm prec 55] .
  op _or_ : Bool Bool -> Bool [assoc comm prec 59] .
  op _xor_ : Bool Bool -> Bool [assoc comm prec 57] .
  op not_ : Bool -> Bool [prec 53] .
  op _implies_ : Bool Bool -> Bool [gather (e E) prec 61] .
  vars A B C : Bool .
  eq true and A = A .
  eq false and A = false .
  eq A and A = A .
  eq false xor A = A .
  eq A xor A = false .
  eq A and (B xor C) = A and B xor A and C .
  eq not A = A xor true .
  eq A or B = A and B xor A xor B .
  eq A implies B = not(A xor A and B) .
endfm
```

The `special` attribute associated to the operators states that these are treated as built-in operators having associated C++ code. The attributes `assoc` and `comm` will be explained in the next chapter. In this course, we will ignore the `gather` attribute (see the Maude manual for an explanation if you are interested in this parsing issue).

In the rest of this course, we will use the built-in booleans and not our own module `BOOLEAN`.

□ The `xor` operator denotes "exclusive or," where x `xor` y is `true` if exactly one of x and y is `true`. □

The `poly` attribute states that the corresponding arguments (of "sort `Universal`") may have *any* sort. Therefore, the operators defined in the module with the pretentious name `TRUTH` may be applied to all kinds of terms. The operator `if_then_else_fi` behaves as expected, `x == y` holds if and only `x` and `y` are equal, and conversely for the inequality operator.

In Maude, if `b` is a term of sort `Bool`, then a condition `b = true` in an equation can be written just `b`, so we can write

```
  ceq M monus N = 0 if M <= N .
```

instead of

```
  ceq M monus N = 0 if (M <= N) = true .
```

Finally, for any term t and sort s, the expression t `::` s (note the *double* colon) is a term of sort `Bool` which is `true` if and only if t has sort s.

□ The prelude file then defines the module

```
fmod EXT-BOOL is
  protecting BOOL .
  op _and-then_ : Bool Bool -> Bool [strat (1 0) gather (e E) prec 55] .
  op _or-else_ : Bool Bool -> Bool [strat (1 0) gather (e E) prec 59] .
  var B : [Bool] .
  eq true and-then B = B .
  eq false and-then B = false .
  eq true or-else B = true .
  eq false or-else B = B .
endfm
```

The attribute `strat (1 0)` tells Maude to execute an expression t `and-then` t' by first evaluating the *first* ('1') argument t, and then evaluating the *whole* ('0') expression t `and-then` t' using one of the first two equations above. Therefore, if the first argument t evaluates to `false`, the system does not need to worry about the second argument t', since t `and-then` t' will anyways be `false`. The situation is similar with `or-else`: if its first argument evaluates to `true`, the whole expression evaluates to `true`. For example, such a strategy is useful to evaluate an expression like `N > 0 and-then (M / N) > 3`. □

2.7.2 The Natural Numbers

Maude provides the following module for arbitrarily large *natural numbers*[15], whose implementation uses the GMP library [39].

```
fmod NAT is
  protecting BOOL .
  sorts Zero NzNat Nat .
  subsort Zero NzNat < Nat .

  op 0 : -> Zero [ctor] .
  op s_ : Nat -> NzNat [ctor iter special ...] .

  op _+_ : NzNat Nat -> NzNat [assoc comm prec 33 special ...] .
  op _+_ : Nat Nat -> Nat [ditto] .

  op sd : Nat Nat -> Nat [comm special ...] .

  op _*_ : NzNat NzNat -> NzNat [assoc comm prec 31 special ...] .
  op _*_ : Nat Nat -> Nat [ditto] .

  op _quo_ : Nat NzNat -> Nat [prec 31 gather (E e) special ...] .
  op _rem_ : Nat NzNat -> Nat [prec 31 gather (E e) special ...] .

  op _^_ : Nat Nat -> Nat [prec 29  gather (E e) special ...] .
  op modExp : Nat Nat NzNat ~> Nat [special ...] .

  op gcd : NzNat NzNat -> NzNat [assoc comm special ...] .
```

[15]Notice that the numbers are not restricted to 32 or 64 bits

```
  op gcd : Nat Nat -> Nat [ditto] .
  op lcm : NzNat NzNat -> NzNat [assoc comm special ...] .
  op lcm : Nat Nat -> Nat [ditto] .

  op min : NzNat NzNat -> NzNat [assoc comm special ...] .
  op min : Nat Nat -> Nat [ditto] .

  op max : NzNat Nat -> NzNat [assoc comm special ...] .
  op max : Nat Nat -> Nat [ditto] .

  op _<_ : Nat Nat -> Bool [prec 37 special ...] .
  op _<=_ : Nat Nat -> Bool [prec 37 special ...] .
  op _>_ : Nat Nat -> Bool [prec 37 special ...] .
  op _>=_ : Nat Nat -> Bool [prec 37 special ...] .

  op _divides_ : NzNat Nat -> Bool [prec 51 special ...] .
  ...
endfm
```

The constructors for `Nat` are `0` and `s_` so the natural numbers are represented by the terms `0`, `s 0`, `s s 0`, ... However, for convenience, we may also write `0`, `1`, `2`, ...:

```
Maude> red s s 0 + s s s 0 .
result NzNat: 5
Maude> red 1234567 * 89 .
result NzNat: 109876463
```

There is no subtraction function on the natural numbers (since $m - n$ could be a negative number). Instead, the function `sd` denotes the (symmetric) difference (see Exercise 19) between two numbers.

Example 12 The factorial function may be defined either by induction on the constructors as in

```
fmod FACTORIAL is
  protecting NAT .
  op _! : Nat -> Nat .
  var N : Nat .
  eq 0 ! = 1 .
  eq (s N) ! = s N * (N !) .
endfm
```

or directly:

```
fmod FACTORIAL2 is
  protecting NAT .
```

```
  op _! : Nat -> Nat [prec 29] .
  var N : Nat .
  eq N ! = if N == 0 then 1 else N * sd(N, 1) ! fi .
endfm
```

□(In this last example, `_!` is given a lower precedence (29) than multiplication (31), so that `_!` binds tighter than `*`. This could of course have been done in the first module as well.)□

♠

The function `quo` defines division, `rem` the *remainder* function, `^` exponentiation ($m \hat{} n = m^n$), `modExp` is exponentiation modulo (`modExp`$(m, n, p) = m^n \ \% \ p$), `gcd` is the greatest common divisor, `lcm` is the least common multiple, and `<`, `<=`, `>`, and `>=` are the usual comparison operators.

□ Although we have not shown it above, the module `NAT` also has bit manipulating functions such as bitwise and (`&`), bitwise or (`|`), bitwise xor (`xor`), and, finally, right shift (`>>`) and left shift (`<<`). □

Exercise 35 *Define a module which imports* `NAT` *and contains a function* `double` *which doubles its argument and a function* `monus` *which defines the "monus" operator. Test your specification by running it in Maude.*

Exercise 36 *Define a function*

```
  op isPrime : NzNat -> Bool .
```

which returns `true` *if and only if its argument is a prime number (that is, it is a number which is not divisible by any number except 1 and itself). Test your specification by checking whether 14091 (not a prime!), 2 (prime), 31 (prime), 232557 (not), and 135727 (?) are prime numbers.*

The `ditto` Attribute

□ Subsort overloaded operators must have the same set of attributes, except for `ctor`. If a function `f` is declared

```
  op f : s -> s [ctor assoc comm gather (e E) special ...] .
```

then a function declaration of `f` from a sort `s'` which is in the same connected component as `s` to some sort `s''` must be declared either

```
  op f : s' -> s'' [ctor assoc comm gather (e E) special ...] .
```

or

```
  op f : s' -> s'' [assoc comm gather (e E) special ...] .
```

(Don't worry about the meaning of the attributes; they will be explained later.) Maude provides the abbreviation `ditto` to avoid having to repeat these attributes. The attribute `ditto` stands for *all attributes except* `ctor` in previous declarations of the same (subsort overloaded) function symbol. Therefore, the above two declarations of `f` on `s'` could also have been declared, respectively,

```
op f : s' -> s'' [ctor ditto] .
```

and

```
op f : s' -> s'' [ditto] .
```

This also applies to functions with more than one argument, as is shown in the above module `NAT`. □

2.7.3 The Integers

The *integers* are constructed from the natural numbers by having a constructor

```
op -_ : NzNat -> NzInt [ctor special ...] .
```

so that negative numbers can be written as `- s 0`, `- 2003`, ..., and also as `-1`, `-2003`, ... (that is, with no space after the '-') in Maude 2.

The built-in efficient implementation of arbitrarily large integers are given in the following module, from which I have omitted all the functions in `NAT` which have a straight-forward extension into the integers (such as `+`, `*`, `<`, ...):

```
fmod INT is
  protecting NAT .
  sorts NzInt Int .
  subsorts NzNat < NzInt Nat < Int .

  op -_ : NzNat -> NzInt [ctor special ...] .
  op -_ : NzInt -> NzInt [ditto] .
  op -_ : Int -> Int [ditto] .

  op _+_ : Int Int -> Int [assoc comm prec 33 special ...] .
  op _-_ : Int Int -> Int [prec 33 gather (E e) special ...] .

  op _^_ : Int Nat -> Int [prec 29  gather (E e) special ...] .
  op abs : Int -> Nat [...] .
  ...
endfm
```

(The `abs` function gives the *absolute value* of a number.) Observe that the function `_-` is a constructor only on the sort `NzNat`, and is a non-constructor on `NzInt` and `Int`. The term `- 2003` is therefore a constructor term of sort `Int`, while `- 0` and `- -1234` are not.

Exercise 37

1. *Explain why the exponentiation function is not declared*

```
op _^_ : Int Int -> Int [...] .
```

2. *Use Maude to compute the value of* $(-7)^5$. *(Remember that you can use the Maude command* `select INT .` *to make* `INT` *your "current" module.)*

2.7.4 * The Rational Numbers

□ The *rational numbers* are defined on top of the integers in the module `RAT` which defines the sorts `NzRat` (for nonzero rational numbers) and `Rat` (for all rational numbers) as follows:

```
fmod RAT is
  protecting INT .
  sorts PosRat NzRat Rat .
  subsorts NzInt < NzRat Int < Rat .
  subsorts NzNat < PosRat < NzRat .
```

The constructor for the rationals is

```
op _/_ : NzInt NzNat -> NzRat [ctor prec 31 gather (E e) special ...] .
```

Exercise 38

1. *How would you represent* $\frac{2}{3}$, 5, *and* $-(\frac{7}{8})$ *as constructor terms in the above module* `RAT`*?*
2. *Why is the constructor not declared*

   ```
   op _/_ : Int NzNat -> Rat [ctor ...] .
   ```

   or

   ```
   op _/_ : NzInt NzInt -> NzRat [ctor ...] .
   ```

Given the constructor, one can define non-constructor functions such as

```
op _/_ : NzRat NzRat -> NzRat [ditto] .
op _/_ : Rat NzRat -> Rat [ditto] .
```

This operator is only a constructor on the domain `NzInt NzNat`, so that e.g. `2 / -3` and `4 / (2 / 3)` are *not* constructor terms of sort `Rat`. The following equations are used to make expressions as these into constructor terms:

```
var I J : NzInt .
var N M : NzNat .
var K : Int .
var Z : Nat .
var Q : NzRat .

eq 0 / Q = 0 .
eq I / - N = - I / N .
eq (I / N) / (J / M) = (I * M) / (J * N) .
eq (I / N) / J = I / (J * N) .
eq I / (J / M) = (I * M) / J .
```

The functions `-_` (unary minus), `+`, `-`, `*` are defined as follows:

```
  op -_ : NzRat -> NzRat [ditto] .
  op -_ : Rat -> Rat [ditto] .
  eq - (I / N) = - I / N .

  op _+_ : Rat Rat -> Rat [ditto] .
  eq I / N + J / M = (I * M + J * N) / (N * M) .
  eq I / N + K = (I + K * N) / N .

  op _-_ : Rat Rat -> Rat [ditto] .
  eq I / N - J / M = (I * M - J * N) / (N * M) .
  eq I / N - K = (I - K * N) / N .
  eq K - J / M = (K * M - J ) / M .

  op _*_ : NzRat NzRat -> NzRat [ditto] .
  op _*_ : Rat Rat -> Rat [ditto] .
  eq Q * 0 = 0 .
  eq (I / N) * (J / M) = (I * J) / (N * M).
  eq (I / N) * K = (I * K) / N .
```

The module `RAT` also defines well-known functions such as `^` and `<`, `<=`, `>`, and `>=` on the rationals.

Exercise 39 *The module* `RAT` *ends with the following function definitions:*

```
  op trunc : Rat -> Int .
  eq trunc(K) = K .
  eq trunc(I / N) = I quo N .

  op frac : Rat -> Rat .
  eq frac(K) = 0 .
  eq frac(I / N) = (I rem N) / N .

  op floor : Rat -> Int .
  op ceiling : Rat -> Int .
  eq floor(K) = K .
  eq ceiling(K) = K .
  eq floor(N / M) = N quo M .
  eq ceiling(N / M) = ((N + M) - 1) quo M .
  eq floor(- N / M) = - ceiling(N / M) .
  eq ceiling(- N / M) = - floor(N / M) .
endfm
```

Describe what these last functions are supposed to compute.

The careful reader may have noticed that constructor terms such as `4 / 6` and `5 / 1` are not further reduced by the equations in the module. At the same time, the constructor `_/_` has a `special` attribute indicating that it has some "built-in" capabilities. This built-in capability reduces expressions such as `4 / 6` and `5 / 1` to resp. `2 / 3` and `5`.

Finally, the very careful reader could come up with some simpler equations (can you think of which functions could be given "nicer" definitions?). The choice of equations is motivated by performance—simpler equations turn out to have a big performance penalty. □

2.7.5 * The Floating Point Numbers

□ The built-in module `FLOAT` implements the IEEE-754 double precision *floating point numbers*. The floating point numbers are declared

```
fmod FLOAT is
  protecting BOOL .
  sorts FiniteFloat Float .
  subsort FiniteFloat < Float .
  op <Floats> : -> FiniteFloat [special (id-hook FloatSymbol)] .
  op <Floats> : -> Float [ditto] .
```

This syntax means that the constructors are built-in. The floating point numbers are given as e.g. `1.0`, `-9.87654321`, and `-1.23e+14` (for $-1.23 \cdot 10^{14}$). The sort `Float` also contains two constants `Infinity` and `-Infinity` which are also used to denote out of range values:

```
Maude> red 3.45e+223 * 2.99e+210 .
result Float: Infinity
```

The module `FLOAT` contains all the expected functions such as `sqrt` (for square root), the trigonometric functions, the logarithm function etc. □

2.7.6 Strings

The following built-in Maude module `STRING` defines the sort `String` of *strings* of the form `"this is a string"`. Strings of length 1, such as `"a"` or `"b"` are constants of a subsort `Char`.

```
fmod STRING is
  protecting NAT .
  sorts String Char FindResult .
  subsort Char < String .
  subsort Nat < FindResult .
  op <Strings> : -> Char [special (id-hook StringSymbol)] .
  op <Strings> : -> String [ditto] .
  op notFound : -> FindResult .

  op ascii : Char -> Nat [special ...] .
  op char : Nat ~> Char [special ...] .
  op _+_ : String String -> String [prec 33 gather (E e) special ...] .
  op length : String -> Nat [special ...] .
  op substr : String Nat Nat -> String [special ...] .
  op find : String String Nat -> FindResult [special ...] .
  op rfind : String String Nat -> FindResult [special ...] .

  op _<_ : String String -> Bool [prec 37 special ...] .
  op _<=_ : String String -> Bool [prec 37 special ...] .
  op _>_ : String String -> Bool [prec 37 special ...] .
  op _>=_ : String String -> Bool [prec 37 special ...] .
endfm
```

The function `ascii` gives the ASCII value of a character; `char` does the opposite. The function `+` denotes string concatenation, and `length` returns, not unexpectedly, the length of a string. `substr`(s, p, l) returns the substring of s which starts at character $p + 1$ and is l characters long. `find`(s_1, s_2, p) finds the starting position (minus 1) of the substring s_2 in s_1, starting at

character number $p+1$ in s_1. `rfind` does the same, but starts looking "from the right." The comparison operators `<`, `<=`, `>`, and `>=` compare strings lexicographically.

Exercise 40

1. *Use Maude to find the ASCII number of the character* `"k"`*, and to find the 17th letter in the alphabet.*

2. *The British and the Americans like to state their sports scores in the form*

   ```
   "Luton 8 Liverpool 0"
   ```

   while the Europeans (and the Norwegians) prefer the notation

   ```
   "Luton - Liverpool 8-0"
   ```

   Define a function `europify : String -> String` *which takes a sports score in American format and returns it in European format. Test your function on some prototypical scores such as* `"MalmoFF 11 Rosenborg 1"` *and* `"49ers 39 Giants 38"`[16]*. You may assume that there are no blanks in the name of a team.*

2.7.7 * Conversions

□ The module `CONVERSION` defines some functions for converting a number to a string and vice versa, and for converting between rational numbers and floating point numbers.

For example, the function

```
op string : Rat NzNat ~> String [special ...] .
```

takes a rational number (and therefore also an integer) and a base (between 2 and 36), and displays the number in the given base. That is, `string`$(n, 10)$ returns the string displaying the decimal notation of n, while `string`$(n, 2)$ returns it binary form, so that `string(5, 2)` returns `"101"` and `string(29, 16)` returns 29 in hexa-decimal notation, namely `"1d"`. The function `rat` does the opposite.

Exercise 41 *Define a function*

```
op binary : Nat -> Nat .
```

which returns the "binary" value of a natural number, so that e.g. `binary(7)` *returns the natural number 111.*

□

[16] Those were the times!

2.7.8 Quoted Identifiers

A *quoted identifier* is any sequence of ASCII characters not containing any of the special characters '`[`', '`]`', '`(`', '`)`', '`{`', '`}`', ' ' (space/blank), and '`,`' (where '`‘`' is the escape character such that '`‘[`' is not a special character) and which is preceded by a quote ('`’`'). For example, `’Maude` is a quoted identifier, and so are `’’’INF220`, `’’Luton`, `’InternaZionale`, `’2002`, and `’<>__+*Harvest’’’Moon`, and `’` (the empty quoted identifier). Obviously, neither `Maude` nor `’Lu[ton` is a quoted identifiers. However, `’Lut‘[on` is a quoted identifier (why?). The sort `Qid` contains a constant for each quoted identifier.

```
fmod QID is
  protecting STRING .
  sort Qid .
  op <Qids> : -> Qid [special ...] .

  op string : Qid -> String [special ...] .
  op qid : String ~> Qid [special ...] .
endfm
```

2.7.9 The Rest of `prelude.maude`

The file `prelude.maude` goes on to define theories, views (mapping an "interface" theory to a module), parameterized modules, and instantiated parameterized modules. The file then defines Maude's *meta-level* and other useful modules, and ends with the Maude commands

```
set include BOOL on .
select CONVERSION .
```

The first command means that every module will automatically import the module `BOOL`. If you at any time want to turn off this automatic importation, as we wanted when we defined our own Boolean values, you may give the Maude command `set include BOOL off .` The `set include` and `set protect` commands can of course also be used for other modules. You may e.g. want to automatically include the integers in all your modules, in which case you could give the command `set protect INT on .` The second command makes `CONVERSION` the "current" module when you start Maude without giving any arguments.

2.8 A Final Important Remark

At the end of this chapter, it worth restating something from the beginning of the chapter: A module consists of a *set* (and *not* a *list*!) of declarations. This means that the order of declarations in a module does not matter, and *should not matter*. This also makes it easy to understand a specification by inspecting each equation by itself. Each equation should be mathematically correct!

Chapter 3

Operational Semantics of Equational Specifications

The previous chapter explained how to write equational specifications in Maude without really explaining the precise *meaning* of such specifications. This and the following chapters explain the meaning (or *semantics*) of equational Maude specifications. The *operational semantics* describes the "computational meaning" of a specification, namely, how the specification is "executed" in the Maude system. In Chapter 4 we define the *"logical meaning"* of a specification: Does it "follow logically" from the specification that the terms t and u are "equivalent"? Unfortunately, it is beyond the scope of this course to present also the *denotational semantics* (or "mathematical meaning" or *algebraic semantics*) of equational specifications: What mathematical object does a Maude specification define?

Once a specification has a well-defined meaning, it is possible to reason about the specification in various ways.

This chapter formally describes the operational semantics of equational specifications. We can reason about computations in Maude once we have an operational semantics for Maude. In this chapter we focus on *termination* and *confluence* properties. Termination (i.e., absence of infinite loops) is an important property of equational specifications (and of programs in other languages). Confluence essentially means that the result obtained by a computation in Maude is independent of how the equations are applied to a term. Termination and confluence are important properties which you have to get right, since Maude *assumes* that your specification satisfies these properties. Maude will not check them for you.

In more detail, this chapter deals with the following subjects:

- Formally describing how Maude computes with equations. This is done by defining mathematically (don't be afraid, no deep knowledge of mathematics is assumed) what it means that a ground term t *reduces* in one step to a term t' using some equation in the specification (Section 3.1).
- Defining what it means for a specification to be terminating and confluent, and showing that each term has a *unique normal form* ("value") in terminating and confluent systems (Section 3.2).

- Termination. It would be nice to have some way of *deciding* whether a specification is terminating or not. It is in general *undecidable* whether a specification is terminating. This means that there is no technique/algorithm/method which can *always* be used to determine whether or not a specification is terminating. Undaunted by this fact, Section 3.3 shows some techniques that can *quite often* be used to mathematically prove that a specification is terminating. In particular, we focus on *simplification orderings* and explain why these work.
- Confluence is decidable when the specification is terminating. We show how to check whether a terminating specification is confluent by checking whether it is *locally confluent* (Section 3.4).

The goal of this chapter is that

- you understand how equations are applied in Maude, and
- you can figure out whether a system is terminating and confluent

in order to write specifications that are both terminating and confluent. In some cases it is not possible and/or natural to write terminating and/or confluent specifications. Section 3.6 briefly discusses some methods for treating such systems:

- In Maude, we can rewrite/reduce modulo some equations such as commutativity. Section 3.6.1 shows how data types such as lists and multisets can be elegantly defined by declaring some functions to be *associative* and/or *commutative*. Section 3.7 ends this chapter by showing how the well known merge-sort and quick-sort algorithms can be easily and succinctly defined on such lists.
- There is a process, called *Knuth-Bendix completion* [57], which sometimes can turn a non-confluent system into a terminating and confluent system by adding equations.

In order to keep the exposition as simple as possible by avoiding technical details, we assume in this chapter that—unless stated otherwise—our specifications are *unsorted* (or *one-sorted*), meaning that there is only *one* sort in the specification, and that the equations are *unconditional*. At the end of this chapter we then discuss how our definitions and results carry over to the order-sorted case.

This (and the next) chapter is mostly based on Baader and Nipkow's *Term Rewriting and All That* [2] and (less so) on Bjørn Kirkerud's lecture notes on term rewriting [55]. The section on termination is based on Dershowitz [21].

3.1 The Reduction Relation

This section aims at formally defining what it means that a term t *reduces*[1] in one step using an equation in the module. For example, in the module `NAT-ADD` in the previous chapter, the

[1]Such reduction is usually called *rewriting* (or (equational) *simplification*) elsewhere, but to avoid confusion with non-equational rewriting in rewriting logic, I use *reduction* when equations are applied, and *rewriting* for the application of ("non-equational") rewrite rules in rewriting logic. Similarly, I use the symbol $\rightsquigarrow$ instead of the more common arrow $\longrightarrow$ for equational reduction.

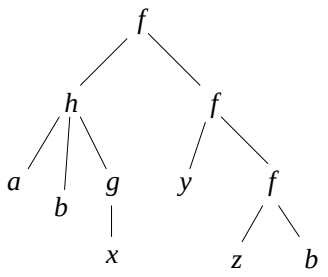

Figure 3.1: The tree structure of the term $f(h(a, b, g(x)), f(y, f(z, b)))$.

term `s(s(0 + s(0))) + 0` reduces to `s(s(s(0))) + 0` using the equation `0 + M = M`.

Notation. Unless stated otherwise, this chapter treats one-sorted signatures. Function symbols are not declared explicitly, but their declarations may be inferred from the context. We will often denote constants by a, a', b, c, ..., non-constant function symbols by f, g, h, ..., terms by t, t_1, t', u, ..., and variables by x, x', x_1, y, z, ... Therefore, a specification

$$\{f(a, g(b, x), y) = f(a, b, y)\,,\ h(c, c, z) = h(a, b, c)\}$$

denotes the same equational specification as the Maude module

```
mod M is
  sort s .
  ops a b c : -> s .
  ops f h : s s s -> s .
  op g : s s -> s .
  vars x y z : s .
  eq f(a, g(b,x), y) = f(a,b,y) .
  eq h(c,c,z) = h(a,b,c) .
endm
```

To define the notion of a reduction step, we first need to define the notions of *position*, *substitution*, and *matching*. These notions are fairly natural, and their mathematical description should not be too surprising. Please try to understand each of the concepts and don't worry too much about the definitions and technicalities of e.g., positions, variable substitutions, etc. Just get an intuitive feeling of what these notions are.

3.1.1 Positions

A term has tree structure. For example, the term $f(h(a, b, g(x)), f(y, f(z, b)))$ can be seen as the tree in Fig. 3.1. A *position* in a term is a string of numbers (with ϵ (or `e` in the figures) denoting the empty string) as seen in Fig. 3.2:

□

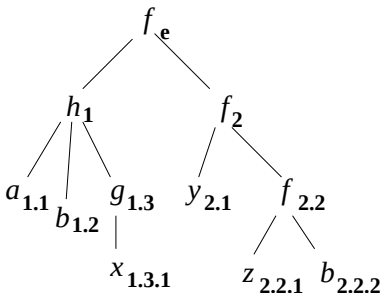

Figure 3.2: The positions of the term $f(h(a, b, g(x)), f(y, f(z, b)))$.

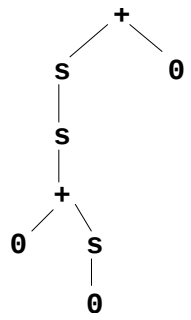

Figure 3.3: The tree structure the term `s(s(0 + s(0))) + 0`.

Definition 12 (Position) *A* position *is a* string *of natural numbers which denotes a place in a term/tree. The root position is denoted by the empty string, written ϵ, and $i.p$ denotes the position p in the ith subtree of the term. (We omit trailing dots.)*

□

In Fig. 3.2 we have drawn all the positions of the term $f(h(a, b, g(x)), f(y, f(z, b)))$. Note that e.g., 3, 1.1.1, and 2.2.3 are *not* positions in the term $f(h(a, b, g(x)), f(y, f(z, b)))$.

A term with infix function symbols can also be written in prefix form, so that `s(s(0 + s(0))) + 0` can be regarded as the term with tree structure given in Fig. 3.3.

Exercise 42 *Decorate the tree in Fig. 3.3 with its positions.*

If p is a position in a term t, we denote by $t|_p$ the *subterm of t at position p.*

□

Definition 13 *If p is a (possibly empty) position in a term t, then $t\,|_p$ can be defined inductively as follows:*

$$t\,|_{\epsilon} = t$$
$$f(t_1, \ldots, t_n)\,|_{i.p} = t_i\,|_p$$

□

The term $t\,|_p$, for p a position in t, is called a *subterm* of t. If $p \neq \epsilon$, then $t\,|_p$ is called a *proper subterm* of t.

Example 13 The subterm of $h(a, b, g(x))$ at position 3 is $g(x)$ and $h(a, b, g(x))\,|_{3.1}$ is x. The subterms of $h(a, b, g(x))$ are $h(a, b, g(x))$ and a and b and $g(x)$ and x. The last four are proper subterms. ♠

If t and u are terms, and p is a position in t, then $t[u]_p$ is the term t where $t\,|_p$ is replaced by u in position p. That is, we put u into t at position p in t:

□

Definition 14 *If t and u are terms, and p is a position in t, then $t[u]_p$ is defined inductively as follows:*

$$t[u]_\epsilon = u$$
$$f(t_1, \ldots, t_i, \ldots, t_n)[u]_{i.p} = f(t_1, \ldots, t_i[u]_p, \ldots, t_n)$$

□

Example 14 For example, $f(a, f(x, g(y)))[b]_2$ is $f(a, b)$, and $f(a, f(x, g(y)))[c]_\epsilon$ is just c, and $f(a, f(x, g(y)))[c]_{2.2.1}$ is $f(a, f(x, g(c)))$. ♠

Exercise 43

1. *What is $f(a, b)\,|_2$, and what is $f(h(c), g(d, g(a, f(a, b))))\,|_{2.2.1}$?*
2. *What is* `(s(s(0 + s(0))) + 0)[s(0)]`$_{1.1.1}$*? And $f(h(c), g(d, g(a, f(a, b))))[f(b, b)]_{2.2}$?*

We denote by $var(t)$ the set of variables in a term t; e.g., $var(f(a, g(x, f(b, z)))) = \{x, z\}$.

3.1.2 Variable Substitutions

A *variable substitution* (or just *substitution*) is an assignment of terms to variables, and is usually written explicitly as $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where we do not mention variables which are mapped to themselves. If σ is a substitution $\sigma : X \to T_\Sigma(Y)$, then we also denote by σ its (homomorphic) extension $\sigma : T_\Sigma(X) \to T_\Sigma(Y)$ which takes a term and (simultaneously) replaces each variable in the term according to the substitution. We often write substitutions "postfix." For example, if σ is $\{x \mapsto a,\ y \mapsto g(x, y),\ z \mapsto h(z, z)\}$ and t is the term $f(x, x, f(x, y, z))$, then $t\sigma$ is $f(a, a, f(a, g(x, y), h(z, z)))$.

Exercise 44 *Let t and σ be as above. What is $(t\sigma)\sigma$? What is $t(\sigma \circ \sigma)$ (where $\circ$ denotes ordinary function composition: i.e., $(f \circ g)(x) = f(g(x))$ for all f, g, x)?*

A substitution which assigns a ground term to each variable is called a *ground substitution*.

3.1.3 Matching

Definition 15 (Matching) *A term t* matches[2] *a term u if there is a substitution σ such that $t\sigma = u$. In this case u is called an* instance *of t.*

Example 15 The term $f(x, y, z)$ matches $f(a, g(x), h(z))$ since $(f(x, y, z))\,\sigma = f(a, g(x), h(z))$ for the substitution $\sigma = \{x \mapsto a,\ y \mapsto g(x),\ z \mapsto h(z)\}$. Obviously, $f(x, y, z)$ does not match $g(x)$. ♠

Exercise 45

1. *Does $g(x)$ match $h(g(a))$? Why/why not?*
2. *Does $f(x, x)$ match $f(a, b)$?*
3. *Does $f(x, y)$ match $f(g(a), g(a))$?*
4. *Does $f(x, x)$ match* any subterm *of $f(g(f(a, a)), f(a, z))$?*
5. *How many subterms of $f(f(a, a), f(a, a))$ are matched by $f(x, x)$?*
6. *Does* `s(`x`) + `y *match any subterm of* `s(s(0 + s(0))) + 0`*?*
7. *Does* `0 +` x *match any subterm of* `s(s(0 + s(0))) + 0`*?*

3.1.4 The Reduction Relation

We now have all the notions needed to define what is a *reduction step* in a specification. It is the application of an equation $l = r$ to a term t, so that l matches some subterm of t (which could be t itself or a *proper* subterm of t), and that part of t is replaced by the appropriate instance of r. For example, if $g(x) = h(x)$ is an equation, then $f(a, g(b))$ reduces in one step to $f(a, h(b))$.

Definition 16 (Reduction relation) *Given a set of equations E (where each equation is "directed" from left to right). Then a term t* reduces *(in one step) to a term u, written $t \rightsquigarrow_E u$ if and only if there is an equation $l = r$ in E, a position p in t, and a substitution σ such that $t|_p = l\sigma$, such that u is $t[r\sigma]_p$. That is, $t = t[l\sigma]_p \rightsquigarrow_E t[r\sigma]_p = u$. (We will write $t \rightsquigarrow u$ instead of $t \rightsquigarrow_E u$ when E is given by the context or is unimportant.)*

The above definition was just the hard way to say the fairly obvious thing.

Example 16 If $E = \{f(x, y, z) = g(y)\}$, then $f(a, b, b) \rightsquigarrow g(b)$ and $h(g(b), f(a, g(x), h(z))) \rightsquigarrow h(g(b), g(g(x)))$. ♠

[2]There is a slight confusion surrounding the use of the word "matching," which is sometimes used in the opposite way. I follow the usage in [2].

Example 17 In the module `NAT-ADD` we have e.g. the reduction steps

$$\texttt{s(s(0 + s(0))) + 0} \rightsquigarrow \texttt{s(s(s(0))) + 0}$$

and

$$\texttt{s(s(0 + s(0))) + 0} \rightsquigarrow \texttt{s(s(0 + s(0)) + 0).}$$

♠

Exercise 46

1. *For each of the four reduction steps in Examples 16 and 17, find the equation, the position, and the match (substitution) used, and show that each step is indeed a reduction step according to Definition 16.*
2. *Given the above equation $\{f(x, y, z) = g(y)\}$, how many possible reduction steps are there from the term $g(h(f(b, g(f(a, b, f(a, b, c))), f(f(a, a, a), b, c))))$?*
3. *How many possibilities are there to reduce the term $h(f(b, g(f(a, b, f(a, b, c))), f(f(a, a, a), b, c)))$ if the equation is instead $f(x, x, y) = c$?*

3.1.5 Some Derived Relations

For any binary relation R, and in particular for the relation $\rightsquigarrow_E$ we define

- $\leftsquigarrow_E$ such that $t \leftsquigarrow_E u$ holds if and only if $u \rightsquigarrow_E t$ holds[3]
- $\leftrightsquigarrow_E$ such that $t \leftrightsquigarrow_E u$ holds if and only if $t \rightsquigarrow_E u$ or $u \rightsquigarrow_E t$ (or both) hold[4]
- $\overset{*}{\rightsquigarrow}_E$ such that $t \overset{*}{\rightsquigarrow}_E u$ holds if and only if t reduces to u in 0, 1, or more steps. That is, either t and u are the same term, or $t \rightsquigarrow_E u$, or there are n terms $t_1, \ldots, t_n$ such that $t \rightsquigarrow_E t_1 \rightsquigarrow \cdots \rightsquigarrow_E t_n \rightsquigarrow u$.[5]
- The relation $\overset{*}{\leftrightsquigarrow}_E$ can be defined similarly; $t \overset{*}{\leftrightsquigarrow}_E u$ holds if and only if there is a "path" from t to u using $\leftrightsquigarrow_E$-steps.[6]
- The relation $\overset{+}{\rightsquigarrow}_E$ (and analogously $\overset{+}{\leftrightsquigarrow}_E$) is defined by $t \overset{+}{\rightsquigarrow}_E u$ if and only if t reduces to u in one or more steps.[7] The difference between $\overset{*}{\rightsquigarrow}$ and $\overset{+}{\rightsquigarrow}$ is that $\overset{*}{\rightsquigarrow}$ also allows a "zero-step" derivation (where t must then be the same as u).

Example 18 In the module `NAT-ADD` we have both $\texttt{s(s(0 + s(0))) + 0} \overset{*}{\rightsquigarrow} \texttt{s(s(s(0)))}$ and $\texttt{s(s(0 + s(0))) + 0} \overset{+}{\rightsquigarrow} \texttt{s(s(s(0)))}$. ♠

Exercise 47 *Explain why we have* $\texttt{s(s(0 + s(0))) + 0} \overset{*}{\leftrightsquigarrow} \texttt{s(0) + s(s(0))}$ *in* `NAT-ADD`.

[3] This is called the *inverse* relation.
[4] This is called the *symmetric closure* of $\rightsquigarrow_E$.
[5] This is called the *reflexive-transitive closure* of $\rightsquigarrow_E$.
[6] This is the reflexive-symmetric-transitive closure of $\rightsquigarrow_E$.
[7] This is the *transitive closure* of $\rightsquigarrow_E$.

3.2 Operational Properties: Overview

In this section we formally define the crucial properties mentioned in the introduction. First we introduce some terminology:

- t is *reducible* if there is a term u such that $t \rightsquigarrow u$.
- t is *irreducible* (or *is in normal form*) if and only if t is not reducible.
- u is a *normal form* of t if and only if $t \stackrel{*}{\rightsquigarrow} u$ and u is irreducible. If u is the *unique* (that is, the *only*) normal form of t, we write $t!$ for this unique normal form u.
- u is a *successor* of t if and only if $t \stackrel{+}{\rightsquigarrow} u$.
- A *derivation*, or *reduction sequence*, in a specification E is a finite sequence of the form

  $$t_1 \rightsquigarrow_E t_2 \rightsquigarrow_E \cdots \rightsquigarrow_E t_n$$

  or an infinite sequence of the form

  $$t_1 \rightsquigarrow_E t_2 \rightsquigarrow_E t_3 \rightsquigarrow_E \cdots$$

  of reduction steps $t_i \rightsquigarrow_E t_{i+1}$ in E.
- A *computation* in an equational specification E is either an infinite derivation in E, or a finite derivation in E which cannot be extended (that is, the last term in the derivation is irreducible).

3.2.1 Termination

The first crucial property mentioned in the introduction to this chapter is *termination*:

Definition 17 (Termination) *A specification E is* terminating *if and only if there is no infinite derivation in E.*

Unfortunately it is *undecidable* whether a specification is terminating. This means that there is no (terminating!) method/algorithm/procedure of the form

```
bool terminates (specification E) {
  ...
  if <E is terminating> return true; else return false;
}
```

Should we just give up reasoning about termination and just write specifications and hope that they are terminating? Oh no! Even though there is no single method which works for *all* specifications, there are techniques which can *prove* termination of *many* terminating specifications. In Section 3.3 we will show some techniques for proving termination.

3.2.2 Each Term has a Unique Normal Form

The result of a computation should *not* dependent on Maude's evaluation strategy. Therefore, each term t should have a *unique normal form*. This is the case when the specification is terminating and *confluent*.

Definition 18 (Confluence) *A specification is* confluent *if and only if for any terms t, t_1, t_2 where $t \overset{*}{\rightsquigarrow} t_1$ and $t \overset{*}{\rightsquigarrow} t_2$, there is a term u such that $t_1 \overset{*}{\rightsquigarrow} u$ and $t_2 \overset{*}{\rightsquigarrow} u$.*

Exercise 48 *Example 17 showed that the term* `s(s(0 + s(0))) + 0` *reduces to (at least) two different terms in* `NAT-ADD`.

1. *Show that each possible computation from* `s(s(0 + s(0))) + 0` *in* `NAT-ADD` *leads to the same value (which is . . .).*
2. *Have you now proved that* `NAT-ADD` *is confluent?*
3. *Exactly how does Maude reduce the term* `s(s(0 + s(0))) + 0`*? (*Hint: *Set Maude's trace facilities* `on` *(see Appendix B) and then use the* `red` *command.)*
4. *Is the specification $\{ f(f(x)) = g(x) \}$ confluent? (*Hint: *Check the term $f(f(f(x)))$.)*

Proposition 1 *Each term t has a unique normal form in a specification which is terminating and confluent.*

Proof. Assume that the proposition does not hold. If this assumption leads to a contradiction, then the proposition must hold. If the proposition does not hold, then there are at least two *distinct normal forms* u_1 and u_2 of t. Since they are normal forms, we have $t \overset{*}{\rightsquigarrow} u_1$ and $t \overset{*}{\rightsquigarrow} u_2$. But then, according to the definition of confluence, there must be a term u such that $u_1 \overset{*}{\rightsquigarrow} u$ and $u_2 \overset{*}{\rightsquigarrow} u$. This is impossible unless $u_1 = u = u_2$, since u_1 and u_2 are both normal forms and therefore irreducible. ♣

It is therefore enough to prove termination and confluence to prove that each term has a unique normal form. Although it is undecidable whether a specification is confluent, it is decidable whether a *terminating* specification is confluent. This means that there is a method for checking confluence of terminating specifications, and this method is presented in Section 3.4.

3.3 Termination

This section deals with some techniques to decide and prove whether a specification is terminating. We assume that each specification has at least one ground term. (The specification would be quite meaningless otherwise!)

Example 19 The system $\{f(x) = f(g(x))\}$ is nonterminating since there is an infinite derivation

$$f(x) \rightsquigarrow f(g(x)) \rightsquigarrow f(g(g(x))) \rightsquigarrow \cdots$$

The specification $\{f(x) = f(g(x)),\ f(x) = a\}$ is also nonterminating since it allows the same infinite sequence as above.[8] ♠

□ Since we have assumed that each sort has at least one ground term, we don't need to consider infinite sequences of terms with variables, since we could just replace x in the infinite derivation above by a ground term t, getting another infinite derivation

$$f(t) \rightsquigarrow f(g(t)) \rightsquigarrow f(g(g(t))) \rightsquigarrow \cdots$$

Therefore, a specification is terminating if it does not allow an infinite derivation

$$t_1 \rightsquigarrow t_2 \rightsquigarrow t_3 \rightsquigarrow \cdots$$

of *ground* terms $t_1, t_2, t_3, \ldots$ □

Example 20 The specification $\{0 + x = x,\ s(x) + y = s(x + y)\}$ is terminating. However, you may need some time to convince yourself and others that it is indeed terminating. After reading this section, you should be able to quickly *prove* that this specification terminates. ♠

While termination may be the most crucial property of equational specifications, it is not always easy to see whether a specification terminates. For example, is

$$\{f(g(x, y)) = g(g(f(f(x)), y), y)\}$$

terminating? And how about

$$\{f(g(g(x))) = f(f(g(f(g(f(x)))))),\ f(f(f(x))) = f(g(f(x)))\}$$

It would of course be very desirable to have an algorithm which for any given E can figure out whether E terminates. Unfortunately, no such algorithm can exist:

Theorem 1 *It is undecidable whether a specification is terminating (even if it has only one equation).*

The theorem follows from the famous result in logic which says that it is undecidable whether a *Turing machine* is terminating. Any Turing machine can be simulated by an equational specification so that the Turing machine is terminating if and only if the simulation is terminating.

As mentioned above, termination is too important to just give up. Even though we cannot *always* decide whether a specification is terminating, we can in many cases

- discover (and therefore prove) that a specification is nonterminating by finding an infinite derivation, or

[8] The second specification is *normalizing* since each term has normal form(s) (which have the form $g(\ldots(g(a))\ldots)$ for zero or more g's).

- *prove* that a specification is terminating for all input/initial terms.

Exercise 49 *What are the two (or three) main reasons why it is impossible to prove termination of a specification by just running test cases and see whether the execution of these test cases terminate?*

This section on termination is, as mentioned, mostly based on Dershowitz' [21]. Chapter 2 in [76] gives a treatment of termination in Norwegian. So does [55] in English.

Before you continue, you may take some minutes to ponder on which of the specifications below are terminating. For the terminating ones, we will later *prove* that they are indeed terminating.

Exercise 50 *Ponder on whether the following specifications are terminating.*

1. $\{f(f(x)) = f(x)\}$
2. $\{f(x) = f(f(x))\}$
3. $\{f(x) = g(x),\ g(b) = f(a)\}$
4. $\{f(x) = g(x),\ g(b) = f(a),\ a = b\}$
5. $\{f(x, y) = f(y, x)\}$ *(Commutativity of f)*
6. $\{f(g(x)) = g(f(x))\}$
7. $\{f(a, b, x) = f(x, x, x)\}$
8. $\{g(x, y) = x,\ g(x, y) = y\}$
9. *(Difficult?)* $\{f(a, b, x) = f(x, x, x),\ g(x, y) = x,\ g(x, y) = y\}$ *(This is the union of the two above, and note that the two systems do not share any function symbols.)*
10. $\{f(s(y), x, z) = f(y, x + y + z, z + z)\}$
11. $\{f(x, s(y), z) = f(x + y + z, y, z + z)\}$
12. $\{f(x) = g(x)\}$
13. $\{h(f(f(x))) = h(f(g(f(x))))\}$
14. *(Difficult?)* $\{f(g(x, y)) = g(g(f(f(x)), y), y)\}$
15. $\{x * (y + z) = (x * y) + (x * z)\}$ *(Distributivity)*
16. *The Ackermann function:*

$$\begin{aligned} \{\ & ack(0, x) = s(x), \\ & ack(s(x), 0) = ack(x, s(0)), \\ & ack(s(x), s(y)) = ack(x, ack(s(x), y))\} \end{aligned}$$

17. *(Difficult?)* $\{f(g(g(x))) = f(f(g(f(g(f(x))))))),\ f(f(f(x))) = f(g(f(x)))\}$

18. $\{f(a,b) = f(b,a)\}$

19. $\{g(x,a) = g(b,x)\}$

20. $\{h(x,a,x) = h(a,b,x)\}$

21. *The union of the last three:*

$$\begin{aligned}\{\, f(a,b) &= f(b,a),\\ g(x,a) &= g(b,x),\\ h(x,a,x) &= h(a,b,x)\}\end{aligned}$$

3.3.1 Nontermination

A specification E is *looping* if there exists terms t and u such that $t \overset{+}{\rightsquigarrow}_E u$ and t is a subterm of u. Obviously, a looping specification is nonterminating.

Example 21

- In the specification $\{f(x) = f(f(x))\}$ there is a reduction $f(x) \rightsquigarrow f(f(x))$ which is a looping derivation since $f(x)$ is a subterm of $f(f(x))$. Therefore the specification is nonterminating:

$$f(x) \rightsquigarrow f(f(x)) \rightsquigarrow f(f(f(x))) \rightsquigarrow f(f(f(f(x)))) \rightsquigarrow \cdots$$

- In the system $\{f(x,y) = f(y,x)\}$ we have

$$f(x,y) \rightsquigarrow f(y,x) \rightsquigarrow f(x,y)$$

  and therefore a looping, and hence a nonterminating, specification, in which the above steps can be repeated.

- The system $\{f(x) = g(x),\ g(b) = f(a),\ a = b\}$ is looping and therefore nonterminating since there is a looping derivation

$$f(b) \rightsquigarrow g(b) \rightsquigarrow f(a) \rightsquigarrow f(b).$$

♠

Exercise 51 *Show that the specification*

$$\{f(a,b,x) = f(x,x,x),\ g(x,y) = x,\ g(x,y) = y\}$$

is nonterminating.

Hint: *Start with the term* $f(g(a,b), g(a,b), g(a,b))$.

If the righthand side of an equation contains a variable which is not in the lefthand side, then the specification is always nonterminating since the new variable can be instantiated with anything, including the term being reduced. Therefore, no equation should introduce a (new) variable in its righthand side.

Example 22 In the specification $\{f(x) = g(x, y)\}$ there is an infinite (and looping) derivation

$$f(x) \rightsquigarrow g(x, f(x)) \rightsquigarrow g(x, g(x, f(x))) \rightsquigarrow \cdots$$

♠

Just to make the picture more complicated, there are also nonterminating systems which are not looping.

Example 23 The specification $\{f(x) = f(g(x))\}$ is not looping, but is nonterminating:

$$f(x) \rightsquigarrow f(g(x)) \rightsquigarrow f(g(g(x))) \rightsquigarrow f(g(g(g(x)))) \rightsquigarrow \cdots$$

♠

3.3.2 Proving Termination

The following sections describe some techniques and methods for proving termination of many (terminating!) specifications. Since the problem of figuring out whether a specification is terminating is undecidable, there is no single algorithm (or finite set of algorithms) which can be used. This means that

- we are probably not able to prove termination of some terminating systems, and/or
- sometimes we must use some ingenuity ("smart ideas") to prove termination.

In Section 3.3.3 we show a range of "simple" and some ingenious techniques to prove termination. These techniques may require some thinking and imagination. In Section 3.3.4, we describe the *path orderings* which are fairly powerful and can be used "automatically" without thinking.

3.3.3 "Weight-Based" Techniques for Proving Termination

The main idea of proving termination is to assign to each ground term t a "weight" $w(t)$ which can be e.g. a natural number. The specification is terminating if one can show that for each possible reduction $t \rightsquigarrow u$ we have $w(t) > w(u)$. That is because all weights $w(t)$ are nonnegative numbers, and if there were an infinite derivation

$$t_0 \rightsquigarrow t_1 \rightsquigarrow t_2 \rightsquigarrow t_3 \rightsquigarrow \cdots$$

then there would also be an infinite sequence

$$w(t_0) > w(t_1) > w(t_2) > w(t_3) > \cdots$$

of decreasing *natural* numbers. This is clearly impossible, no matter how big $w(t_0)$ is. So we need to find a *weight function* $w : \mathcal{T}_\Sigma \to \mathbf{N}$ such that $t \rightsquigarrow u$ implies $w(t) > w(u)$. A weight function is often called a *progress function* since it measures the "progress" of each step.

Apart from finding an appropriate progress function w, the problem with this method is that there are often an *infinite* number of reductions $t \rightsquigarrow u$ to consider.

Example 24 To use the "progress function method" to prove that $\{f(x) = g(x)\}$ is terminating we need to find a progress function w such that $w(f(a)) > w(g(a))$, $w(f(g(a))) > w(g(g(a)))$, $w(f(f(a))) > w(f(g(a)))$, $w(f(f(a))) > w(g(f(a)))$, ... ♠

To avoid having to consider all *contexts*, we want w to be *monotonic*, which means that for each function symbol f in Σ,

$$w(t) > w(u) \text{ implies } w(f(\ldots, t, \ldots)) > w(f(\ldots, u, \ldots))$$

for all lists ... of ground terms. When the weight function w is monotonic, it is enough to prove that

$$w(l\sigma) > w(r\sigma)$$

holds for each ground substitution σ and each equation $l = r$ in the specification to prove that the specification is terminating.

Example 25 Let's consider the specification $\{f(x) = g(x)\}$ again. Let $w(t)$ be "the number of occurrences of f in t." Obviously, the range of w is the set of natural numbers (we can't have -2 occurrences of f). To prove termination we need to prove that

1. w is monotonic, and
2. $w(f(x)\sigma) > w(g(x)\sigma)$ for each ground substitution σ.

Is w monotonic? Assume $w(t) > w(u)$. Then we need to prove that $w(f(t)) > w(f(u))$ and that $w(g(t)) > w(g(u))$. Since $f(t)$ has one more occurrence of f than t, we have that $w(f(t)) = 1 + w(t)$ and that $w(f(u)) = 1 + w(u)$. Since $w(t) > w(u)$ we have that

$$w(f(t)) = 1 + w(t) > 1 + w(u) = w(f(u))$$

which is what we wanted. Monotonicity w.r.t. g is easier: $w(g(t)) > w(g(u))$ follows from the assumption $w(t) > w(u)$ since $w(g(t)) = w(t)$ and $w(g(u)) = w(u)$.

For the second property, let $\sigma = \{x \mapsto t\}$ for any ground term t. Then we need to prove

$$w(f(x)\sigma) = w(f(t)) = 1 + w(t) > w(t) = w(g(t)) = w(g(x)\sigma)$$

which is trivial.

Therefore, we have proved that the specification is terminating. ♠

Example 26 The system $\{f(f(x)) = f(x)\}$ is terminating. We can again use $w(t) =$ "the number of f's in t," or $w'(t) =$ "the number of function symbols in t." This second progress function just counts all the symbols in a term and is also a good progress function for some specifications. ♠

Exercise 52 *Prove that the specification*

$$\begin{aligned}\{\ & f(g(h(x))) = g(g(x)),\\ & g(h(f(x))) = h(h(x)),\\ & h(f(g(x))) = f(f(x))\ \}\end{aligned}$$

is terminating.

Sometimes the simplest options don't work, as illustrated by the following examples and exercises:

Example 27 The specification

$$\{\ f(x) = g(x),\ g(b) = f(a)\ \}$$

is terminating (why?), but we cannot use progress functions such as "the number of f's in a term," "the number of g's in a term," or "the number of function symbols in a term," since none of these would decrease in both equations. Instead, let's try

$$w(t) = (2 * \text{"the number of } b\text{'s in } t\text{"}) + \text{"the number of } f\text{'s in } t\text{"}$$

Is w monotonic? We assume $w(t) > w(u)$ and must prove $w(f(t)) > w(f(u))$ and $w(g(t)) > w(g(u))$, which both hold (why?).

Next we have to prove that each equation application reduces the weight:

- $w(f(x)\sigma) > w(g(x)\sigma)$, and
- $w(g(b)) > w(f(a))$.

The first of these holds since

$$w(f(x)\sigma) = 1 + w(x\sigma) > w(x\sigma) = w(g(x)\sigma).$$

The second inequality holds since

$$w(g(b)) = 2 > 1 = w(f(a)).$$

♠

Exercise 53 *(Slightly tricky?) Consider the specification* $\{f(g(x)) = g(f(x))\}$.

- *What is the (unique) normal form of* $g(f(f(g(f(g(a))))))$*?*
- *Argue informally why the specification is terminating.*
- *Prove formally that the specification is terminating.*

  Hint*: The same symbols appear on both sides of the equation. You must therefore somehow use that "f before g" weighs more than "g before f."*

  Hint 2*: Let the smallest weight of any term be* ≥ 1 *and remember that* $(2 \cdot n)^3 > 2 \cdot (n^3)$ *for any natural number* $n \geq 1$.

Example 28 For the specification $\{ f(h(x,y)) = h(x,x) \}$ it could be tempting to use the "size" function to prove termination. That is a bad idea since

$$f(h(f(f(f(f(f(a))))),b)) \rightsquigarrow h(f(f(f(f(f(a))))),f(f(f(f(f(a))))))$$

so the size can increase in a reduction step. Similarly, just counting the f's don't help much either. Therefore, it seems somewhat difficult to prove termination of this system (see the following exercise). (However, with some techniques presented later it will be trivial to prove its termination.) ♠

Exercise 54 *In Example 28 above, what's wrong with the following progress functions:*

1. $w(t) =$ *"size of* t*"*
2. $w(t) =$ *"number of* f*'s in* t*"*
3. $w(a) = 2,\ w(f(t)) = 5 \cdot w(t),\ w(h(t,u)) = 1 + w(t)$
4. *As the previous, but with* $w(h(t,u)) = 0$

Exercise 55 *(Difficult?) Can you try to prove termination of the system in Example 28? (Don't spend too much time on it. I have no simple solution myself at the moment.)*

Example 29 The system $\{h(f(f(x))) = h(f(g(f(x))))\}$ is terminating even though a reduction step increases the size of the term. A good progress function is

$$w(t) = \text{"the number of "adjacent" pairs of } f\text{'s in } t\text{"}.$$

(Adjacent = next to each other.) ♠

Sometimes it is not sufficient to use the natural numbers. Instead of using the natural numbers as the codomain of the progress function and using $>$ for comparison, we can use any set S and *well-founded strict partial ordering* $\succ$ on S, and have a progress function $w : \mathcal{T}_\Sigma \to S$ which satisfies $t \rightsquigarrow u$ implies $w(t) \succ w(u)$.

Definition 19 (Well-founded strict partial ordering) *A* strict partial ordering $\succ$ *over a set* S *is a relation* $\succ \subseteq S \times S$ *which is*

- *irreflexive:* $\neg(\exists s \in S)\ s \succ s$
- *transitive:* $(\forall s_1, s_2, s_3 \in S)\ (s_1 \succ s_2 \wedge s_2 \succ s_3 \Longrightarrow s_1 \succ s_3)$

A strict partial ordering $\succ$ *is* well-founded *over* S *if there is* no *infinite sequence*

$$s_1 \succ s_2 \succ s_3 \succ \cdots$$

of S*-elements* $s_1, s_2, s_3, \ldots$

Exercise 56 *Show that the greater-than relation $>$ over the natural numbers is a strict partial ordering, and that it is well-founded over the natural numbers. Is $>$ also a strict partial ordering over the integers? Is it well-founded over the integers?*

The lexicographic comparison $>^{lex}$ is well-founded over $\mathbf{N} \times \mathbf{N}$, where $(m_1, m_2) >^{lex} (n_1, n_2)$ if either $m_1 > n_1$ or ($m_1 = n_1$ and $m_2 > n_2$). The lexicographic comparison $>^{lex}$ can be extended to compare tuples of length 3, 4, ...

In general, if $\succ$ is well-founded over a set S, then so is the lexicographic extension $\succ^{lex}$ over S^n.

Example 30 Let's consider the specification

$$\begin{array}{l}\{\, f(g(g(x))) = f(f(g(f(g(f(x)))))),\\ \quad f(f(f(x))) = f(g(f(x)))\,\}\end{array}$$

The first equation needs a tricky progress function. Probably $w(t) =$ "the number of adjacent pairs of g's in t." However, the second equation will not reduce the weight with this function. Instead we use pairs of natural numbers as domain of the progress function, and compare them lexicographically:

$$w(t) = (\text{"number of pairs of adjacent } g\text{'s in } t\text{"}, \text{"number of } f\text{'s in } t\text{"})$$

It is fairly easy to see that w is monotonic and that

$$w(l\sigma) >^{lex} w(r\sigma)$$

for each ground substitution σ and for each of the two equations $l = r$ in the specification. Therefore, since $>^{lex}$ is well-founded over $\mathbf{N}\times\mathbf{N}$, the system has been proved to be terminating. ♠

Exercise 57 *Hopefully we can adapt these techniques to prove termination of imperative programs.*

1. *Explain why the following program terminates for any* `m` *and* `n`*:*

```
int x := m;   int y := n;
while (x>2 and y>0) {
  if x>y then {x := x-1;  y := x+y;}
  else y := y/2;
}
```

2. *(Slightly tricky?) Explain why the following "Euclidean" algorithm (adapted from [17]) for computing the greatest common divisor of two natural numbers terminates for all* `m` *and* `n`.[9]

[9] m `%` n gives the remainder when m is divided by n.

```
int gcd(int m, int n) {   // m,n > 0
  int x := m;   int y := n;
  int r := x % y;
  while (r>0) {
    x := y;
    y := r;
    r := x % y;
  }
  return y;
}
```

□ A well-founded strict partial ordering $\succ$ over S can be extended to a well-founded strict partial ordering $\succ^{ms}$ over finite multisets of S, where $m_1 \succ^{ms} m_2$ holds if m_1 has the greatest element when all the elements m_1 and m_2 have in common have been removed. □

A *termination ordering* is a well-founded strict partial ordering $\succ$ on the set $\mathcal{T}_\Sigma$ of ground terms. Every weight function w which can be used to prove termination also induces a termination ordering $\succ$ defined by $t \succ u$ if and only if $w(t) > w(u)$.

Exercise 58 *Let E be a terminating specification.*

1. *Show that $\overset{+}{\rightsquigarrow}_E$ is a termination ordering.*
2. *Explain why neither $\rightsquigarrow_E$ nor $\overset{*}{\rightsquigarrow}_E$ is a termination ordering.*

3.3.4 Simplification Orderings

The above proofs of termination required some more or less brilliant ideas for the progress function, and are therefore not suitable for "automatic" and automated proofs of termination. In this section we introduce some *simplification orderings* which are fairly strong and which can be applied without any thinking.

The notion of simplification ordering is due to Dershowitz (see e.g. [20]). Simplification orderings are not directly based on well-founded orderings[10] and progress functions. Instead they are based on a characteristic feature of all nonterminating systems. We saw earlier that "looping" does *not* characterize all nonterminating specifications. (That is, by showing that a system has no looping derivation we have *not* proved that it is terminating.)

To define a characteristic feature of nonterminating specifications we need the notion of *embedding* ("inneslutning" in Norwegian?). Intuitively, a term t embeds a term u if u is "inside" t, in the sense that if we remove some function symbols from t we get u.

□

[10] but orderings that are *well-founded for derivations*

Definition 20 (Embedding) *A ground term* $t = f(t_1, \ldots, t_m)$ embeds *a ground term* $u = g(u_1, \ldots, u_n)$ *(for* $m, n \geq 0$*)*[11]*, denoted*

$$t \trianglerighteq u$$

if either there is an $i \leq m$ *such that* $t_i \trianglerighteq u$*, or* $f = g$ *and* $t_i \trianglerighteq u_i$ *for each* $1 \leq i \leq m$*.*

□

Example 31 We have $f(g(f(a))) \trianglerighteq f(f(a))$ since $f(f(a))$ is inside $f(g(f(a)))$, in the sense that we can get $f(f(a))$ by removing a g from $f(g(f(a)))$. Formally, $f(g(f(a))) \trianglerighteq f(f(a))$ since (second case) $g(f(a)) \trianglerighteq f(a)$ since (first case) $f(a) \trianglerighteq f(a)$ since (second case) $a \trianglerighteq a$ since (second case) $a = a$. ♠

□ A possibly more intuitive definition of embedding is given by $t \trianglerighteq u$ if and only if $t \overset{*}{\rightsquigarrow}_{EMB} u$ in the specification EMB given by

$$EMB \;=\; \{f(x_1, \ldots, x_m) = x_i \mid 1 \leq i \leq m\} \cup \{g(x_1, \ldots, x_n) = x_i \mid 1 \leq i \leq n\} \cup \ldots$$

for all non-constant function symbols $f, g, \ldots$. □

It may not be too surprising that some "patterns" must be repeated in an infinite sequence of ground terms constructed by a finite set of function symbols:

Theorem 2 (Kruskal's Tree Theorem) *If* Σ *has a finite set of function symbols, then any infinite sequence*

$$t_1, t_2, \ldots, t_j, \ldots, t_k, \ldots$$

of ground terms in $\mathcal{T}_\Sigma$ *contains two terms* t_j *and* t_k *with* $j < k$ *such that* $t_k \trianglerighteq t_j$*.*

The theorem implies that if a finite specification does *not* have any *self-embedding* derivation, i.e., a derivation of the form

$$t_1 \rightsquigarrow t_2 \rightsquigarrow \ldots \rightsquigarrow t_j \rightsquigarrow \ldots \rightsquigarrow t_k \rightsquigarrow \ldots$$

with $t_k \trianglerighteq t_j$ for some $k > j$, then it must be terminating! Indeed, if we have a strict partial ordering $\succ$ on $\mathcal{T}_\Sigma$ such that $t \rhd u$ (where $t \rhd u$ is defined by $(t \neq u \wedge t \trianglerighteq u)$) implies $t \succ u$, and $t \rightsquigarrow u$ implies $t \succ u$, then the specification is terminating! Why? Because if it did *not* terminate, there would be an infinite sequence

$$t_1 \succ t_2 \succ \ldots \succ t_j \succ \ldots \succ t_k \succ \ldots$$

By Kruskal's Theorem, $t_k \trianglerighteq t_j$, and therefore either $t_k = t_j$ or $t_k \rhd t_j$. The case $t_k = t_j$ is impossible, since we have that $t_j \succ t_k$ (because $\succ$ is transitive), and then it cannot be that $t_k = t_j$ because $\succ$ is irreflexive. The case $t_k \rhd t_j$ is also impossible: Since $t_k \rhd t_j$ implies $t_k \succ t_j$ (by the assumption on the definition of $\succ$), we have both $t_j \succ t_k$ and $t_k \succ t_j$. Since $\succ$ is transitive we get that $t_j \succ t_j$ which is impossible since $\succ$ is irreflexive.

[11] Don't forget that this definition also applies to constants since the symbol f is a constant when $m = 0$, and similarly for g.

Based on the above argument, the idea is to have a strict partial ordering $\succ$ which includes $\rhd$. So why not use $\rhd$ directly instead of some extension $\succ$? Because it is undecidable whether a specification is self-embedding.

Any strict partial ordering which extends $\rhd$ and which is monotonic (so that we don't have to worry about contexts) can therefore be used to prove termination:

Definition 21 (Simplification ordering) *A monotonic strict partial ordering $\succ$ is a* simplification ordering *if*

$$f(t_1, \ldots, t_n) \succ t_i$$

for all ground terms $f(t_1, \ldots, t_n)$ and each $i \leq n$.

Proposition 2 *Any simplification ordering extends the (strict part of the) embedding relation:*

$$t \rhd u \;\; \textit{implies} \;\; t \succ u$$

for all simplification orderings $\succ$ and all ground terms t and u.

The main result follows trivially from the above facts:

Theorem 3 *A specification with a finite number of function symbols and/or a finite number of equations is terminating if there is a simplification ordering $\succ$ such that $l\sigma \succ r\sigma$ holds for each ground substitution σ for each equation $l = r$ in the specification.*

Proof. First of all, it is easy to see (and to prove) that $l\sigma \succ r\sigma$ and the fact that $\succ$ is monotonic implies that $t \rightsquigarrow u \Longrightarrow t \succ u$, and, since $\succ$ is transitive, we also have that $t \overset{+}{\rightsquigarrow} u \Longrightarrow t \succ u$. Now, assume that the specification is *not* terminating. Then, there is an infinite reduction

$$t_0 \rightsquigarrow t_1 \rightsquigarrow \cdots \rightsquigarrow t_j \rightsquigarrow \cdots \rightsquigarrow t_k \rightsquigarrow \cdots .$$

All terms in this reduction are built from a *finite* set of function symbols. (If we have an infinite number of function symbols, but a finite set of equations, then all the terms in the above reduction are constructed from the function symbols in t_0 and from those function symbols which appear in the right-hand sides of the equations. Given a finite set of equations, we can only have a finite number of distinct function symbols in these right-hand sides.) Therefore, Kruskal's Theorem applies, and we have both $t_k \trianglerighteq t_j$ and $t_j \succ t_k$ (since $t_j \overset{+}{\rightsquigarrow} t_k \Longrightarrow t_j \succ t_k$). This is impossible ($t_k = t_j$ is impossible because $\succ$ is irreflexive, and $t_k \rhd t_j$ implies that $t_k \succ t_j$ by Proposition 2, and $\succ$ being a strict partial ordering we cannot have both $t_j \succ t_k$ and $t_k \succ t_j$)! ♣

So to have your own simplification ordering $\succ_{mine}$, just make sure that it is irreflexive, transitive, monotonic (i.e., $t \succ_{mine} u$ implies $f(\ldots, t, \ldots) \succ_{mine} f(\ldots, u, \ldots)$ for all t, u and function symbols f), and that it satisfies the *subterm property* $f(t_1, \ldots, t_n) \succ_{mine} t_i$ for all terms $f(t_1, \ldots, t_n)$. If you can then prove $l\sigma \succ_{mine} r\sigma$ for each equation $l = r$ and each ground substitution σ, then you have proved that your specification is terminating.

We have seen earlier that there are self-embedding specifications which are terminating (such as $\{f(f(x)) = f(g(f(x)))\}$). Simplification orderings essentially just search for self-embeddings and are therefore not able to prove termination of such terminating *and* self-embedding specifications.

Exercise 59 *In the previous subsection we showed that some of the specifications of Exercise 50 are terminating. Some of these systems are self-embedding yet terminating. Which ones?*

Hint: *(At least) find the terminating system(s) with "self-embedding" equation(s).*

Exercise 60 *(Adapted from an example in [21].) Sally from Merrittville*[12] *is given a specification with constants a, b, and c, a binary (infix) function symbol $*$, and an equation*

$$(x * y) * z = x * (y * z).$$

She suggests to prove termination of this system using a weight function

$$\mathit{weight} : \mathcal{T} \to \mathbf{R}_+,$$

which assigns a nonnegative real *number to each ground term as follows:*

- *the weight of a constant is 10^{-6};*
- *the weight$(t * u)$ of a ground term of the form $t * u$ is*

$$\sqrt{2} \cdot \mathit{weight}(t) + \mathit{weight}(u).$$

1. *Is $(\mathbf{R}_+, >)$ a well-founded strict partial ordering?*
2. *Is the equation weight-decreasing for all instances of it?*
3. *Sally claims that she has proved termination of the spedification since the ordering $\succ$ on ground terms defined by*

$$t \succ u \quad \textit{if and only if} \quad \mathit{weight}(t) > \mathit{weight}(u)$$

is well-founded, while each reduction is $\succ$-decreasing. Is Sally's argumentation correct? Explain!

In case you don't want to define your own simplification ordering, you can use some of the standard *path orderings*.

[12] from the Dream Syndicate classic "Merrittville"

Path Orderings

The *path orderings* form a particular class of simplification orderings which are fairly powerful and which can be applied automatically. The first path ordering we will look at is the *lexicographic path ordering* (lpo) [52]. It requires that you have an ordering $\succ$ on the function symbols in Σ. For example, $f \succ h \succ a \succ g \succ b$ could be such a *precedence*.

Definition 22 (Lexicographic path ordering) *Given an ordering $\succ$ on the function symbols in the specification, the* lexicographic path ordering $\succ_{lpo}$ *is defined as follows:*

lpo-1: If $t_i \succ_{lpo} u$ or $t_i = u$ for some t_i, then

$$f(\ldots, t_i, \ldots) \succ_{lpo} u$$

lpo-2: If $f \succ g$ and $f(t_1, \ldots, t_n) \succ_{lpo} u_i$ for all $i \leq m$, then

$$f(t_1, \ldots, t_n) \succ_{lpo} g(u_1, \ldots, u_m)$$

for $n, m \geq 0$

lpo-3: If $(t_1, \ldots, t_n) \succ_{lpo}^{lex} (u_1, \ldots, u_n)$ for $\succ_{lpo}^{lex}$ the lexicographic extension of $\succ_{lpo}$, and $f(t_1, \ldots, t_n) \succ_{lpo} u_i$ for each $i \leq n$, then

$$f(t_1, \ldots, t_n) \succ_{lpo} f(u_1, \ldots, u_n)$$

Furthermore, $\succ_{lpo}$ is the smallest relation satisfying the above criteria.

Note that if $n = 0$ or $m = 0$ above, then the respective function symbol (f or g) is a constant.

The lexicographic path ordering *can* be extended to terms with variables so that a variable is not comparable in $\succ$ to anything but itself. Then $l \succ_{lpo} r$ implies the desired $l\sigma \succ_{lpo} r\sigma$ for all substitutions σ.

Proposition 3 *The lexicographic path ordering $\succ_{lpo}$ is a simplification ordering for any precedence $\succ$.*

□ To convince yourself that lpo is a simplification ordering you have to make sure that

- $f(\ldots, t_i, \ldots) \succ_{lpo} t_i$ holds for all ground terms $f(\ldots, t_i, \ldots) \succ_{lpo} t_i$ and each $i \leq n$. This follows directly from rule lpo-1 in the definition of lpo.
- $\succ_{lpo}$ is monotone. That is, $t_i \succ_{lpo} u_i$ implies that $f(t_1, \ldots, t_i, \ldots, t_n) \succ_{lpo} f(t_1, \ldots, u_i, \ldots, t_n)$. This follows from lpo-3 since $(t_1, \ldots, t_i, \ldots, t_n) \succ_{lpo}^{lex} (t_1, \ldots, u_i, \ldots, t_n)$, and $f(t_1, \ldots, t_i, \ldots, t_n) \succ_{lpo} t_j$ and $f(t_1, \ldots, t_i, \ldots, t_n) \succ_{lpo} u_i$ are also fairly easy to prove.
- $\succ_{lpo}$ is irreflexive (fairly easy to see) and transitive (also somewhat easy to see).

□

Therefore, one way of proving termination of a finite[13] specification is to

[13] A finite specification in this case is one which contains a finite number of function symbols and/or a finite number of equations. This should be case for our Maude modules (with possible exception for some built-in modules).

define a precedence $\succ$ on the function symbols (and extend it to variables so that no variable is comparable in $\succ$ with any other symbol) such that $l \succ_{lpo} r$ holds for each equation $l = r$ in the specification.

It is helpful to choose a suitable precedence $\succ$. Often in specifications new functions are defined with the help of already defined functions. For example, for the natural numbers, multiplication ($*$) is defined in terms of addition ($+$), and exponentiation ($**$) is defined in terms of multiplication, etc. In these cases, termination can often be shown by choosing $\succ$ so that it satisfies $** \succ * \succ +$.

Example 32 We prove termination of

$$\begin{aligned} \{\ 0 + x &= x, \\ s(x) + y &= s(x + y), \\ 0 * x &= 0, \\ s(x) * y &= y + (x * y), \\ x ** 0 &= s(0), \\ x ** s(y) &= x * (x ** y)\ \} \end{aligned}$$

using $\succ_{lpo}$ induced by the precedence $** \succ * \succ + \succ s \succ 0$. To prove termination it is as mentioned sufficient to show that each equation is $\succ_{lpo}$-decreasing:

- $0 + x \succ_{lpo} x$ holds because of lpo-1 (the prefix form of $0 + x$ is $+(0, x)$).
- $s(x) + y \succ_{lpo} s(x + y)$ holds by lpo-2 since $+ \succ s$, and it therefore enough to prove $s(x) + y \succ_{lpo} x + y$. This latter is proved using lpo-3 since '+' is the main function symbol in both places. That is, we must prove $(s(x), y) \succ_{lpo}^{lex} (x, y)$ and $(s(x), y) \succ_{lpo} x$ and $(s(x), y) \succ_{lpo} y$. The latter two follows from lpo-1. $(s(x), y) \succ_{lpo}^{lex} (x, y)$ holds because $s(x) \succ_{lpo} x$ thanks to lpo-1.
- $0 * x \succ_{lpo} 0$ holds because of lpo-1.
- $s(x) * y \succ_{lpo} y + (x * y)$. Since $* \succ +$ we use rule lpo-2 and have to prove $s(x) * y \succ_{lpo} y$ and $s(x) * y \succ_{lpo} (x * y)$. The first of these holds by lpo-1. $s(x) * y \succ_{lpo} (x * y)$ holds by lpo-3 since $(s(x), y) \succ_{lpo}^{lex} (x, y)$ holds (proof above) and $s(x) * y \succ_{lpo} x$ and $s(x) * y \succ_{lpo} y$ hold by lpo-1.
- The last two equations: see the following exercise!

We have then proved that the specification is terminating.

♠

Exercise 61 *Show that the last two equations in the specification in Example 32 are $\succ_{lpo}$-decreasing with the given ordering $\succ$.*

(I have earlier promised the use of the lexicographic path ordering to be fully automatic, yet I write above that "it is helpful to choose a suitable precedence $\succ$." The lexicographic

path ordering *is* fully automatic since a finite set of function symbols can only have a finite number of precedences $\succ$ to check. We could therefore check for termination for each of these precedences. However, it is obviously much more efficient (and therefore helpful) not to have to try each of the precedences.)

Exercise 62 *Use lpo to prove termination of the following specifications, which we have earlier proved to be terminating:*

1. $\{f(f(x)) = f(x)\}$
2. $\{f(x) = g(x),\ g(b) = f(a)\}$
3. $\{f(g(x)) = g(f(x))\}$
4. $\{g(f(x)) = f(g(x))\}$

Exercise 63 *We saw earlier that*

$$\{\ h(f(f(x))) = h(f(g(f(x))))\ \}$$

is terminating. Why cannot lpo be used to prove its termination?

Exercise 64 *Why do we need the "extra" conditions $f(t_1, \ldots, t_n) \succ_{lpo} u_i$ in the rule lpo-3 in the definition of lpo? That is, give a nonterminating specification (one equation should be enough) whose equations are all $\succ_{lpo}$-decreasing* without *the "extra" condition in lpo-3.*

Exercise 65 *I was not able to prove termination of $\{\ f(h(x,y)) = h(x,x)\ \}$ using "smart idea"-techniques. Can you prove termination of this specification using lpo?*

Exercise 66 *Some of the specifications in Exercise 50 whose termination is not so easy to prove using "smart idea/progress function" techniques are*

1. $\{\ f(s(y), x, z) = f(y, x + y + z, z + z)\ \}$
2. $\{\ x * (y + z) = (x * y) + (x * z)\}$ *(maybe not so difficult using other techniques either?)*
3. *The Ackermann function:*

$$\begin{gathered}\{\ ack(0,x) = s(x),\\ ack(s(x),0) = ack(x,s(0)),\\ ack(s(x),s(y)) = ack(x,ack(s(x),y))\}\end{gathered}$$

Use lpo to prove that these specifications are terminating.

Exercise 67 *Is there a simplification ordering which can prove that the terminating specification*

$$\{f(a,b,x) = f(x,x,x)\}$$

is terminating?

Hint: *We have in Exercise 51 seen that $\{f(a,b,x) = f(x,x,x),\ g(x,y) = x,\ g(x,y) = y\}$ is nonterminating.*

Exercise 68

1. *Show that the ordering $\succ_{lpo}$ is not necessarily well-founded for signatures which contain infinitely many function symbols. (The definition of a signature does not require the set of function symbols to be a finite set.)*

2. *Show that lpo, despite the fact above, can be used prove termination of specification with an infinite number of function symbols, but with a finite number of equations.*

Multiset Path Ordering

In the case lpo-3 in the definition of lpo we have that $f(t_1, \ldots, t_n) \succ_{lpo} f(u_1, \ldots, u_n)$ if $(t_1, \ldots, t_n) \succ_{lex}^{lex} (u_1, \ldots, u_n)$ and .... That is, we compare the subterms *lexicographically*. The *multiset path ordering* (mpo) $\succ_{mpo}$ is exactly the same as $\succ_{lpo}$ except that $(t_1, \ldots, t_n)$ and $(u_1, \ldots, u_n)$ are compared as multisets. That is, mpo has the rules mpo-1 and mpo-2 which are the same as lpo-1 and lpo-2, and mpo-3 is

mpo-3: If $\{t_1, \ldots, t_n\} \succ_{mpo}^{ms} \{u_1, \ldots, u_n\}$ (where $\succ_{mpo}^{ms}$ is the "multiset extension" of $\succ_{mpo}$), then

$$f(t_1, \ldots, t_n) \succ_{mpo} f(u_1, \ldots, u_n)$$

The orderings mpo and lpo are incomparable in strength. That is, there are specifications whose termination can be proved using lpo and not mpo, and vice versa.

Exercise 69 *Given the specifications*

$$E_1 = \{f(a, b) = f(b, a)\}$$

and

$$E_2 = \{g(x, a) = g(b, x)\}.$$

One of these can be proved terminating using lpo and not mpo, and vice versa. For each of the above specifications, prove termination using lpo or mpo, and explain why the other ordering cannot be used.

Combining and Extending mpo and lpo

□ The path orderings can be combined and extended in e.g. the following ways:

- In mpo-3/lpo-3 we can say that for a certain function symbol f we compare the arguments $(t_1, \ldots, t_n)$ and $(u_1, \ldots, u_n)$ *lexicographically*, while for another function symbol g we compare them by multiset comparison. Indeed, it is also possible to compare $(t_1, \ldots, t_n)$ and $(u_1, \ldots, u_n)$ lexicographically in *any fixed order*, say, by first comparing t_2 and u_2, and then t_5 and u_5, etc. However, for each function symbol f, the way we compare $(t_1, \ldots, t_n)$ and $(u_1, \ldots, u_n)$ must be same throughout the system. We cannot compare $f(t_1, t_2)$ and $f(u_1, u_2)$ first lexicographically, and then compare $f(t_1', t_2')$ and $f(u_1', u_2')$ by multisets for the same f.

- Sometimes it is convenient to allow two function symbols f and g to have the same precedence in $\succ$; that is, $f \approx g$. The cases lpo-3/mpo-3 are changed accordingly to

  $$f(t_1, \ldots, t_n) \succ_{lpo/mpo} g(u_1, \ldots, u_m)$$

  if $f \approx g$ and $((t_1, \ldots, t_n) \succ^{lex}_{lpo/mpo} (u_1, \ldots, u_m)$ or $\{t_1, \ldots, t_n\} \succ^{ms}_{lpo/mpo} \{u_1, \ldots, u_m\}$ or whatever ...). Note that terms are considered equivalent in the orderings and comparisons if they are equivalent up to $\approx$-equivalent function symbols.

Example 33 We can prove termination of the specification

$$\{\, f(x, s(y), z) = f(x + y + z, y, z + z) \,\}$$

by $\succ_{lpo}$ when for symbol f, the elements are compared lexicographically in the order 2. element, 1. element, and 3. element. By lpo-3 we then have

$$f(x, s(y), z) \succ_{lpo} f(x + y + z, y, z + z)$$

since $s(y) \succ_{lpo} y$, and $f(x, s(y), z) \succ_{lpo} x + y + z$ and $f(x, s(y), z) \succ_{lpo} z + z$ when $s \succ +$ and $f \succ +$. ♠

Exercise 70

1. *Show that*

   $$\{f(a) = g(b),\ g(a) = f(b),\ f(x) = a\}$$

   cannot be shown to be terminating in lpo or mpo if we cannot have function symbols with the same precedence in $\succ$.

2. *Use $\succ_{lpo}$ to prove termination of the above specification if two function symbols may have the same precedence in $\succ$.*

Exercise 71

1. *Use a combination of mpo and lpo to prove termination of*

   $$\begin{aligned} \{\, f(a, b) &= f(b, a), \\ g(x, a) &= g(b, x), \\ h(x, a, x) &= h(a, b, x) \,\} \end{aligned}$$

2. *Can you also use a combination of mpo and lpo to show that*

   $$\begin{aligned} \{\, f(a, b) &= f(b, a), \\ f(x, a) &= f(b, x) \,\} \end{aligned}$$

   is terminating?

□

3.4 Confluence: Unique Normal Forms

Goals/requirements: This section contains some necessary technicalities. In this course it is sufficient that you can figure out whether a given specification is confluent (including being able to find unifiers).

Let's recall the definition of *confluence*:

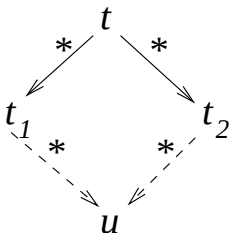

Figure 3.4: Confluence.

Definition 23 (Confluence) *A specification* (Σ, E) *is* confluent *if and only if for all terms* t, t_1, t_2 *with* $t \overset{*}{\rightsquigarrow} t_1$ *and* $t \overset{*}{\rightsquigarrow} t_2$*, there is a term* u *such that* $t_1 \overset{*}{\rightsquigarrow} u$ *and* $t_2 \overset{*}{\rightsquigarrow} u$*.*

Confluence means that if t can be reduced to two different terms t_1 and t_2 (for instance by applying different equations to t), we can always "join" t_1 and t_2 back by reducing both to a common term u. This property is shown in Fig. 3.4, where a solid arrow means "for all $\overset{*}{\rightsquigarrow}$" and a broken arrow means "there exists $\overset{*}{\rightsquigarrow}$".

Confluence is significant because, as we have seen, each term t has a unique normal form ("value") $t!$ when the specification is terminating and confluent. We will later show that confluence is undecidable. However, confluence is decidable when the specification is terminating, and that's all we need since we want our specifications to be terminating. This section presents an algorithm which you can use to check whether your specification is confluent. Therefore, you don't need any brilliant ideas to prove confluence.

Example 34 The specification $\{f(f(x)) = g(x)\}$ is not confluent since $f(f(f(x))) \rightsquigarrow f(g(x))$ and $f(f(f(x))) \rightsquigarrow g(f(x))$, and $g(f(x))$ and $f(g(x))$ cannot be reduced to some common element (in fact, they cannot be reduced at all). We will later be able to *prove* that by adding a new equation $f(g(x)) = g(f(x))$ we have a terminating and confluent system which is "logically equivalent"[14] to the original specification. ♠

Checking "directly" whether a specification is confluent by checking the confluence property for all t, t_1, t_2 with $t \overset{*}{\rightsquigarrow} t_1$ and $t \overset{*}{\rightsquigarrow} t_2$ is not possible, because there are infinitely many terms t to start with. Furthermore, there could be many terms t_1 and t_2 reachable from some start term t. We need something better.

In what follows we show that for each start term t, we only need to check the confluence property for terms t_1 and t_2 reachable in *one* reduction step from t.

Definition 24 (Local confluence) *A specification* E *is* locally confluent *if and only if for each* t *and all* t_1, t_2 *such that* $t \rightsquigarrow t_1$ *and* $t \rightsquigarrow t_2$*, there exists a term* u *such that* $t_1 \overset{*}{\rightsquigarrow} u$ *and* $t_2 \overset{*}{\rightsquigarrow} u$*.*

It is sufficient to check for local confluence instead of global confluence because of

[14] We do not destroy equivalence-properties by adding the equality $f(g(x)) = g(f(x))$ explicitly to the specification since $g(f(x)) = f(f(f(x))) = f(g(x))$.

Theorem 4 (Newman's Lemma) *A terminating specification is confluent if it is locally confluent.*

Proof. PENSUM?? ♣

The next step is to avoid having to check local confluence for an infinite number of "start" terms t, but just for a few terms. For that, we need the notion of *unification*.

3.4.1 Unification

Definition 25 (Unifier) *A* unifier *of two terms t and u is a substitution σ such that $t\sigma = u\sigma$.*

Example 35

- $f(x, h(b))$ and $f(h(y), z)$ have a unifier $\sigma = \{x \mapsto h(y),\ z \mapsto h(b)\}$. Of course, any instance of σ is also a unifier, such as $\sigma' = \{x \mapsto h(f(f(f(a, a), a), a)),\ z \mapsto h(b)\}$.
- The pair $f(g(x))$ and $f(h(z))$ has no unifier (why not?), neither has the pair $f(x)$ and $g(y)$, nor the pair $f(a)$ and $f(g(x))$, nor the pair $f(x)$ and $f(g(x))$. Why not?

♠

We saw in the example above that there can be many unifiers of the same pair of terms. We are interested in finding the *most general unifier* (mgu), which is a unifier ρ such that all other unifiers σ are "extensions" of ρ. That is, ρ is an mgu of a pair of terms if for each unifier σ of the pair, there is a substitution π such that $\sigma = \pi \circ \rho$.

Example 36 The substitution $\rho = \{x \mapsto h(y),\ z \mapsto h(b)\}$ is the mgu of the pair $f(x, h(b))$ and $f(h(y), z)$. Two other unifiers of the pair are $\sigma = \{x \mapsto h(f(f(f(a, a), a), a)),\ z \mapsto h(b)\}$ and $\sigma' = \{x \mapsto h(h(h(h(h(z))))),\ z \mapsto h(b),\ y \mapsto h(h(h(h(z))))\}$. Both σ and σ' are extensions of ρ as follows:

$$\sigma = \{y \mapsto f(f(f(a, a), a), a)\} \circ \rho$$
$$\sigma' = \{y \mapsto h(h(h(h(z))))\} \circ \rho$$

♠

Proposition 4 *If two terms have a unifier, then they have a most general unifier. Furthermore, the most general unifier is unique up to renaming of the variables.*

A *renaming* just changes the names of the variables in a term/equations/...so that the term/equation/...is "the same" just with other names.[15] For example, $f(x', y')$, $f(x, y)$, $f(y, x)$, and $f(x, z)$ are all renamed versions of $f(x, y)$, but $f(x, x)$ and $f(z, a)$ are not. Obviously, a renaming doesn't change the "logic" of a specification. The specification $\{f(x, y) = g(x),\ h(x, y) = f(x, y)\}$ is logically the same as $\{f(x, z) = g(x),\ h(x', y') = f(x', y')\}$.

[15] Formally, a renaming is a bijective substitution.

Fortunately it is fairly easy to find the mgu if two terms are unifiable, or to figure out that two terms are not unifiable. (If you think that you can find mgu's in small examples you may skip the following algorithm in a first reading.)

□ The algorithm for finding an mgu is as follows:

- The "data structure" of the algorithm is a pair of the form

$$(UP, \rho)$$

where UP is a *set* of *unification problems* of the form $t \stackrel{?}{=} u$, and ρ is the mgu we are constructing. Initially, UP is the unification problem we want to solve (find an mgu for) and ρ is the identity (that is, the substitution which "does nothing").

Then the algorithm proceeds by applying the following steps until it returns `<Not unifiable>` or the desired mgu:

1. If UP contains a unification problem of the form $f(t_1, \ldots, t_n) \stackrel{?}{=} g(u_1, \ldots, u_m)$ where $f \neq g$ then `return <Not unifiable>`. (Obviously there is no unifier for this unification (sub)problem.)
2. If UP has the form[16]

$$\{f(t_1, \ldots, t_n) \stackrel{?}{=} f(u_1, \ldots, u_n)\} \cup UP'$$

then we must find unifiers for $t_1 \stackrel{?}{=} u_1$, and $\ldots$, and $t_n \stackrel{?}{=} u_n$. That is,

$$UP := \{t_1 \stackrel{?}{=} u_1, \ldots, t_n \stackrel{?}{=} u_n\} \cup UP';$$

3. If UP contains a unification problem of the form $t \stackrel{?}{=} t$, then just remove this trivial unification problem from UP.
4. If UP contains a unification problem $x \stackrel{?}{=} t$ (or $t \stackrel{?}{=} x$) where x and t are different terms and x occurs in t, then `return <Not unifiable>`; (The terms x and $f(x)$ are for example not unifiable.)
5. If UP contains a unification problem of the form $x \stackrel{?}{=} t$ (or $t \stackrel{?}{=} x$) where x and t are (syntactically) different terms and x does not occur in t, then:
   - remove this unification problem from UP,
   - apply the substitution $\{x \mapsto t\}$ on all remaining unification problems in UP, and
   - apply the substitution $\{x \mapsto t\}$ on ρ (one effect will be that $x \mapsto t$ is "added" to ρ, since ρ does not contain an assignment of x and therefore has $x \mapsto x$).
6. If UP is empty, then `return` ρ; which is the desired mgu.

Example 37 Let's find the mgu of the pair $f(x, h(x))$ and $f(h(y), z)$ using the algorithm: We start with

$$(\{f(x, h(x)) \stackrel{?}{=} f(h(y), z)\}, Id)$$

where Id is the identity substitution $\{x \mapsto x, y \mapsto y, z \mapsto z\}$. Then

$$\begin{array}{ll}
(\{f(x, h(x)) \stackrel{?}{=} f(h(y), z)\}, Id) \implies & \text{(by step 2)} \\
(\{x \stackrel{?}{=} h(y),\ h(x) \stackrel{?}{=} z\}, Id) \implies & \text{(by step 5)} \\
(\{h(h(y)) \stackrel{?}{=} z\}, \{x \mapsto h(y)\}) \implies & \text{(by step 5)} \\
(\emptyset, \{x \mapsto h(y), z \mapsto h(h(y))\}) \implies & \text{(by step 6)} \\
\texttt{return } \{x \mapsto h(y), z \mapsto h(h(y))\} &
\end{array}$$

♠

[16] Strictly speaking, the union in the line below should be disjoint union so the "bigger" unification problem is actually removed in the step.

We state without proof that the given algorithm is correct and that it is terminating. □

Exercise 72 *(From [2]) Decide whether the following unification problems have unifiers, and if so, find the mgu:*

1. $f(x, y) \stackrel{?}{=} f(h(a), x)$
2. $f(x, y) \stackrel{?}{=} f(h(x), x)$
3. $f(x, b) \stackrel{?}{=} f(h(y), z)$
4. $f(x, x) \stackrel{?}{=} f(h(y), y)$

3.4.2 Critical Pairs

By checking local confluence instead of global confluence we only need to consider all t_1, t_2 reachable in *one* step from the initial term t to prove confluence of the specification. So far, however, this must be done for *all* terms t, which means that we still don't have an algorithm for checking for confluence. Obviously, the next step must be to restrict the set of initial terms t for which to check local confluence.

Let $l_i = r_i$ and $l_j = r_j$ be two equations in our specification (they could be the same equation!), and rename if necessary the variables in $l_j = r_j$ so that l_i and l_j do not share variables. Let p be a position in l_i so that $l_i|_p$ is *not* a variable. If $l_i|_p$ and l_j are unifiable with mgu ρ, then the term $l_i\rho$ may reduce to $r_i\rho$ (by applying $l_i = r_i$ at the top (position ϵ)). The term $l_i\rho$ may also reduce to $(l_i\rho)[r_j\rho]_p$ (by applying $l_j = r_j$ at position p). That is,

$$\begin{array}{ll} l_i\rho \rightsquigarrow r_i\rho & \text{and} \\ l_i\rho \rightsquigarrow (l_i\rho)[r_j\rho]_p & \end{array}$$

To check local confluence we need to check whether the *critical pair* $(r_i\rho,\ (l_i\rho)[r_j\rho]_p)$ is *joinable* (that is, whether there exists a term u such that $r_i\rho \stackrel{*}{\rightsquigarrow} u$ and $(l_i\rho)[r_j\rho]_p \stackrel{*}{\rightsquigarrow} u$). This has to be done for all pairs of equations, and for all positions, and then we have checked local confluence:

Theorem 5 (Critical Pair Lemma) *A specification is locally confluent if all critical pairs are joinable.*

There were many symbols in the description above. Just think of the check for confluence as follows:

1. Choose two (not necessarily distinct) equations from the specification and change the names of the variables in one of them (by e.g. replacing each x with an x') so that the two equations have no variable in common.

2. Check if the lefthand sides of the two equations overlap. That is, the top of l_j should "fit/unify with" some non-variable position in l_i. Perform the two reductions possible from this "overlap" and reduce the resulting two terms to their normal forms. If both reductions lead to the same normal form then repeat step 2 with a different position. If the reductions lead to different normal forms then the specification is not confluent and the algorithm exits.

3. Repeat step 1 until all pairs of equations have been checked.

4. If all pairs of equations and all overlap-positions within each such pair has been checked successfully, then the specification is confluent.

Example 38 Let's check whether $\{f(f(x)) = g(x)\}$ is confluent. The only pair of equations is $(f(f(x)) = g(x),\ f(f(x)) = g(x))$. Since they share x we rename one of them to $f(f(x')) = g(x')$ and check the pair $(f(f(x)) = g(x),\ f(f(x')) = g(x')\ )$. Now, l_i is $f(f(x))$ and we need to check all non-variable positions of $f(f(x))$ for an overlap with $f(f(x'))$. What are the non-variable positions of $f(f(x))$? It is ϵ and 1. (The position 1.1 is the position of x so we don't worry about that.)

OK, so we first check position ϵ: Are $f(f(x))\ |_\epsilon$ and $f(f(x'))$ unifiable? Of course, with mgu $\{x' \mapsto x\}$ (or its renaming $\{x \mapsto x'\}$). Then $f(f(x))$ reduces to $g(x)$ using the equation $f(f(x)) = g(x)$ at the top, and $f(f(x))$ reduces to the same term $g(x)$ using the equation $f(f(x')) = g(x')$ at position p (which happened to be ϵ). We then have to check if $g(x)$ and $g(x)$ have a common normal form, which is obviously the case.

OK, position ϵ was trivial. Now we check position 1 in $f(f(x))$. Are $f(f(x))\,|_1\ (= f(x))$ and $f(f(x'))$ unifiable? Oh yes, with mgu $\rho = \{x \mapsto f(x')\}$. So now we have $l_i\rho = f(f(f(x')))$ which can be reduced to $g(f(x'))$ by using the first equation at the top, and to $f(g(x'))$ by using the second equation at position 1. Now we need to check whether the critical pair $(g(f(x')),\ f(g(x')))$ is joinable. Do the terms have a common normal form? No, because neither $g(f(x'))$ nor $f(g(x'))$ can be reduced further. Therefore, the specification is not confluent! ♠

Example 39 Is the good old specification $\{0 + x = x,\ s(x) + y = s(x + y)\}$ confluent?

Let's check all pairs of equations:

- The pair $0 + x = x$ and $0 + x' = x'$: There is an overlap at position ϵ in $0 + x$. The mgu is trivial and the two reductions are very uninteresting: $0 + x \rightsquigarrow x$ in both cases.

- The pair $0 + x = x$ and $s(x') + y = s(x' + y)$: The top symbol of $s(x') + y$ is $+$, so the only possible overlap with $0 + x$ is at the top of $0 + x$. However, $0 + x$ and $s(x) + y$ are not unifiable, so there is nothing to check here.

- The pair $s(x) + y = s(x + y)$ and $s(x') + y' = s(x' + y')$: Fairly boring: Same equation, only overlap at the top, the term reduces to the same term in both cases.

There were no interesting overlaps here, and the specification is confluent. ♠

Exercise 73 *(From [55])*

1. *Is* $\{(x + y) + z = x + (y + z)\}$ *confluent?*
2. *Show that* $\{(x + y) + z = x + (y + z),\ x + 0 = x\}$ *is not confluent.*

Exercise 74 *(From [2])*

1. *Find terms* r_1 *and* r_2 *such that* $\{f(g(x)) = r_1,\ g(h(x)) = r_2\}$ *is confluent (and terminating).*
2. *Is* $\{f(g(f(x))) = g(x)\}$ *confluent?*
3. *Consider the specification*

$$\begin{array}{ll} \{\ f(f(x)) = f(x), & g(g(x)) = f(x), \\ f(g(x)) = g(x), & g(f(x)) = g(x)\ \} \end{array}$$

   - *Prove that the specification is confluent.*
   - *(Tricky?) Can you determine the normal form of a term as a function of the number of* f*'s and* g*'s in the term?*
     Hint: *Are there an odd number of* g*'s?*

3.5 Order-Sorted Specifications and Conditional Equations

To simplify the exposition we have treated *unsorted* specifications and have only used *unconditional* equations. In this section we mention *very* briefly and informally some differences between this simpler setting and, respectively, the order-sorted case and the setting with conditional equations.

3.5.1 Order-Sorted Equational Specifications

Since Maude supports not only order-sorted specification but also the specification of *membership equational logic* theories, I will not even define the order-sorted reduction relation but only note that:

- Only terms of sort s can be substituted for variables of sort s in an equation.
- For various reasons, some of them exemplified below, it is an advantage if the equations are *sort-decreasing*. That is, the least sort of each instance of the lefthand side of an equation should be greater than or equal (in the subsort relation $\leq$) to the least sort of the corresponding instance of the righthand side of the equation.

An introduction to order-sorted specifications in Norwegian can be found in [76]. The standard references to order-sorted reduction are e.g. [41, 43]. Here, we will just mention some further issues in connection with the themes we have explored for unsorted specifications, including termination, confluence, and "equivalence."

□

Termination

One way of proving termination of an order-sorted specification is just to ignore the sorts and use the techniques for unsorted systems described earlier. Although this is perfectly OK, it is not a strong method as there are some terminating systems whose "unsorted version" would not terminate.

Example 40 Consider

```
fmod TERM is
  sorts s s' .
  subsort s' < s .
  op f : s -> s .
  op g : s -> s .
  op a : -> s' .
  var x : s' .
  eq f(x) = f(g(x)) .
endfm
```

The "unsorted version" of this specification is nonterminating since it allows an infinite sequence

$$\mathtt{f(a)} \rightsquigarrow \mathtt{f(g(a))} \rightsquigarrow \mathtt{f(g(g(a)))} \rightsquigarrow \cdots$$

However, the order-sorted specification is terminating since the equation $\mathtt{f(x)} = \mathtt{f(g(x))}$ cannot be applied to $\mathtt{f(g(a))}$ because $\mathtt{g(a)}$ is not a term of sort $\mathtt{s'}$. One of the ideas of showing termination is to label each function symbols with the least sorts of its arguments, so that the equation would be $\mathtt{f}_{\mathtt{s}'}(\mathtt{x}) = \mathtt{f}_{\mathtt{s}}(\mathtt{g}_{\mathtt{s}}(\mathtt{x}))$ and then termination can be proved using lpo or mpo with precedence $\mathtt{f}_{\mathtt{s}'} \succ \mathtt{f}_{\mathtt{s}} \succ \mathtt{g}_{\mathtt{s}}$. See e.g. [76, 78, 40] for a treatment of termination of order-sorted systems. ♠

Confluence

We proved confluence by checking critical pairs (overlaps etc.). Any system without critical pairs due to overlaps is confluent in unsorted specifications. This is no longer the case in sort-increasing order-sorted specifications:

Example 41 In the specification

```
fmod CONFL-TROUBLE is
  sorts s s' .
  subsort s' < s .
  op f : s -> s .
  op a : -> s' .
  op b : -> s .
  var x : s' .
  eq a = b .
  eq f(x) = x .
endfm
```

There are no nontrivial overlaps here, but the specification is not confluent since `f(a)` reduces to both `a` and then to `b` and to `f(b)`. Neither `b` nor `f(b)` can be further reduced so the specification is not confluent. Again, sort-decreasingness ensures that the lack of non-joinable critical pairs implies local confluence. ♠

Unification

A unifiable pair of terms may in general have more than one most general unifier in the order-sorted setting. Indeed, there may be an infinite number of mgu's in an sort-increasing specification, but only a finite number in a sort-decreasing specification [95]. □

3.5.2 Conditional Equations

Maude applies a conditional equation

$$l = r \textbf{ if } t_1 = u_1 \wedge \ldots \wedge t_n = u_n$$

using a substitution σ by checking whether $(t_1\sigma)!$ equals $(u_1\sigma)!$, ..., and $(t_n\sigma)!$ equals $!u_n\sigma)!$.

□ That is, the specification must still be terminating and confluent, when taking the conditional equations into account. For example, the specification

$$\{a = e,\ a = e',\ b = b',\ c = d \textbf{ if } b = b',\ e = e' \textbf{ if } c = d\}$$

is confluent only if we do not ignore the conditional equations. Such specifications are OK in Maude, since the system can find out whether an equation can be applied.

This is not the case with specifications like

$$\{a = b \textbf{ if } a = b\}.$$

If we try to execute the Maude command `red` a `.`, the system will check whether the equation can be applied by checking whether $a = b$, which is done by checking whether the equation can be applied, and so on, leading to infinite looping. □

3.6 Dealing with Nonterminating and Nonconfluent Specifications

Not all specifications are naturally terminating and/or confluent. This section shows some techniques used in Maude to treat some nonterminating specifications. (Section 4.1.1 briefly describes the idea of *completing* a nonconfluent and possibly nonterminating specification by adding new equations when non-joinable critical pairs are found.)

3.6.1 Equational Attributes in Maude

Sometimes equations which lead to nontermination are needed. Consider the data type of sets (or multisets). Obviously, the sets $\{a, b\}$ and $\{b, a\}$ are the same set, and therefore their representations in Maude should be considered equivalent. To achieve that, we need to somehow state that $\{x, y\} = \{y, x\}$ for all elements x and y of the given sort.

In general, there is sometimes a need to impose *commutativity* on some function f:

$$f(x, y) = f(y, x).$$

We have seen that such an equation leads to nontermination since there would be infinite derivations like

$$f(x,y) \rightsquigarrow f(y,x) \rightsquigarrow f(x,y) \rightsquigarrow \cdots$$

However, commutativity is too important to give up, so what should we do? The Maude solution to this problem of having both commutativity and termination is to

- omit to explicitly declare the equation $f(x,y) = f(y,x)$, and
- instead declare that "f is commutative" and that Maude always "keeps in mind" that f is commutative when computing.

When a function is declared to be commutative in this special way in Maude, computations are no longer performed on *terms* but on *C-equivalence classes* of terms where C is the commutativity axiom given by

$$f(x,y) = f(y,x).$$

In the same way, we may compute modulo *associativity* and *identity*, and any combination of commutativity, associativity, and identity in Maude. Indeed, we will do that quite frequently in this course to define lists, sets, and multisets.

Commutativity

In Maude we can declare that a function `f` is commutative by giving it an attribute `comm` as illustrated in the following example:

```
fmod COMM1 is
  sort s .
  op f : s s -> s [comm] .
  ops a b c : -> s .
  eq f(a,b) = b .
endfm
```

□ The function `f` has been declared to be commutative, and one therefore works on the set $\mathcal{T}_{\Sigma,C} = \mathcal{T}_\Sigma/C$ of equivalence classes of terms

$$\mathcal{T}_{\Sigma,C} = \{[t]_C \mid t \in \mathcal{T}_\Sigma\}$$

modulo commutativity with

$$[t]_C = \{u \mid t \overset{*}{\leftrightsquigarrow}_C u\}$$

where C is the specification $\{\mathtt{f}(x,y) = \mathtt{f}(y,x)\}$. For example, $[\mathtt{f(a,b)}]_C = \{\mathtt{f(a,b)},\ \mathtt{f(b,a)}\}$, and

$$[\mathtt{f(a,f(b,c))}]_C = \begin{array}{l}\{\mathtt{f(a,f(b,c))},\\ \ \mathtt{f(a,f(c,b))},\\ \ \mathtt{f(f(b,c),a)},\\ \ \mathtt{f(f(c,b),a)}\}\end{array}$$

□

Important: We will write t for $[t]_C$.

The term `f(b,a)` can be reduced to `b` in `COMM1` since by `f(b,a)` we mean $[\texttt{f(b,a)}]_C$ and

$$[\texttt{f(b,a)}]_C = [\texttt{f(a,b)}]_C \rightsquigarrow [\texttt{b}]_C.$$

Exercise 75 *Test out this in Maude by first entering the module* `COMM1` *and then executing the Maude command* `red f(b,a) .`

Exercise 76 *For each of the (equivalence classes of the) terms*

- `f(f(b,a),a)`
- `f(b,b)`
- `f(f(a,b),f(b,a))`

first compute its normal form in `COMM1` *"by hand" and by Maude using its* `red` *command.*

Example 42 Only function symbols with the `comm` attribute are treated in this special way. For example, in

```
fmod COMM2 is
  including COMM1 .
  op g : s s -> s .
  eq g(a,b) = b .
endfm
```

the term `g(b,a)` does not reduce to `b`. ♠

We may sometimes use the `comm` attribute even when it is not really necessary, just to get "nicer" specifications.[17]

Example 43 A function `minimum` which returns the smallest of two integers can be defined as follows:

```
fmod MIN1 is
  protecting INT .
  op minimum : Int Int -> Int [comm] .
  vars I J : Int .
  ceq minimum(I, J) = I if I <= J .
endfm
```

♠

[17] "Beauty is our business," according to José Meseguer, the inventor of rewriting logic and Maude.

Exercise 77 *Explain why the Maude command* `red minimum(8,5)` *will return* `5` *when executing the specification* `MIN1`.

Exercise 78 *In Exercise 17 you defined the module* `BOOLEAN` *of Boolean values. Define a module* `BOOLEAN-COMM` *where* `and` *and* `or` *are declared to be commutative and define these functions.*

Associativity

A function f is *associative* if

$$f(f(x,y),z) = f(x,f(y,z))$$

holds for all x, y, z. The addition function on the integers is associative since $(x + y) + z = x + (y + z)$. In Maude, we can say that a function is associative by declaring it with the `assoc` attribute:

```
op f : s s -> s [assoc] .
```

The term $\texttt{f}(\texttt{f}(a,b),\texttt{f}(c,d))$ is considered the same as $\texttt{f}(a,\texttt{f}(b,\texttt{f}(c,d)))$ and $\texttt{f}(\texttt{f}(\texttt{f}(a,b),c),d)$ and $\texttt{f}(\texttt{f}(a,\texttt{f}(b,c)),d)$ and $\texttt{f}(a,\texttt{f}(\texttt{f}(b,c),d))$ when `f` is declared associative. Since the parentheses can be rearranged arbitrarily for associative operators, they are no longer needed for `f` and we can write

$$\texttt{f}(a,b,c,d)$$

instead of either of the above terms. Likewise, if the infix symbol `+` is declared associative, we may write `1 + 2 + 3 + 4`.

Do we really need to give associativity this special treatment? The associativity axiom $f(f(x,y),z) = f(x,f(y,z))$ does *not* cause nontermination (prove it!). There are at least two reasons to treat associativity in this way:

- Specifications of common and important data types such as lists and sets/multisets are much more elegant as we may omit parentheses and define functions on such types much more naturally.
- Although associativity by itself does not lead to nontermination, it leads to nontermination if the function already is declared commutative:

  ```
  op f : s s -> s [comm] .
  vars X Y Z : s .
  eq f(f(X,Y),Z) = f(X,f(Y,Z)) .    *** Associativity
  ```

  The specification is nonterminating modulo commutativity since there is an infinite derivation

  $[\texttt{f(f(a,b),c)}]_C \rightsquigarrow [\texttt{f(a,f(b, c))}]_C = [\texttt{f(f(b,c),a)}]_C$
  $\rightsquigarrow [\texttt{f(b,f(c,a))}]_C = [\texttt{f(f(c,a),b)}]_C \rightsquigarrow [\texttt{f(c,f(a,b))}]_C = [\texttt{f(f(a,b),c)}]_C \rightsquigarrow \cdots$

  Therefore, if `f` is declared commutative, associativity of `f` must be taken care of by adding `assoc` as an attribute:

  ```
  op f : s s -> s [assoc comm] .
  ```

Associativity: Lists of Integers

(From [13]) In Section 2.3.4 we defined lists of natural numbers using a constructor

```
op __ : List Nat -> List [ctor] .
```

and a constant constructor `nil`. All lists are of the form $(\ldots(((\texttt{nil}\ n_1)\ n_2)\ n_3)\ldots)\ n_k$ (even though the parentheses may be omitted since this is the only way to parse the term). However, it is more natural to view lists as "flat" structures. Therefore, we propose a new version of lists using associativity of `__`.

The new version works with integers instead of our natural numbers, has a constructor `nil` of sort `List` as before, a subsort declaration `Int < List` which states that each integer is also a list, and a *concatenation* constructor

```
op __ : List List -> List [ctor assoc] .
```

The terms `nil` n, $n_1\ n_2$ `nil nil` n_3, and $n_1\ n_2\ n_3$ are all lists, and since `__` is associative, the lists are "flat." To avoid the `nil`s, we may add equations `L nil` = `L` and `nil L` = `L` for `L` a variable of sort `List`.

Lists have some partial functions such as `first`, `last`, and `rest`, which are not naturally defined on empty lists (which integer is the first one in an empty list?). Following the guidelines in Section 2.4.4 we declare a subsort `NeList` of `List` and add a declaration

```
op __ : NeList NeList -> NeList [ctor assoc] .
```

(Remember that the attributes (with the possible exception of `ctor`) must be the same in all subsort overloaded declarations of a function symbol.) The specification of lists of integers could therefore be:

```
fmod ASSOC-LIST-INT is
  protecting INT .

  sorts List NeList .
  subsorts Int < NeList < List .

  op nil : -> List [ctor] .
  op __ : List List -> List [assoc ctor] .
  op __ : NeList NeList -> NeList [assoc ctor] .

  op length : List -> Nat .
  ops first last : NeList -> Int .
  op empty? : List -> Bool .
  op rest : NeList -> List .
  op reverse : List -> List .
  op _occursIn_ : Int List -> Bool .
```

```
  *** More specific for lists on totally ordered domains:
  op max : NeList -> Int .
  op isSorted : List -> Bool .

  vars I J : Int .
  var L : List .
  var NEL : NeList .

  eq L nil = L .
  eq nil L = L .
  eq length(nil) = 0 .
  eq length(I) = 1 .
  eq length(I NEL) = 1 + length(NEL) .
  eq first(I) = I .
  eq first(I NEL) = I .
  eq empty?(L) = L == nil .
  eq reverse(nil) = nil .
  eq reverse(I) = I .
  eq reverse(I NEL) = reverse(NEL) I .
  eq I occursIn nil = false .
  eq I occursIn J = I == J .
  eq I occursIn J NEL = (I == J) or (I occursIn NEL) .
  eq max(I) = I .
  eq max(I NEL) = if I > max(NEL) then I else max(NEL) fi .
  ...
endfm
```

The definitions are based on the fact that a list is either `nil`, a one-element list of the form k for an integer k, or a list with two or more elements of the form k *nel* (and, because of associativity, of the form *nel*$'$ k') for *nel* a non-empty list. We need not worry about `nil`'s in the lists since they are removed by the first two equations. Hopefully you can see from these definitions how much easier life is with `assoc` than in Section 2.3.4 for instance in the definition of `first` and `reverse`.

Exercise 79 *Finish the module* `ASSOC-LIST-INT` *by defining the non-constructor functions* `last` *and* `rest`*. Test the functions by executing them in Maude.*

Identity Attributes

An attribute `idl:` t, for a ground term t, of a binary function f declares that t is the *left identity* of f. That is, computations are performed modulo the equation

$$f(t, x) = x.$$

Similarly, if `idr:` t is an attribute of f, it means that t is the *right identity* of f; the mathematical meaning is $f(x, t) = x$ for all x. Finally, `id:` t means that t is both the left and the right identity of f:

$$f(t, x) = x \text{ and } f(x, t) = x.$$

Associativity and Identity: Lists

The module `ASSOC-LIST-INT` contained the identities `L nil` = `L` and `nil L` = `L` as equations. These equations suggest that `nil` is the identity element of the concatenation operator `__` and should be declared as such. The gain is that every list has the form `nil` or k l for k an integer and l a list, since $k = k$ `nil`. In the definitions below, we therefore need to consider fewer cases (we don't need to treat the case where the list consists of exactly one integer) as illustrated by the shorter (and nicer?) definitions. Using both associativity and identity seems to be the preferred way of specifying lists in Maude:

```
fmod LIST-INT is
  protecting INT .

  sorts List NeList .
  subsorts Int < NeList < List .

  op nil : -> List [ctor] .
  op __ : List List -> List [assoc id: nil ctor] .
  op __ : NeList NeList -> NeList [assoc id: nil ctor] .

  op length : List -> Nat .
  ops first last : NeList -> Int .
  op empty? : List -> Bool .
  op rest : NeList -> List .
  op reverse : List -> List .
  op _occursIn_ : Int List -> Bool .
  op max : NeList -> Int .
  op isSorted : List -> Bool .

  vars I J : Int .
  var L : List .  var NEL : NeList .

  eq length(nil) = 0 .
  eq length(I L) = 1 + length(L) .
  eq last(L I) = I .
  eq empty?(L) = L == nil .
  eq rest(I L) = L .
  eq I occursIn nil = false .
  eq I occursIn J L = (I == J) or (I occursIn L) .
  ...
endfm
```

Exercise 80 *Complete the module* `LIST-INT` *by defining the non-constructor functions* `first`, `reverse`, `max`, *and* `isSorted` *which are not already defined above. Then test each of the functions by executing them in Maude on the following lists:*

- `nil` *(where possible)*

- `200 2`
- `1 2 3 4 5`
- `5 6 7 7 8`
- `12`
- `5 7 6`

Exercise 81 *Define (in the module* `LIST-INT` *or in an extension of that module) a function*

```
op comesBeforeIn : Int Int List -> Bool .
```

such that `comesBeforeIn`(i, j, l) *is* `true` *if and only if there are elements* i *and* j *in the list* l*, and where the first occurrence of* i *comes before the first occurrence of* j *in* l*. Test your function in Maude. For example,*

- `comesBeforeIn(2, 3, 1 2 4 5 6 3 4)` *should return* `true`*,*
- `comesBeforeIn(2, 2, 1 2 8 2)` *should return* `false`*,*
- `comesBeforeIn(2, 3, 3 5 2 7 6 3 1)` *should return* `false`*,*
- `comesBeforeIn(2, 3, 1 2)` *should return* `false`*,*
- `comesBeforeIn(2, 3, 2 3)` *should return* `true`*, etc.*

Exercise 82 *Define a function*

```
op _>lex_ : List List -> Bool .
```

such that l_1 `>lex` l_2 *is* `true` *if* l_1 *is lexicographically greater than* l_2*. For example,*

- `1 2 3 4 >lex 1 2 3` *should return* `true`*,*
- `2 3 4 >lex 1 2 3 4` *should return* `true`*,*
- `1 >lex 1 0` *should return* `false`*,*
- `1 2 3 >lex 1 2 3` *should return* `false`*, and*
- `2 5 6 >lex 2 4 678 12` *should return* `true`*.*

Associativity, Commutativity, and Identity: Multisets and Sets

A multiset can be seen as a list where the order of the elements does not matter. Multisets may be modeled by "lists," where the multiset union operator `__` is also commutative. The data type of finite multisets of the machine integers could be specified in Maude as follows:

```
fmod MSET-INT is
  protecting INT .

  sorts Mset NeMset .                    *** Multisets and non-empty multisets
  subsorts Int < NeMset < Mset .         *** A number is also a multiset

  op none : -> Mset [ctor] .             *** Empty multiset
  op __ : Mset Mset -> Mset [ctor assoc comm id: none] .
  op __ : NeMset NeMset -> NeMset [ctor assoc comm id: none] .

  op size : Mset -> Nat .                *** No of elements in a multiset
  op mult : Int Mset -> Nat .            *** Multiplicity of an element
  op delete : Int Mset -> Mset .         *** Remove ONE occurrence of element
  op _in_ : Int Mset -> Bool .           *** Is element in multiset?
  op max : NeMset -> Int .               *** Largest element
  op empty? : Mset -> Bool .             *** Empty?
  op _>mul_ : Mset Mset -> Bool .        *** Multiset comparison

  vars I J : Int .
  var MS : Mset .  vars NEMS NEMS' : NeMset .

  eq size(none) = 0 .
  eq size(I MS) = 1 + size(MS) .
  eq delete(I, I MS) = MS .
  ceq delete(I, MS) = MS if not I in MS .
  eq I in MS = mult(I, MS) > 0 .
  ...
endfm
```

Exercise 83 *Functions such as* `size` *could also be defined by*

```
  eq size(none) = 0 .
  eq size(I) =  1 .
  eq size(NEMS NEMS') = size(NEMS) + size(NEMS') .
```

Explain why one should not use variables `MS` *and* `MS'` *of sort* `Mset` *instead of the variables* `NEMS` *and* `NEMS'` *of sort* `NeMset` *in the definitions above.*

Hint: `size(4)` *and* `size(4 none)` *are equivalent modulo attributes of* `__`*.*

Exercise 84 *Explain why* `delete(2002, 1 2 2002 3)` *returns the multiset* `1 2 3` *when the function is defined as above.*

Exercise 85 *Define the functions* `mult`, `max`, `empty?`, *and the multiset comparison operator* `>mul` *which are declared but not defined in the module* `MSET-INT`. *Test your specification using Maude.*

Exercise 86 *Assume that we have already defined two sorts* `Obj` *and* `Msg`. *Define a sort* `Mset-ObjMsg` *whose elements are multisets of* `Obj`*- and* `Msg`*-elements (that is, a multiset may contain both* `Obj` *and* `Msg` *elements).*

Only define the sort `Mset-ObjMsg` *with subsorts and constructors and don't worry about declaring or defining functions such as* `size`, `delete`, *...*

Sets

A *set* is essentially a multiset where the multiplicity of elements do not matter. The data type of sets of machine integers can be defined as multisets of machine integers where one adds the extra axiom

```
eq I I = I .
```

(for `I` a variable of sort `Int`) which removes duplicates from the set.

Exercise 87 *Given the above sketch of the definition of sets, show that the set* `3 5 8 5 6 5` *reduces to* `3 5 8 6`.

Exercise 88 *Define a data type of sets of integers which contains functions* `_in_` *(does the given number belong to the set?),* `delete` *(which removes an element from a set),* `card` *(for the* cardinality *(the number of (distinct) elements) of a set),* `_setMinus_` *(set difference), and* `_intersect_` *(which returns the intersection of two sets).*

Take care that your specification is confluent. `delete(1, 0 1 2 1)` *should always return* `0 2` *no matter how the equations are applied. Similarly, the cardinality of the set* `0 1 2 1` *is always* `3`.

Exercise 89 *(Slightly tricky?)*

1. *Define a data type* `ListQid` *of lists of* `Qid`*-elements. In this exercise you will use both lists and sets, and you are therefore advised to use a symbol other than* `__` *for list concatenation. Use e.g.,* `_:_` *or* `_._` *or whatever your favorite list concatenation operator is. Only declare and define functions you will need in this exercise.*

2. *Define a data type* `Set-ListQid` *of sets of lists of quoted identifiers. Again, define functions needed to solve the next part of this exercise.*

3. *Define a function*

   ```
   op perm : ListQid -> Set-ListQid .
   ```

which takes a list of `Qid`*'s and returns the set of all* permutations *of this list. (A permutation of a list is a list where the elements are the same but are "rearranged.") For instance, the set of all permutations of the list* `'a : 'b : 'c` *(I use* `_:_` *as the list concatenation operator) is the set*

```
('a : 'b : 'c) ('a : 'c : 'b) ('b : 'a : 'c)
('b : 'c : 'a) ('c : 'a : 'b) ('c : 'b : 'a)
```

Hint: *This exercise seems to be harder than most exercises we have seen so far in this course. One idea to generate all permutations of a list such as* `'a : 'b : 'c` *is to generate "*`'a` *plus all permutations of* `'b : 'c`*" and "*`'b` *plus all permutations of* `'a : 'c`*" and "*`'c` *plus all permutations of* `'a : 'b`*." That is, we gradually construct each permutation.*

In my solution, the job of actually generating all permutations is done by a function `p` *where*

$$\mathtt{p}(L_1, L_2, L_3)$$

generates all permutations of $L_1 : L_2 : L_3$ *which start with* L_1*, and where the next qid (after* L_1*) is taken from the list* L_2*. The* L_3*-elements have already been taken.*

Hopefully, these hints can get you going to solve this slightly tricky problem. Don't be surprised if your specification is very short and quite elegant!

Lists and multisets are important data types so their specification should be *parametric* in the sort of the elements, so that we don't have to write new specifications for multisets of Booleans, integers, quoted identifiers, etc. Although Maude supports such parameterization, we will, as mentioned earlier, not treat parameterization in this course to avoid introducing more Maude technicalities. Instead we will "repeat" specifications of lists, multisets, etc.

3.6.2 * C- and AC-matching is NP-hard

□ To check whether an equation can be applied to a term when function symbols have attributes such as `assoc` and `comm`, it has to be checked whether the lefthand side l matches a subterm $t|_p$ *modulo* the (equations corresponding to the) attributes of the function symbols. The equation $f(a, x) = r$ can for instance be applied to $f(b, a)$ when f is declared to have the attribute `comm`.

Matching modulo commutativity, associativity, or associativity and commutativity (denoted, respectively, C-, A-, and AC-matching) may produce more than one match. For example, $f(x, y)$ matches $f(a, b)$ via both $\{x \mapsto a,\ y \mapsto b\}$ and $\{x \mapsto b,\ y \mapsto a\}$ when f is commutative. (Similarly, $g(x, y)$ matches $g(g(a, b), c)$ via both $\{x \mapsto g(a, b),\ y \mapsto c\}$ and $\{x \mapsto a,\ y \mapsto g(b, c)\}$ when g is declared with the attribute `assoc`.)

In general, E-matching is undecidable for arbitrary theories E (since $t \overset{*}{\leftsquigarrow}_E u$ if and only if t E-matches u for ground terms t and u, and we will see in Section 4.1 that it is undecidable whether $t \overset{*}{\leftsquigarrow}_E u$). However, finding all matches modulo A, C, AC, ..., is decidable, although it may not be very efficient. In fact, we show that C- and AC-matching is NP-hard by reduction from positive 1-in-3-SAT, which is known to be an NP-complete problem [37].

Theorem 6 *C-matching is NP-hard.*

Theorem 7 *AC-matching is NP-hard.*

Proof. We prove the theorem by reduction from positive 1-in-3-SAT which is known to be an NP-complete problem. A 1-in-3-SAT instance consists of a set of propositional variables and a set

$$\{(p_i \vee q_i \vee r_i) \mid 1 \leq i \leq n\}$$

of clauses where all the p_i, q_i, and r_i are all propositional variables. The problem of 1-in-3-SAT is to decide whether there exists a valuation of the propositional variables (to `true` or `false`) such that for each clause $(p_i \vee q_i \vee r_i)$ *exactly one* of the propositional variables p_i, q_i, or r_i is `true`.

Intuitively, each propositional variable p corresponds to a variable x_p in the corresponding matching problem. Furthermore, we have two constants `true` and `false`. The operator $\vee$ is modeled by an AC-operator $\vee$. An instance of the 1-in-3-SAT problem is a "yes" instance if and only if the set

$$\{x_{p_i} \vee x_{q_i} \vee x_{r_i} \text{ "matches" } \texttt{true} \vee \texttt{false} \vee \texttt{false} \mid 1 \leq i \leq n\}$$

of matching problems has a solution. (A *set* $\{t_i$ "matches" $u_i \mid 1 \leq i \leq n\}$ of matching problems can be seen as *one* matching problem

$$f(t_1, \ldots, t_n) \text{ "matches" } f(u_1, \ldots, u_n)$$

for a new symbol f.) ♣

Although the theorems indicate that computing with functions that have the attributes `assoc` and/or `comm` may be very time-consuming, the Maude developers have put a lot of effort and ingenuity into the making the A, C, and AC-matching algorithms as fast as possible. In most cases AC-reduction is not really a bottleneck, although the theorems indicate that (unless P=NP) there are cases where computing with AC-symbols must indeed take a lot of time. □

3.6.3 * Telling Maude how to Evaluate an Expression

□ Consider our definition of the factorial function:

```
eq N ! = if N > 0 then N * sd(N, 1) ! else 1 fi .
```

Although `if_then_else_fi` is a built-in function, we could assume that it is explicitly defined by the following equations:

```
eq if true then X else Y fi = X .
eq if false then X else Y fi = Y .
```

The specification of `!` is nonterminating since we have the following derivation:

```
0 !  ⇝  if 0 > 0 then 0 * sd(0,1) ! else 1 fi
     ⇝  if 0 > 0 then 0 * 1 ! else 1 fi
     ⇝  if 0 > 0 then 0 * (if 1 > 0 then 1 * sd(1,1) ! else 1 fi) else 1 fi
     ⇝  if 0 > 0 then 0 * (if 1 > 0 then 1 * 0 ! else 1 fi) else 1 fi
     ⇝  ...
```

Since the derivation started with `0 !` and has reached a term in which `0 !` is a subterm we have proved that the specification is nonterminating.

There are no tricks. The specification *is* nonterminating. The point is that we see `if_then_else_fi` and *assume* that it first computes the value of its first argument, then evaluates "itself" by using the `if_then_else_fi`-equations above. However, a term `if` b `then` t `else` u `fi`

could equally well be evaluated by evaluating the *second* argument of `if_then_else_fi` first, as happened above.

Such cases of nontermination can be avoided in Maude by telling Maude how to evaluate a term. This is done by giving an *operator evaluation strategy* [29] to a function using the attribute `strat`. For example, if we declare

```
op f : s1 s2 s3 -> s [strat (2 0 1 3 0)] .
```

then an expression $\texttt{f}(t_1, t_2, t_3)$ will be evaluated in the following order:

1. t_2 is evaluated as much as possible to t_2';
2. the term $\texttt{f}(t_1, t_2', t_3)$ is evaluated "at the top" by using `f`-equations;
3. reduce t_1 if still in the form $\texttt{f}(t_1, t_2'', t_3')$;
4. reduce t_3 if possible;
5. reduce $\texttt{f}(\ldots)$ at the top if possible.

That is, Maude computes an expression of the form $f(t_1, \ldots, t_n)$ by following the `strat`-list of the function symbol f "from left to right." A number i in the `strat`-list means "evaluate the ith subterm (namely t_i)" while a number `0` means "evaluate at the top." Each `strat`-list must end with a `0` due to some subtleties [29].

Example 44 The operator `if_then_else_fi` should have the attribute `strat (1 0 2 3 0)`, which exactly states that the test is computed first, then the whole `if_then_else_fi`-expression, and only thereafter the last two arguments. ♠

The default evaluation strategy of a function with n arguments in Maude is, by the way, `(1 2` $\ldots$ n `0)`, which is usually denoted *eager* evaluation.

Example 45 The efficient evaluation strategy for a function f defined by the equation $f(x, y, z) = y$ is `(2 0)` or `(0)`. ♠

Exercise 90 *What's the difference between the two suggested evaluation strategies for f in Example 45? Why is the strategy* `(1 2 3 0)` *unnecessary inefficient? What is the most efficient strategy for the function g defined by $g(x, y, z) = h(y, y)$?*

Exercise 91 *The Boolean tests* `&&` *and* `||` *evaluate their second argument only if necessary in languages like C and Java, so that b_2 is not evaluated in b_1* `&&` *b_2 if b_1 evaluates to "false." The built-in functions* `and` *and* `or` *evaluate both their arguments in Maude:*

```
Maude> red 0 > 0 and (5 / 0 > 4) .
result [Bool]: false and 5 / 0 > 4
```

Define two Boolean functions `and-then` *and* `or-else` *which work more like the C conjunctions and disjunctions.*

□

3.7 Specification *is* Programming: Quick-Sort and Merge-Sort

As mentioned in the introduction, I once had a student who complained about all the details of the various sorting algorithms in Java[18] and asked why you couldn't just ask the computer to sort a list without having to worry about minor details.

[18] Is it `i=0` or `i=1`? `j=i` or `j=i+1`? `i++` or `++i`? Until `j>k` or `j>=k`? A `-1` or `+1` missing somewhere? etc.

Well, this may be too much to ask for, not least because there are many different classes of sorting algorithms. Instead, you can *describe* (or *specify*) the different algorithms using equations, and this specification is at the same time the desired program. In addition, the intuitive Maude formalism may well provide a more understandable and readable description of an algorithm than both a textbook description and a description using conventional code (which is more or less impossible to understand if you don't know fairly well what is happening).

We illustrate this point with specification of the *quick-sort* and *merge-sort* algorithms for sorting a list of integers.

3.7.1 Quick-Sort

If you never really grasped the quick-sort algorithm because you got lost in the details, you may remember the following Maude specification:

```
fmod QUICK-SORT is
  protecting LIST-INT .
  op quicksort : List -> List .

  vars L L' : List .
  vars M N : Int .

  eq quicksort(nil) = nil .
  eq quicksort(L N L') = quicksort(smallerElements(L L', N))
                         equalElements(L N L', N)
                         quicksort(greaterElements(L L', N)) .
```

where `smallerElements`(l, n) removes from the list l all elements which are greater than or equal to n:

```
  ops smallerElements greaterElements equalElements : List Int -> List .

  eq smallerElements(nil, N) = nil .
  eq smallerElements(N L, M) = if N < M then (N smallerElements(L, M))
                               else smallerElements(L, M) fi .
  eq equalElements(nil, N) = nil .
  eq equalElements(N L, M) = if N == M then (N equalElements(L, M))
                             else equalElements(L, M) fi .
  eq greaterElements(nil, N) = nil .
  eq greaterElements(N L, M) = if N > M then (N greaterElements(L, M))
                               else greaterElements(L, M) fi .
endfm
```

The following description of the quick-sort function above is taken from the "standard" textbook on algorithms [45]:

"The quick-sort algorithm sorts a list L using a simple recursive approach. The main idea is to apply the divide-and-conquer technique, whereby we divide L into sublists whose range of elements are disjoint, recurse to sort each sublist, and then combine the sorted sublists by a simple concatenation. In particular, the quick-sort algorithm consists of the following three steps:

1. **Divide:** [...] select a specific element `N` from L [where L is `L N L'` in the equation above], which is called the *pivot*. For instance, let the pivot `N` be the last element. Remove all elements from L and put them into three lists:
   - S, storing the elements in L less than `N`
   - E, storing the elements in L equal to `N`
   - G, storing the elements in L greater than `N`

   Of course, if the elements of L are all distinct, then E holds just one element—the pivot.
2. **Recurse:** Recursively sort the lists S and G.
3. **Conquer:** Put back the elements into L by first inserting the elements of S, then those of E, and finally those of G."

The Maude definition is a (more) general *specification* in that is chooses the pivot `N` "nondeterministically" instead of being forced to choose "for instance" the last element.

Which description is easier to read, the Maude specification or the textbook description?

Exercise 92 *The quick-sort definition in Maude is the "most general" in that we do not specify* which *element should be the pivot. In this exercise we will specify a potentially more efficient quick-sort version, which, for lists of at least two elements, will look at the* first *and* last *element in the list, and choose as pivot element the number* $\frac{first+last}{2}$*. (It is possible that such a number is not an element in the list, but that doesn't matter.)*

1. *Specify and execute this version of quick-sort in Maude. (It could also be interesting in checking on large examples how much more efficient this version is compared to the original one.)*
2. *Compare the execution times of your quick-sort programs in Maude and in Java.*
3. *Argue convincingly (but possibly informally) that your specification is terminating.*

3.7.2 Merge-Sort

Another standard textbook on algorithms [97] describes the *merge-sort* algorithm as follows:

1. If the number of items to sort is 0 or 1, return (such a list is of course already sorted).

2. Otherwise: split the list into two halves with the same number of elements in each half[19], and recursively sort the first and second halves separately.
3. Merge the two sorted halves into a sorted group.

A specification of the merge-sort algorithm can be given as follows in Maude:

```
fmod MERGE-SORT is protecting LIST-INT .
  op mergeSort : List -> List .
  op merge : List List -> List [comm] .   --- notice 'comm' attribute

  vars L L' : List .
  vars NEL NEL' : NeList .
  vars I J : Int .

  eq mergeSort(nil) = nil .
  eq mergeSort(I) = I .
  ceq mergeSort(NEL NEL') = merge(mergeSort(NEL), mergeSort(NEL'))
       if length(NEL) == length(NEL')  or length(NEL) == s length(NEL') .

  eq merge(nil, L) = L .
  ceq merge(I L, J L') = I merge(L, J L') if I <= J .
endfm
```

As indicated by its name, `merge` merges two lists, so that `merge(1 8 9, 3 5 11)` gives the list `1 3 5 8 9 11`.

The *raison d'être* for merge-sort is that its execution time is $O(n \log n)$. However, the above specification is significantly less efficient, since the splitting of a list into two halves is done by matching. The usefulness of this specification is that (i) it provides a precise description of merge-sort that is easy to understand, and (ii) that it defines a *prototype* that can be used to quickly test the merge-sort algorithm before a more detailed and efficient algorithm is implemented. (Of course, we already knew that the merge-sort algorithm is supposed to work. But imagine that you have an algorithm that you hope solves a difficult problem. It is then very useful to *quickly* being able to develop and test a Maude prototype/model of your algorithm *before* implementing it in all its glorious detail.)

Exercise 93 *Modify the* `mergeSort` *function so that it becomes much more efficient. Then test it against your Java implementations of merge-sort, if you have any. Which one is fastest?*

[19] or such that the first part has one more element than the second part

Chapter 4

Logical Semantics of Equational Specifications

This chapter defines the mathematical (or logical) meaning of an equational specification by describing when two terms are "logically equivalent" according to the specification.

Many mathematical theories, such as the theory of groups, rings, etc., are equational specifications. Given two terms t and u, a mathematician may be interested in knowing whether the equivalence $t = u$ "follows logically" from the equations. For example, do $x \circ (x^{-1}) = e$ and $(x^{-1})^{-1} = x$ hold in all *groups*? That is, do they follow logically from the *group axioms* $\{(x \circ y) \circ z = x \circ (y \circ z),\ e \circ x = x,\ (x^{-1}) \circ x = e\}$? A somewhat more ambitious goal is to model all known and relevant laws of physics and mathematics as equations and prove that $E(x) = m(x) \times c^2$ *really* follows logically from these equations for all objects x.

To help the mathematicians we need to know what it means that $t = u$ "follows logically" from a set of equations. We will first interpret "t and u are equivalent according to a set of equations" (or "$t = u$ follows logically from a set of equations") in the most general way, namely that $t = u$ can be deduced from the equations using the rules of *equational logic*.

Although it is undecidable whether t and u are "equal," we can decide whether t and u are equal when the specification is terminating and confluent by checking whether $t!$ and $u!$ are the same term. (For example, the normal form of `s(0) + s(s(0))` and `s(s(0)) + s(0)` is `s(s(s(0)))` in both cases; the terms are therefore equivalent in `NAT-ADD`.)

It is nice to show that `s(0) + s(s(0))` = `s(s(0)) + s(0)` is a *logical consequence* of our specification `NAT-ADD`. However, we have more ambitious goals. We will show how to prove that expected/desired properties such as m `+` $n = n$ `+` m, `reverse(reverse(`bt`))` $= bt$, and `length(concat(`l_1, l_2`))` = `length(`l_1`) + length(`l_2`)` are logical consequences of our specifications for *all* natural numbers m n, for *all* binary trees bt, and for *all* lists l_1 and l_2. Once we have proved such desired consequences we can be more confident that our specifications of `+`, `reverse`, `length`, and `concat` are indeed correct.

However, before we become too ambitious about proving properties, we need to make precise *exactly* what we mean by m `+` $n = n$ `+` m. Could m and n be *anything* (including possible additions such as infinity values), or should they only range over *all the ground constructor terms* of sort `Nat`? In Section 4.1 we describe what it means that " m `+` $n = n$ `+` m is a logical

consequence of the equations for any m and n," and in Section 4.2 we show what it means that "$m + n = n + m$ holds for all ground constructor terms m, n of sort `Nat`," and how to use induction to prove properties of this latter kind.

4.1 Equational Logic

Given a set E of equations.[1] We write $E \vdash t = u$ for the *sequent* which means that the equivalence $t = u$ follows from the equations E according to the rules of *equational logic*. That is, when $t = u$ "follows logically" from E.

Definition 26 (Equational logic) *Given a set E of equations, we write $E \vdash t = u$, for terms t and u, for "t equals u follows logically from E," if and only if $E \vdash t = u$ can be derived according to the following rules of equational logic:*

E_1 **(Substitutivity):** *The sequent $E \vdash l\sigma = r\sigma$ holds for any equation $l = r$ in E and any substitution σ.*

E_2 **(Reflexivity):** *$E \vdash t = t$ holds for any term t.*

E_3 **(Symmetry):** *If $E \vdash t = u$ holds then $E \vdash u = t$ holds.*

E_4 **(Transitivity):** *If $E \vdash t_1 = t_2$ and $E \vdash t_2 = t_3$ both hold, then $E \vdash t_1 = t_3$ holds.*

E_5 **(Congruence):** *If $E \vdash t_1 = u_1$, ..., and $E \vdash t_n = u_n$ all hold, then*

$$E \vdash f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n)$$

holds for each function symbol f.

Reasoning with these kinds of logics, or *deduction systems*, may take some time getting used to if you have never seen them before. Essentially, $t = t'$ follows logically from a E of equations, written $E \vdash t = t'$, *only if* this can be deduced using the above axiom schemes (**Substitutivity** and **Reflexivity**) and deduction rules (**Symmetry**, **Transitivity**, and **Congruence**). The basic facts that we can start each deduction with are, therefore, that we can deduce $E \vdash t\sigma = t'\sigma$ for each equation $t = t'$ in E and each substitution σ, and that we can deduce $E \vdash t = t$ for each term t. From these basic facts, we can then use the deduction rules of equational logic to deduce new facts, as exemplified below.

Example 46 Given $E = \{f(x) = g(x),\ a = b\}$. Does $b = a$ follow logically from E? Yes, we can deduce $E \vdash b = a$ as follows:

1. By **Substitutivity** we know that $E \vdash a = b$, since $a = b$ is an equation in E. The substitution is of course just the empty substitution.

[1] To avoid detail, we don't worry too much about the signature Σ which is given from E when we are in the unsorted world. Furthermore, we assume that our specifications are *unsorted* and do not contain conditional equations.

2. Now we have proved $E \vdash a = b$. The deduction rule **Symmetry** says that *if* $E \vdash a = b$ holds, then so does $E \vdash b = a$. That's all! We have proved the unsurprising fact that $b = a$ follows logically from the above equations E.

Does $f(a) = g(b)$ follow logically from E? That is, can we prove $E \vdash f(a) = g(b)$?

1. Again, we can prove $E \vdash a = b$ because of the **Substitutivity** rule, since the equation $a = b$ is in E.

2. We have now proved that $E \vdash a = b$ holds. The **Congruence** rule says that then $E \vdash f(a) = f(b)$ also holds.

3. Again, by **Substitutivity** w.r.t. the equation $f(x) = g(x)$ and substitution $\sigma = \{x \mapsto b\}$, we also have $E \vdash f(b) = g(b)$.

4. By (2) above, we know that $E \vdash f(a) = f(b)$ holds, and by (3) we know that $E \vdash f(b) = g(b)$ holds. Rule **Transitivity** then says that $E \vdash f(a) = g(b)$ also holds. This is what we wanted to prove! Q.E.D.

The above *proof* of $E \vdash f(a) = g(b)$ can be summarized in the following shorter form (remember that E_1 denotes **Substitutivity**, and so on):

1.	$E \vdash a = b$	(E_1; equation $a = b$)
2.	$E \vdash f(a) = f(b)$	(E_5; from 1)
3.	$E \vdash f(b) = g(b)$	(E_1; equation $f(x) = g(x)$)
4.	$E \vdash f(a) = g(b)$	(E_4; from 2, 3)

Each line in such a deduction/proof must be justified, either by following directly from **Substitutivity** or **Reflexivity**, or by following from claims which have already been justified (and are "higher up" in the deduction) and either **Symmetry**, **Transitivity**, or **Congruence**. ♠

Exercise 94 *Let E still be $\{f(x) = g(x),\ a = b\}$ and prove each of the following claims:*

- $E \vdash f(b) = f(a)$
- $E \vdash f(f(a)) = g(f(a))$
- $E \vdash g(b) = f(a)$
- $E \vdash f(g(z)) = g(f(z))$
- $E \vdash f(g(a)) = g(g(b))$

You do not need to re-prove something you have already proved, if you need that fact later. For instance, we have proved above that $E \vdash f(a) = g(b)$. If you need this fact, you can just use it. We have already proved it once!

Exercise 95 *Let E' be $\{f(a, x) = f(b, x),\ c = d\}$.*

1. *Prove* $E' \vdash f(a,c) = f(b,c)$.
2. *Can you prove* $E' \vdash f(a,c) = f(a,d)$*? Explain.*
3. *Can you prove* $E' \vdash f(a,b) = f(b,c)$*? Explain.*
4. *Can you prove* $E' \vdash f(a,d) = f(a,d)$*? Explain.*
5. *Can you prove* $E' \vdash a = b$*? Explain.*

To show that an equality follows logically from a set of equations you "just" need to give the sequence of deductions, but it is in principle impossible to say that something, like $E' \vdash f(a,a) = f(b,b)$, does *not* hold. We can only say something like "I have tried a bunch of deductions and I still could not deduce the desired property $E' \vdash f(a,a) = f(b,b)$." But this could in principle be either because $E' \vdash f(a,a) = f(b,b)$ does not hold, or because you are not clever enough in your use of the deduction rules. However, Theorem 12 shows that it is easy to prove that "$E \vdash t = t'$ does *not* hold," written $E \nvdash t = t'$, *when the equations* E *are terminating and confluent.*

□ There is another way of proving that some equality $t = t'$ does *not* follow logically from E. Intuitively, it seems obvious that `s(0)` = `0` should not follow logically from the equations in the specification `NAT-ADD`. But how can we prove this trivial fact? According to that algebraic model theory which is beyond the scope of this course, it is the case that $E \vdash t = t'$ implies that (the interpretation of) t equals (the interpretation of) t' in *all* mathematical structures/models/interpretations where the equations E hold. The equations in `NAT-ADD` all hold for the natural numbers, where `0` is supposed to mean the number 0, `s(N)` is supposed to mean 1 plus the interpretation of `N`, and `+` is supposed to mean addition on natural numbers. Therefore, all the equalities that follow from `NAT-ADD` *must* hold for the natural numbers. Clearly, `s(0)` = `0` does not hold for the natural numbers since $1 \neq 0$. We can conclude that `s(0)` = `0` does *not* follow logically from `NAT-ADD`. □

Example 47 `NAT-ADD` $\vdash$ `s(s(0)) + s(0) = s(0) + s(s(0))` holds because it can be derived as follows in equational logic:

1.	`NAT-ADD` $\vdash$ `s(s(0)) + s(0) = s(s(0) + s(0))`	(E_1; equation `s(x) + y = s(x + y)`)
2.	`NAT-ADD` $\vdash$ `0 + s(0) = s(0 + s(0))`	(E_1; equation `s(x) + y = s(x + y)`)
3.	`NAT-ADD` $\vdash$ `0 + s(0) = s(0)`	(E_1; equation `0 + x = x`)
4.	`NAT-ADD` $\vdash$ `s(0 + s(0)) = s(s(0))`	(E_5; from 3)
5.	`NAT-ADD` $\vdash$ `s(0) + s(0) = s(s(0))`	(E_4; from 2, 4)
6.	`NAT-ADD` $\vdash$ `s(s(0) + s(0)) = s(s(s(0)))`	(E_5; from 5)
7.	`NAT-ADD` $\vdash$ `s(s(0)) + s(0) = s(s(s(0)))`	(E_4; from 1, 6)
8.	`NAT-ADD` $\vdash$ `s(0) + s(s(0)) = s(0 + s(s(0))`	(E_1; equation `s(x) + y = s(x + y)`)
9.	`NAT-ADD` $\vdash$ `0 + s(s(0)) = s(s(0))`	(E_1; equation `0 + x = x`)
10.	`NAT-ADD` $\vdash$ `s(0 + s(s(0))) = s(s(s(0)))`	(E_5; from 9)
11.	`NAT-ADD` $\vdash$ `s(0) + s(s(0)) = s(s(s(0)))`	(E_4; from 8, 10)
12.	`NAT-ADD` $\vdash$ `s(s(s(0))) = s(0) + s(s(0))`	(E_3; from 11)
13.	`NAT-ADD` $\vdash$ `s(s(0)) + s(0) = s(0) + s(s(0))`	(E_4; from 7, 12)

♠

Exercise 96 *Prove that* `NAT-ADD` $\vdash$ `s(0) + s(0) = s(s(0))`.

Exercise 97 *Let* `NAT-ADD'` *equal* `NAT-ADD` $\cup \{$`s(0)` $=$ `0`$\}$*, and prove* `NAT-ADD'` $\vdash$ `s(s(0))` $=$ `0`*. Does this seem reasonable?*

The following theorem may not come as a major surprise after seeing how complicated it was to deduce `NAT-ADD` $\vdash$ `s(s(0)) + s(0)` $=$ `s(0) + s(s(0))`.

Theorem 8 *It is undecidable whether*

$$E \vdash t = u$$

holds, even for ground terms t *and* u*.*

The proof of this theorem involves combinatory logic or undecidable semigroups and is way beyond the scope of this course.

We are interested in *reductions* and not in *deductions* in equational logic. Fortunately, that's more or less the same thing:

Theorem 9 (Birkhoff's Theorem) *For any set* E *of equations and terms* t, u *we have*

$$E \vdash t = u \text{ if and only if } t \overset{*}{\leftrightsquigarrow} u.$$

□

Proof. The part "$E \vdash t = u$ implies $t \overset{*}{\leftrightsquigarrow} u$" can be proved by induction the structure of the deduction of $E \vdash t = u$. For example, if $E \vdash t = u$ follows from $E \vdash t = t'$ and $E \vdash t' = u$ by transitivity, then as the induction hypotheses we have that $t \overset{*}{\leftrightsquigarrow} t'$ and $t' \overset{*}{\leftrightsquigarrow} u$ and the desired $t \overset{*}{\leftrightsquigarrow} u$ follows since $\overset{*}{\leftrightsquigarrow}$ is transitive. The other cases (substitutivity, reflexivity, symmetry, and congruence) are almost equally simple.

The other direction, "$t \overset{*}{\leftrightsquigarrow} u$ implies $E \vdash t = u$," can be proved by induction on the length (the number of steps) of the derivation $t \overset{*}{\leftrightsquigarrow} u$ and causes no problems. ♣

Exercise 98 *Prove Theorem 9 in detail.*

□

It follows that it is undecidable whether $t \overset{*}{\leftrightsquigarrow} u$ holds, even when t and u are ground terms. In particular, we have:

Theorem 10 *It is undecidable whether* $t \overset{*}{\rightsquigarrow} u$ *holds, even for ground terms* t *and* u*.*

Proof. Let E^*, for any E, contain each equation $l = r$ in E, and its symmetric version $r = l$. Then $t \overset{*}{\leftrightsquigarrow}_E u$ if and only if $t \overset{*}{\rightsquigarrow}_{E^*} u$. ♣

□ It is decidable whether $t \overset{*}{\leftrightsquigarrow}_E u$ when E only contains equations without variables.

As a corollary to undecidability, we can prove that confluence is undecidable since we can decide $\overset{*}{\leftrightsquigarrow}$ if we can decide confluence as follows (from [2]): Let E^* again contain each equation in E and its symmetric version. Trivially, E^* is confluent. Now add a *new* constant a and two equations $a = t$ and $a = u$ to E^*. Then it is not too difficult to prove that E^* is confluent if and only if $t \overset{*}{\leftrightsquigarrow} u$. □

So far we have some undecidability results, which is not so positive. However, in terminating and confluent specifications it is possible to decide whether $E \vdash t = u$ by, as expected, checking whether $t!$ and $u!$ are the same term:

Theorem 11 *For a terminating and confluent specification E we have*

$$t \overset{*}{\leftrightsquigarrow} u \quad \textit{if and only if} \quad t! = u!$$

for all terms t and u.

□

Proof. "if" directions is trivial: $t \overset{*}{\leftrightsquigarrow} t! = u! \overset{*}{\leftrightsquigarrow} u$.

"only if" direction: By induction on the length of the derivation of $t \overset{*}{\leftrightsquigarrow} u$. If the length is 0, then $t = u$ and $t! = u!$ holds trivially. If the length of the derivation is $n+1$, then $t \leftrightsquigarrow t' \overset{*}{\leftrightsquigarrow} u$ for some t'. The length of $t' \overset{*}{\leftrightsquigarrow} u$ is n, so the induction applies and gives $t'! = u!$. Now, since $t \leftrightsquigarrow t'$ we have that either $t \rightsquigarrow t'$ or $t' \rightsquigarrow t$. If $t \rightsquigarrow t'$, then $t! = t'1 = u!$. If $t' \rightsquigarrow t$, then we have that $t' \overset{*}{\rightsquigarrow} t$ and $t' \overset{*}{\rightsquigarrow} u!$. By confluence, there must be a t^* such that $t \overset{*}{\rightsquigarrow} t^*$ and $u! \overset{*}{\rightsquigarrow} t^*$. Since $u!$ cannot be reduced we have that $t^* = u!$ and we have that $t \overset{*}{\rightsquigarrow} u!$, which implies that $t! = u!$ since $t \overset{*}{\rightsquigarrow} u!$ and $u!$ is irreducible. ♣

□

We end this section by stating its main result:

Theorem 12 *For terminating and confluent E we have*

$$E \vdash t = u \quad \textit{if and only if} \quad t! = u!$$

for all terms t and u.

Proof. Follows directly from Theorem 9 and Theorem 11. ♣

Exercise 99 *We consider again the following specification of the Boolean values:*

```
fmod BOOLEAN is
  sort Boolean .
  ops true false : -> Boolean [ctor] .
  op not_ : Boolean -> Boolean [prec 53] .
  op _and_ : Boolean Boolean -> Boolean [prec 55] .
  op _or_ : Boolean Boolean -> Boolean [prec 59] .
  op _implies_ : Boolean Boolean -> Boolean [prec 61] .
  op if_then_else_fi : Boolean Boolean Boolean -> Boolean .

  vars X Y : Boolean .
  eq true and X = X .
  eq false and X = false .
  eq false or X = X .
  eq true or X = true .
  eq not true = false .
```

```
  eq not false = true .
  eq true implies X = X .
  eq false implies X = true .
  eq if true then X else Y fi = X .
  eq if false then X else Y fi = Y .
endfm
```

1. *Use the deduction rules of equational logic to prove that*

$$\texttt{BOOLEAN} \vdash \texttt{true implies X} = \texttt{(not true) or X}$$

2. *Why is it impossible to prove (the desired?) property*

$$\texttt{BOOLEAN} \vdash \texttt{Y implies X} = \texttt{(not Y) or X}$$

   (Hint: what "operational" properties does the specification have?)

3. *Prove that*

$$\texttt{BOOLEAN} \vdash t \texttt{ implies X} = \texttt{(not } t\texttt{) or X}$$

   holds for each *ground constructor* t *term of sort* `Boolean`.

□

Birkhoff's Theorem in Order-Sorted Specification

When we want to decide whether $E \vdash t = u$ holds, we use Birkhoff's Theorem which says that $E \vdash t = u$ holds if and only if $t \overset{*}{\leftrightsquigarrow}_E u$ holds. In order-sorted specifications this property does not hold for *sort-increasing* specifications:

Example 48

```
fmod NOT-BIRKHOFF is
  sort s s' .
  subsort s' < s .
  op f : s' -> s .
  ops a b : -> s' .
  op c : -> s .
  eq a = c .
  eq b = c .
endfm
```

In equational logic, `a` = `b` holds due to transitivity, and by congruence we have `f(a)` = `f(b)`. However, in pure order-sorted reduction we would not have `f(a)` $\overset{*}{\leftrightsquigarrow}$ `f(b)` because `f(a)` does not reduce to `f(c)`. However, in Maude's (membership equational logic) extension of order-sortedness, "junk terms" like `f(c)` are allowed and we have `f(a)` $\overset{*}{\leftrightsquigarrow}$ `f(b)`. These problems (if they are problems at all for people using Maude) disappear when the equations are sort-decreasing *and confluent*. ♠

□

4.1.1 * Knuth-Bendix Completion

□ We have seen that it is easily decidable whether t and u are logically equivalent in *terminating* and *confluent* specifications. What could we do if the specification is *not* terminating and confluent? One possible answer: try to make the specification terminating and confluent without changing its equational logic semantics.

Knuth-Bendix completion [57] (see also e.g. [22, 56, 2, 23, 3, 55]) is a process which takes a nonconfluent and possibly nonterminating specification into an "equivalent" confluent and terminating specification. By equivalence of E and E' we mean that $E \vdash t = u$ holds if and only if $E' \vdash t = u$ holds, for all t and u.

The input to the process is a specification and a termination ordering $\succ$, such as $\succ_{lpo}$. The output should be an equivalent confluent and terminating specification.

The main idea is the following: Remember that a specification is not confluent if it has a critical pair (t, u) such that t and u have different normal forms $t^! \neq u^!$. However, if $t^!$ is a normal form[2] of t and $u^!$ is a normal form of u, and since (t, u) is a critical pair from a term $l\sigma$, we have

$$t^! \overset{*}{\leftsquigarrow} t \leftsquigarrow l\rho \rightsquigarrow u \overset{*}{\rightsquigarrow} u^!$$

for some term $l\rho$. That is, we have $t^! \leftrightsquigarrow u^!$, and by Birkhoff's Theorem $E \vdash t^! = u^!$ for our specification E. Therefore, we do not really change the equational theory if we add a new equation $t^! = u^!$ or $u^! = t^!$ (depending on whether $t^! \succ u^!$ or $u^! \succ t^!$) to the specification E. For each non-joinable critical pair the completion process therefore adds such an equation. The process terminates successfully when all equations are $\succ$-decreasing and there are no non-joinable critical pairs.

Example 49 The specification

$$\{f(f(x)) = g(x)\}$$

is, as we have seen, nonconfluent since it has a non-joinable critical pair $(f(g(x)),\ g(f(x)))$. If the Knuth-Bendix completion process is parameterized with an ordering $\succ$ such that $f(g(x)) \succ g(f(x))$, then the equation $f(g(x)) = g(f(x))$ is added to the specification.

The next step in the process is to check whether the new specification

$$\{f(f(x)) = g(x),\ f(g(x)) = g(f(x))\}$$

is confluent. A new critical pair is $(g(g(x)),\ f(g(f(x))))$ where both term are obtained by reducing $f(f(g(x)))$. The term $f(g(f(x)))$ reduces to $g(f(f(x)))$ which reduces to $g(g(x))$, so the critical pair is joinable and nothing needs to be done. There are no other interesting overlaps, so the new specification is confluent. ♠

A more advanced feature of some completion processes is that with the introduction of new equations, some other equations can be simplified. This is exploited when dealing with nontermination. Sometimes a "given" or a "created" equation cannot be directed so that the specification is terminating. These equations are "set aside" in the hope that they can be simplified later.

Example 50 Let E be

$$\{f(x, y) = f(y, x),\ g(x, y) = a,\ g(x, y) = f(x, y)\}.$$

The equation $f(x, y) = f(y, x)$ cannot be directed to be $\succ$-decreasing (since it causes nontermination). Therefore, this troubling equation $f(x, y) = f(y, x)$ is "set aside" and the process concentrates on

$$\{g(x, y) = a,\ g(x, y) = f(x, y)\}.$$

[2]Since the specification may not be confluent, t (and u) may have many distinct normal forms, so we can't use the notation $t!$.

Obviously, $(f(x,y),\ a)$ is a non-joinable critical pair, so the equation $f(x,y) = a$ is added to the specification. The troubling equation $f(x,y) = f(y,x)$ can now be simplified by applying the new equation to each side: $f(x,y) \rightsquigarrow a$ and $f(y,x) \rightsquigarrow a$, so the troubling equation is simplified to $a = a$, which can safely be removed. Furthermore, $g(x,y) = f(x,y)$ can be simplified to $g(x,y) = a$. That is,

$$\{g(x,y) = a,\ f(x,y) = a\}$$

is the new confluent and terminating specification which is equivalent to the old nonconfluent and nonterminating specification. ♠

Completion *cannot* always succeed since it is undecidable whether $E \vdash t = u$. If we could always create an E-equivalent specification E' which is confluent and terminating, then we could decide $E \vdash t = u$ by just creating E' from E and then checking whether $t! = u!$ in E'. In practice the completion process may not terminate because new equations are generated which leads to new critical pairs, and so on. Or there may be an equation which can never be simplified or directed so that it becomes $\succ$-decreasing. In these cases, the process gives up! □

4.2 Inductive Theorems

To increase one's confidence in an equational specification, one could want to verify in equational logic that a specification satisfies some expected "semantic" properties[3]. While a Maude execution can test a specification for *single* instances, for example by testing whether `s(s(0)) + s(s(s(0)))` really reduces to `s(s(s(s(s(0)))))`, such tests cannot show that a function, such as `+`, is defined correctly for *all* possible input values.

One could for example be interested in proving the property

$$\texttt{NAT-ADD} \vdash m \texttt{ + } n = n \texttt{ + } m$$

for all numbers m and n. To increase the confidence in our definition of the `concat` and `length` function on lists on page 48 we would like to verify that the expected property

$$\texttt{LIST-NAT1} \vdash \texttt{length(concat(}l\texttt{,}l'\texttt{))} = \texttt{length(}l\texttt{) + length(}l'\texttt{)}$$

holds for *all* lists l and l'.

The property

$$(\dagger) \qquad \texttt{NAT-ADD} \vdash \texttt{M + N} = \texttt{N + M}$$

for *variables* `M` and `N` would imply that m `+` $n = n$ `+` m holds for all numbers m and n. However, the property (†) does *not* hold. (Try to prove the property yourself!) How can we prove that (†) does not hold? A first useful observation is given in the following exercise:

Exercise 100 *Explain that if $E \vdash t = u$, then it is also the case that $E' \vdash t = u$ for any extension E' of E.*

[3] as opposed to syntactic and operational properties such as well-formedness of equations and termination and confluence

We can prove that (†) does not hold, because if it *did* hold, then it follows from Exercise 100 that also the property

$$(\ddagger) \qquad \texttt{NAT-ADD-EXT} \vdash \texttt{M + N} = \texttt{N + M}$$

would hold in the specification

```
fmod NAT-ADD-EXT is
  including NAT-ADD .
  ops a b : -> Nat [ctor] .
  eq a + b = a .
  eq b + a = b .
endfm
```

where `a` and `b` are two *new* elements. It *seems* that the property (‡) does not hold in this specification, since its instance `a + b` = `b + a` does not seem to hold.

□ Can we be sure that `a + b` = `b + a` does not hold in the module `NAT-ADD-EXT`? The new elements `a` and `b` have to obey the equations of `NAT-ADD`, so that `0 + a` = `a`, `0 + b` = `b`, `s(a) + b` = `s(a + b)`, etc. Is it not possible that these equations "force" `a + b` = `b + a`? To disprove (‡) we need to show that it cannot be proved because it is not necessarily true. To do that, we need to exhibit a *model* (or *interpretation*) of `a` and `b`, such that all equations of `NAT-ADD-EXT` are satisfied in the model, and still `a + b` = `b + a` does not hold. Then, we would have shown that `a + b` = `b + a` is not a *logical consequence* of the equations in `NAT-ADD-EXT`, and that the property (‡) therefore does not hold. In the model, we say that `0` means 0, `s` means "+1" and `+` means usual addition. Let `a` mean an extra "infinite" value ∞_1, and let `b` mean another extra "infinite" value ∞_2, satisfying $\infty_1 + \infty_2 = \infty_1$ and $\infty_2 + \infty_1 = \infty_2$. We of course have $n + \infty_i = \infty_i = \infty_i + n$ for all natural numbers n and $i \in \{1, 2\}$. Do all the equations of `NAT-ADD-EXT` hold in this model? Yes, because e.g., `0 + M` = `M` holds for ∞_i since $0 + \infty_i = \infty_i$. Likewise, the equation `s(M) + N` = `s(M + N)` holds for the infinity values (prove this!).

Therefore, we have a model/interpretation in which all equations of the specification holds, and still `a + b` = `b + a` does not hold. Since $E \vdash t = u$ means that $t = u$ holds whenever E holds, we can conclude that it is *not* the case that

$$\texttt{NAT-ADD-EXT} \vdash \texttt{a + b} = \texttt{b + a}$$

holds. Therefore, it cannot be the case that (†) holds. □

This was the hard way to prove `NAT-ADD` $\not\vdash$ `M + N` = `N + M`, which of course also follows directly from Theorem 12 for *variables* `M` and `N` (why?).

The problem with property (†) is of course that it was intended to hold only for the natural numbers $0, 1, 2, \ldots$; that is, for the *ground constructor terms* `0`, `s(0)`, `s(s(0))`, ... Instead of (†) we are therefore really interested in that the property

$$\texttt{NAT-ADD} \vdash m \texttt{ + } n = n \texttt{ + } m$$

holds for *all ground constructor terms* m and n of sort `Nat`. A theorem of this kind, which is required to hold for all ground constructor terms, is (for reasons which will be apparent after reading this section) called an *inductive theorem*.

Example 51 An even simpler example illustrating the difference between inductive theorems and properties with variables is a specification E with two constants a and b and one equation $a = b$. Here, $E \vdash t = b$ holds for *all ground terms* t. However, $E \vdash x = b$ doesn't hold in equational logic for a *variable* x. Why not? Because by Exercise 100, if $E \vdash x = b$ holds, then $E' \vdash x = b$ would also hold for the extension E' of E which adds a constant c to E (without adding any new equations to E). It is obvious that $E' \vdash x = b$ cannot hold, since then (by the **Substitutivity** rule of equational logic), also $E' \vdash c = b$ would hold, which clearly is not the case. ♠

Exercise 101 *Prove that* $E' \vdash c = b$ *does* not *hold for* E' *given in Example 51.*

4.2.1 * Repetition: Mathematical Induction

☐ In this section we briefly recall the main ideas of mathematical induction.

Definition 27 (Mathematical induction on natural numbers) *Let* $P(n)$ *be a property about a natural number* n*. If you can prove the following properties:*

Basis: $P(0)$ *holds, and*

Induction step: $P(k+1)$ *holds, for* any *natural number* k*, assuming as* induction hypothesis *that* $P(k)$ *holds,*

then you have proved that

$P(n)$ *holds for all natural numbers* n*.*

Example 52 The classical high school property to prove by induction is

$$P(n) \equiv\ 0 + 1 + 2 + \cdots + n = \frac{n \cdot (n+1)}{2}.$$

We prove by induction the property that $P(n)$ holds for all natural numbers n by induction as follows:

Basis: Prove $P(0)$, i.e, $0 = \frac{0 \cdot (0+1)}{2}$ which holds.

Induction step: We have to prove $P(k+1)$ for any k, i.e.,

$$0 + 1 + 2 + \cdots + (k+1) = \frac{(k+1) \cdot ((k+1)+1)}{2}$$

assuming the induction hypothesis $P(k)$, namely

$$0 + 1 + 2 + \cdots + k = \frac{k \cdot (k+1)}{2}.$$

Let's prove

$$0 + 1 + 2 + \cdots + k + (k+1) = \frac{(k+1) \cdot ((k+1)+1)}{2}.$$

Using the induction hypothesis, the left-hand side of the above equality reduces to

$$\frac{k \cdot (k+1)}{2} + (k+1)$$

which equals

$$\frac{k \cdot (k+1) + 2k + 2}{2}$$

which again equals the right-hand side

$$\frac{(k+1)\cdot(k+2)}{2}.$$

Since we have showed both the **Basis** and the **Induction step** we can conclude that

$$0+1+2+\cdots+n=\frac{n\cdot(n+1)}{2}$$

holds for all natural numbers n. ♠

Is the induction schema intuitively correct? If you prove both $P(0)$ and $P(k)$ implies $P(k+1)$ for *any* number k, then

- $P(0)$ holds (proved),
- $P(1)$ holds since $P(0)$ holds, and we have proved that for *any* k, $P(k)$ implies $P(k+1)$. Since k could be any number, we could let $k=0$, in which case $P(0)$ implies $P(1)$. Since $P(0)$ holds, $P(1)$ must also hold.
- $P(2)$ must hold since $P(1)$ holds, and $P(1)$ implies $P(2)$ (according to the induction step, this time with $k=1$),
- $P(3)$ must hold since $P(2)$ holds and $P(2)$ implies $P(3)$,
- ...

Any introductory textbook in mathematical logic says that the induction schema in Def. 27 is equivalent to the following:

Definition 28 (Mathematical induction on natural numbers II)

If *for* any *natural number k, the property $P(k)$ holds when you can assume (as induction hypotheses) $P(k')$ for all $k'<k$*

then *$P(n)$ holds for all natural numbers n.*

Exercise 102 *Show that the above two versions of the induction principle are equivalent.*

You may sometimes want to change the **Basis** value to 1, or 2, or ...:

Exercise 103 *A basic property in "complexity" courses is that the factorial function is exponential. Prove that the property $n!\geq 2^n$ holds for all natural numbers $n\geq 4$.*

□

4.2.2 Induction in the Module `NAT-ADD`

We can give an induction schema for the data type `Nat` with constructors

```
op 0 : -> Nat [ctor] .
op s : Nat -> Nat [ctor] .
```

as follows:

Basis: Prove $P(\mathtt{0})$.

Induction step: For *any* ground constructor term t of sort `Nat`, prove $P(\mathtt{s}(t))$ when you can assume the induction hypothesis $P(t)$.

If you have proved both **Basis** and the **Induction step**, then you have proved $P(t)$ for all ground constructor terms t of sort `Nat`.

Example 53 We prove by induction that

$$\texttt{NAT-ADD} \vdash t \texttt{ + 0} = t$$

holds for all ground constructor terms t.

Basis: The property $\texttt{NAT-ADD} \vdash \texttt{0 + 0} = \texttt{0}$ holds by the **Substitutivity** property of equational logic w.r.t the equation `0 + M = M`.

Induction step: Assuming the induction hypothesis

$$\texttt{NAT-ADD} \vdash t \texttt{ + 0} = t$$

for any ground constructor term t, we have to prove

$$\texttt{NAT-ADD} \vdash \texttt{s}(t) \texttt{ + 0} = \texttt{s}(t)$$

which holds since $\texttt{s}(t) \texttt{ + 0} = \texttt{s}(t \texttt{ + 0})$ follows from the equation `s(M) + N = s(M + N)` in `NAT-ADD`, and $\texttt{s}(t \texttt{ + 0}) = \texttt{s}(t)$ follows from the induction hypothesis (and the **Congruence** property of equational logic). The desired property

$$\texttt{NAT-ADD} \vdash \texttt{s}(t) \texttt{ + 0} = \texttt{s}(t)$$

then holds by **Transitivity**.

We have therefore proved

$$\texttt{NAT-ADD} \vdash t \texttt{ + 0} = t$$

for all ground constructor terms t of sort `Nat`. ♠

Exercise 104 *Is it possible to prove*

$$\texttt{NAT-ADD} \vdash \texttt{M + 0} = \texttt{M}$$

in equational logic, for `M` *a variable of sort* `Nat`*?*

Example 54 We prove commutativity of `+` in `NAT-ADD` for all ground constructor terms. That is, we prove the property that for *all* ground constructor terms m it is the case that for all ground constructor terms n

$$\texttt{NAT-ADD} \vdash m \texttt{ + } n = n \texttt{ + } m.$$

The proof is by "induction on m:"

Basis: It must in the base case be proved that for all ground constructor terms n

$$\texttt{NAT-ADD} \vdash \texttt{0} + n = n + \texttt{0}.$$

This property holds for *all* n if it holds for an "arbitrary" n. Let therefore n be *any arbitrary* ground constructor term of sort `Nat`. The left-hand side `0 +` n equals n in `NAT-ADD` due to the equation `0 + M = M` and the **Substitutivity** property of equational logic. The right-hand side n `+ 0` also equals n due to the recently proved property that

$$\texttt{NAT-ADD} \vdash n \texttt{ + 0} = n$$

holds for all ground constructor terms n.

Induction step: The property must be proved for $\texttt{s}(t)$ for m, assuming the property for t. That is, the property

$$\texttt{NAT-ADD} \vdash \texttt{s}(t) + n = n + \texttt{s}(t)$$

must be proved for all ground constructor terms n assuming that

$$\texttt{NAT-ADD} \vdash t + n = n + t$$

holds for all n.

The left-hand side of the equality to be proved reduces to $\texttt{s}(t \texttt{ + } n)$ by the equation `s(M) + N = s(M + N)` in `NAT-ADD`. By the induction hypothesis (and **Congruence**), $\texttt{s}(t \texttt{ + } n)$ equals $\texttt{s}(n \texttt{ + } t)$. This last term is still not the same as the right-hand side of the equation to be proved. I couldn't find any way of using the equations in `NAT-ADD` which made these two sides equal. However, intuitively $\texttt{s}(n \texttt{ + } t)$ *should* equal $n + \texttt{s}(t)$. What do we do? We prove the *lemma*

$$\texttt{NAT-ADD} \vdash \texttt{s}(n \texttt{ + } t) = n \texttt{ + } \texttt{s}(t)$$

for all n by induction. If this lemma can be proved, then the left- and the right-hand sides of the property to prove in the induction step are equal.

Proof of lemma:

Basis: $\texttt{NAT-ADD} \vdash \texttt{s}(\texttt{0 + } t) = \texttt{0 + } \texttt{s}(t)$ holds because $\texttt{s}(\texttt{0 + } t) = \texttt{s}(t)$ (by equation `0 + M = M`) which is then equal to $\texttt{0 + } \texttt{s}(t)$ (by using the equation `0 + M = M` "backwards").

Induction step: (Prove the lemma for $\texttt{s}(u)$ for n, assuming the lemma for u for n.) The property

$$\texttt{NAT-ADD} \vdash \texttt{s}(\texttt{s}(u) \texttt{ + } t) = \texttt{s}(u) \texttt{ + } \texttt{s}(t)$$

must be proved, assuming the induction hypothesis

$$\texttt{NAT-ADD} \vdash \texttt{s}(u \texttt{ + } t) = u \texttt{ + } \texttt{s}(t).$$

The left-hand side $\texttt{s}(\texttt{s}(u) \texttt{ + } t)$ equals $\texttt{s}(\texttt{s}(u \texttt{ + } t))$ by the second equation in `NAT-ADD`, and this latter term equals $\texttt{s}(u \texttt{ + } \texttt{s}(t))$ by the induction hypothesis. This latter term equals the right-hand side $\texttt{s}(u) \texttt{ + } \texttt{s}(t)$ by using the equation `s(M) + N = s(M + N)` "backwards."

Since the lemma holds, the step $\mathtt{s}(n \ \mathtt{+} \ t) = n \ \mathtt{+} \ \mathtt{s}(t)$ in the main proof holds, and therefore the commutativity property is proved.

♠

The above example illustrated that one may sometimes get "stuck" during a proof, and may need to prove useful lemmas (by induction).

Exercise 105 *Prove the associativity property*

$$\mathtt{NAT\text{-}ADD} \vdash (n_1 \ \mathtt{+} \ n_2) \ \mathtt{+} \ n_3 = n_1 \ \mathtt{+} \ (n_2 \ \mathtt{+} \ n_3)$$

for all ground constructor terms n_1, n_2, n_3 *of sort* `Nat`.

We will later in this section argue that the described induction schema is "correct" (or *sound*, in the sense that no false conclusions can be drawn). To intuitively understand why the induction schema is correct in the case of `NAT-ADD`, assume that $P(\mathtt{0})$ holds, and that $P(t)$ implies that $P(\mathtt{s}(t))$ holds for any ground constructor term t. Then, $P(\mathtt{0})$ must hold because of the base case. By the proved fact "$P(t)$ implies $P(\mathtt{s}(t))$ for any t" it follows as a special case that $P(\mathtt{s}(\mathtt{0}))$ must hold. Since $P(\mathtt{s}(\mathtt{0}))$ holds and "$P(t)$ implies $P(\mathtt{s}(t))$ for any t" it must also be the case that $P(\mathtt{s}(\mathtt{s}(\mathtt{0})))$ must hold, etc.

There is nothing "special" about the signature

```
sort Nat .
op 0 : -> Nat [ctor] .
op s : Nat -> Nat [ctor] .
```

The signature

```
sort s .
op a : -> s [ctor] .
op f : s -> s [ctor] .
```

is "the same" signature which has exactly the same "structure" as the signature for `Nat`. Just the *names* `Nat`, `0`, and `s` are changed to `s`, `a`, and `f`. To prove that a property $Q(t)$ holds for all ground constructor terms t of sort `s` in the above signature it is enough to prove

Basis: that $Q(\mathtt{a})$ holds; and

Induction step: that $Q(\mathtt{f}(t))$ holds, assuming that (the induction hypothesis) $Q(t)$ holds, for any ground constructor term t of sort `s`.

Exercise 106 *Prove that*

$$\mathtt{NAT\text{-}RENAMED} \vdash \mathtt{g}(t, \mathtt{a}) = t$$

holds for all ground constructor terms t *in the module*

```
fmod NAT-RENAMED is
  sort s .
  op a : -> s [ctor] .
  op f : s -> s [ctor] .
  op g : s s -> s .
  vars X Y : s .
  eq g(a, X) = X .
  eq g(f(X), Y) = f(g(X, Y)) .
endfm
```

This example indicates that the induction schema can be applied not only to the module `NAT-ADD`, but to *any* data type. For example, if the sort `S` has the constructors

```
  ops a b : -> S [ctor] .
  ops f g : S -> S [ctor] .
```

then for $Q(t)$ to hold for all ground terms t of sort `S` would mean that $Q(\texttt{a})$, $Q(\texttt{b})$, $Q(\texttt{f(a)})$, $Q(\texttt{f(b)})$, $Q(\texttt{g(a)})$, $Q(\texttt{g(b)})$, $Q(\texttt{f(f(a))})$, $Q(\texttt{f(f(b))})$, $Q(\texttt{f(g(a))})$, $Q(\texttt{f(g(b))})$, $Q(\texttt{g(f(a))})$, $Q(\texttt{g(f(b))})$, $Q(\texttt{g(g(a))})$, $Q(\texttt{g(g(b))})$, $Q(\texttt{f(f(f(a)))})$, etc all hold. In this case the induction schema for proving $Q(t)$ for all ground terms t of sort `S` would amount to proving

Basis: that $Q(\texttt{a})$ and $Q(\texttt{b})$ both hold; and

Induction step: that both $Q(\texttt{f}(t))$ and $Q(\texttt{g}(t))$ hold, assuming that (the induction hypothesis) $Q(t)$ holds, for an arbitrary ground constructor term t of sort `S`.

Exercise 107 *Try to convince yourself of the soundness of the above induction schema by showing that if the* ***Basis*** *and* ***Induction step*** *cases have been proved, then there cannot exist a ground constructor term t such that $Q(t)$ does* not *hold.*

4.2.3 Induction over Data Types

The ideas above can be generalized to an induction schema for proving inductive theorems in *any* many-sorted specification as follows:

Definition 29 (Data type induction) *Let E be a many-sorted equational specification, and let $Q(t)$ be a "statement" about a term t. Then, to prove that*

$$E \vdash Q(t)$$

holds for all ground constructor terms t of a sort s it is enough to prove that

$$E \vdash Q(f(t_1, \ldots, t_n))$$

holds for each constructor of the form

```
  op f : s1 ... sn -> s [ctor]
```

of sort s for arbitrary ground constructor terms $t_1, \ldots, t_n$. (Note that n may be 0 in the above definition, in which case f is a constant.) In each of these proofs we may assume that the induction hypotheses

$$E \vdash Q(t_i)$$

*hold for each i where $s_i = s$. (The proof steps in which no induction hypothesis can be used are often referred to as the **Basis** cases; the others are referred to as **Induction steps**.)*

Example 55 The module `NAT-ADD` has two constructors, namely `0` and `s`. To prove that $Q(t)$ holds for all ground constructor terms t of sort `Nat`, the properties $Q(\mathtt{0})$ and $Q(\mathtt{s}(t))$ have to be proved according to Def. 29. In the proof of $Q(\mathtt{s}(t))$ one may assume the induction hypothesis $Q(t)$ since t is a ground constructor term of sort `Nat`. ♠

Exercise 108 *Show that the other induction schemas presented in this section are instances of the general induction schema given in Def. 29.*

Example 56 To prove that $M \vdash Q(t)$ holds for all ground constructor terms t of sort `s` in the module

```
fmod M is
  sorts s s' .
  ops a b : -> s [ctor] .
  ops c d : -> s' [ctor] .
  ops f g : s s' -> s [ctor] .
  op h : s' s s' -> s' [ctor] .
  op k : s s' s -> s [ctor] .
  ops l p : s -> s .
  ops d : s -> s' .
  ...    *** variables and equations
endfm
```

one would need to prove

Basis: $Q(\mathtt{a})$ and $Q(\mathtt{b})$

Induction steps:

- $Q(\mathtt{f}(t, t'))$ and $Q(\mathtt{g}(t, t'))$ for arbitrary ground constructor terms t and t', in both cases assuming the induction hypothesis $Q(t)$.
- $Q(\mathtt{k}(t_1, t_2, t_3)$ for arbitrary ground constructor terms t_1, t_2, and t_3. In the proof of this property one may assume *both* $Q(t_1)$ and $Q(t_3)$.

♠

Exercise 109 *This exercise concerns the proof schema in Example 56.*

1. *Explain why it is not necessary (or possible) to prove $Q(\mathtt{c})$ and $Q(\mathtt{d})$.*

2. *Explain why $Q(t')$ cannot be assumed as an induction hypothesis when proving $Q(\mathtt{f}(t, t'))$ and $Q(\mathtt{g}(t, t'))$.*

3. *Describe the induction schema for proving $P(u)$ for all ground constructor terms u of sort* `s'` *in the module* `M` *above.*

Example 57 In this example we prove a property of the data type of lists of natural numbers defined in Sections 2.1.5 and 2.3.4:

```
fmod LIST-NAT1 is
  protecting NAT-ADD .

  sort List .
  op nil : -> List [ctor] .
  op __ : List Nat -> List [ctor] .
  op length : List -> Nat .           *** No of elements in a list
  op concat : List List -> List .     *** Concatenate two lists

  var N : Nat .   vars L L' : List .

  eq length(nil) = 0 .
  eq length(L N) = s(length(L)) .
  eq concat(L, nil) = L .
  eq concat(L, L' N) = concat(L, L') N .
  ...
endfm
```

To increase our confidence in this specification we would like to prove the property

$$\mathtt{LIST\text{-}NAT1} \vdash \mathtt{length}(\mathtt{concat}(l, l')) = \mathtt{length}(l) \ \mathtt{+}\ \mathtt{length}(l')$$

for all ground constructor terms l and l' of sort `List`. (The length of the concatenation of two lists should obviously equal the sum of the length of each list.) The induction schema for proving $Q(l)$ for all ground constructor terms l of sort `List` is

Basis: Prove $Q(\mathtt{nil})$.

Induction step: Prove $Q(l\ n)$ for any ground constructor terms l and n (of sorts `List` and `Nat` respectively), assuming the induction hypothesis $Q(l)$.

We prove the above property by induction on l'. (We could also have tried to do induction on l. Our choice is motivated by the fact that `concat` is defined inductively w.r.t. its *second* argument, so an induction on l would not succeed.) That is, the property $Q(l')$ is

> for all ground constructor terms l of sort `List`,
>
> $$\mathtt{LIST\text{-}NAT1} \vdash \mathtt{length}(\mathtt{concat}(l, l')) = \mathtt{length}(l) \ \mathtt{+}\ \mathtt{length}(l')$$

The proof goes as follows:

Basis: (`nil` for l'): Have to prove the property

$$\texttt{LIST-NAT1} \vdash \texttt{length}(\texttt{concat}(l, \texttt{nil})) = \texttt{length}(l) \texttt{ + length}(\texttt{nil})$$

for all l. The left-hand side $\texttt{length}(\texttt{concat}(l, \texttt{nil}))$ reduces to $\texttt{length}(l)$ because of the equation `concat(L, nil) = L`.

The right-hand side $\texttt{length}(l)$ `+` $\texttt{length}(\texttt{nil})$ reduces to $\texttt{length}(l)$ `+ 0`, and by a previous result m `+ 0` $= m$[4] for all ground constructor terms m of sort `Nat` we have that $\texttt{length}(l)$ `+ 0` equals $\texttt{length}(l)$, which equals the left-hand side.

Induction step: (l' n for l'): Here we have to prove

$$\texttt{LIST-NAT1} \vdash \texttt{length}(\texttt{concat}(l, l'\ n)) = \texttt{length}(l) \texttt{ + length}(l'\ n)$$

for any l, l', n assuming the induction hypothesis

$$\texttt{LIST-NAT1} \vdash \texttt{length}(\texttt{concat}(l, l')) = \texttt{length}(l) \texttt{ + length}(l')$$

We prove the property as follows:

$\texttt{length}(\texttt{concat}(l, l'\ n))$	(left-hand side)
$= \texttt{length}(\texttt{concat}(l, l')\ n)$	(def. of `concat`)
$= \texttt{s}(\texttt{length}(\texttt{concat}(l, l'))$	(def. of `length`)
$= \texttt{s}(\texttt{length}(l) \texttt{ + length}(l'))$	(induction hypothesis)
$= \texttt{s}(\texttt{length}(l') \texttt{ + length}(l))$	(commutativity of `+`)
$= \texttt{s}(\texttt{length}(l')) \texttt{ + length}(l)$	(equation for `+`)
$= \texttt{length}(l) \texttt{ + s}(\texttt{length}(l'))$	(commutativity of `+`)
$= \texttt{length}(l) \texttt{ + length}(l'\ n)$	(rhs., def. of `length`)

♠

Exercise 110 *Define the function* `reverse` *on lists in the above data type (or recall your solution from Exercise 20) and prove that you get the original list back if you reverse it twice:*

$$\texttt{LIST-NAT1} \vdash \texttt{reverse}(\texttt{reverse}(l)) = l$$

for all ground constructor terms l of sort `List`*.*

Example 58 In this example we prove that the number of elements in a binary tree is the same as the number of elements in the reversed tree.

We recall from Section 2.3.5 our definition of binary trees:

```
fmod BINTREE-NAT1 is
  protecting NAT-ADD .

  sort BinTree .
  op niltree : -> BinTree [ctor] .
```

[4]The use of m `+ 0` $= m$ is allowed under the assumption that all ground terms of sort `Nat` are indeed equal to some ground constructor term. That's of course what `ctor` is intended to state.

```
  op bintree : BinTree Nat BinTree -> BinTree [ctor] .
  op size : BinTree -> Nat .
  op reverse : BinTree -> BinTree .

  vars BT BT' : BinTree .   var N : Nat .

  eq size(niltree) = 0 .
  eq size(bintree(BT, N, BT')) = s(0) + (size(BT) + size(BT')) .
  eq reverse(niltree) = niltree .
  eq reverse(bintree(BT, N, BT')) = bintree(reverse(BT'), N, reverse(BT)) .
  ...
endfm
```

We prove

$$\texttt{BINTREE-NAT1} \vdash \texttt{size}(\texttt{reverse}(t)) = \texttt{size}(t)$$

for all ground constructor terms t of sort `BinTree`. To prove a property $Q(t)$ for all ground constructor terms t of sort `BinTree` by induction one needs, according to Definition 29, to prove

Basis: $Q(\texttt{niltree})$

Induction step: $Q(\texttt{bintree}(t_1, n, t_2))$, assuming as induction hypotheses *both* $Q(t_1)$ and $Q(t_2)$.

In our particular case we must prove

Basis: $\texttt{BINTREE-NAT1} \vdash \texttt{size}(\texttt{reverse}(\texttt{niltree})) = \texttt{size}(\texttt{niltree})$, which holds since both sides equal `0`.

Induction step: Here we must prove

$$\texttt{BINTREE-NAT1} \vdash \texttt{size}(\texttt{reverse}(\texttt{bintree}(t_1, n, t_2))) = \texttt{size}(\texttt{bintree}(t_1, n, t_2))$$

assuming both

$$\texttt{BINTREE-NAT1} \vdash \texttt{size}(\texttt{reverse}(t_1)) = \texttt{size}(t_1)$$

and

$$\texttt{BINTREE-NAT1} \vdash \texttt{size}(\texttt{reverse}(t_2)) = \texttt{size}(t_2).$$

The left-hand side reduces to

$$\texttt{size}(\texttt{bintree}(\texttt{reverse}(t_2), n, \texttt{reverse}(t_2)))$$

which reduces to

$$\texttt{s}(\texttt{size}(\texttt{reverse}(t_2)) \texttt{ + } \texttt{size}(\texttt{reverse}(t_1)))$$

by the definition of `size`. Now we use the induction hypotheses on both sides and reduce the above expression to

$$\texttt{s}(\texttt{size}(t_2) \texttt{ + } \texttt{size}(t_1)).$$

The right-hand side reduces to this same expression in one step, so we have proved the induction step, and therefore the whole property.

♠

Exercise 111 *Prove*

$$\texttt{BINTREE-NAT1} \vdash \texttt{reverse}(\texttt{reverse}(t)) = t$$

for all ground constructor terms t *of sort* `BinTree`.

Induction Proofs in Order-Sorted Specifications

If s' is a subsort of a sort s in an order-sorted specification, then a constructor of sort s' is also a constructor of sort s (since each s'-term is also an s-term). Therefore, $Q(f(t_1, \ldots, t_n))$ must be proved for all constructors of sort s' to prove $Q(t)$ for all ground constructor terms t of sort s. Likewise, induction hypotheses $Q(t_i)$ may also be assumed for all terms t_i of sort s'.

Example 59 In this example we consider our list specification `LIST-INT` from page 114, where the constructors and the function `length` are declared as follows:

```
fmod LIST-INT is
  protecting INT .

  sorts List NeList .
  subsorts Int < NeList < List .

  op nil : -> List [ctor] .
  op __ : List List -> List [assoc id: nil ctor] .
  op __ : NeList NeList -> NeList [assoc id: nil ctor] .
  op length : List -> Int .

  var I : Int .   var L : List .

  eq length(nil) = 0 .
  eq length(I L) = 1 + length(L) .
```

we are again interested in proving that the length of a concatenation of two lists is the sum of the length of each list:

$$\texttt{LIST-NAT1} \vdash \texttt{length}(l\ l') = \texttt{length}(l)\ \texttt{+}\ \texttt{length}(l')$$

for all ground constructor terms l and l' of sort `List`. The proof is by "induction on l." Recall that each integer is also a constructor for lists since `Int` is a subsort of `List`.

Basis (`nil` for l): It must in this case be proved that

$$\texttt{LIST-NAT1} \vdash \texttt{length}(\texttt{nil}\ l') = \texttt{length}(\texttt{nil})\ \texttt{+}\ \texttt{length}(l')$$

holds for all l'.

The left-hand side `length(nil` l'`)` equals `length(`l'`)` since `nil` $l' = l'$ by the `id: nil` attribute of `__`. The right-hand side `length(nil) + length(`l'`)` equals `0 + length(`l'`)` (by the definition of `length`) which is equal to `length(`l'`)`.

Basis (n for l): In this case one needs to prove

$$\texttt{LIST-NAT1} \vdash \texttt{length}(n\ l') = \texttt{length}(n)\ \texttt{+}\ \texttt{length}(l')$$

for all ground constructor terms n of sort `Int` and l' of sort `List`. This is trivial.

Induction step: In this case we prove the property with $l_1\ \ l_2$ for l, i.e.,

$$\texttt{LIST-NAT1} \vdash \texttt{length}((l_1\ \ l_2)\ l') = \texttt{length}(l_1\ \ l_2)\ \texttt{+}\ \texttt{length}(l')$$

for all l' of sort `List`. We may assume that induction hypotheses

$$\texttt{LIST-NAT1} \vdash \texttt{length}(l_1\ l') = \texttt{length}(l_1)\ \texttt{+}\ \texttt{length}(l')$$

and

$$\texttt{LIST-NAT1} \vdash \texttt{length}(l_2\ l') = \texttt{length}(l_2)\ \texttt{+}\ \texttt{length}(l')$$

for all lists l'.

To prove

$$\texttt{LIST-NAT1} \vdash \texttt{length}((l_1\ \ l_2)\ l') = \texttt{length}(l_1\ \ l_2)\ \texttt{+}\ \texttt{length}(l')$$

for all l' it is enough to prove it for some *arbitrary* l'.

The left-hand side $\texttt{length}((l_1\ \ l_2)\ l')$ equals $\texttt{length}(l_1\ \ (l_2\ \ l'))$ by the `assoc` attribute of the operator `__`. Since the induction hypothesis

$$\texttt{LIST-NAT1} \vdash \texttt{length}(l_1\ l') = \texttt{length}(l_1)\ \texttt{+}\ \texttt{length}(l')$$

was supposed to hold *for all* lists l', it also applies to the list $l_2\ \ l'$, and therefore $\texttt{length}(l_1\ \ (l_2\ \ l'))$ equals $\texttt{length}(l_1)\ \texttt{+}\ \texttt{length}(l_2\ \ l')$. Now, the induction hypothesis for l_2 applies and we get $\texttt{length}(l_1)\ \texttt{+}\ \texttt{length}(l_2)\ \texttt{+}\ \texttt{length}(l')$.

The right-hand side $\texttt{length}(l_1\ l_2)\ \texttt{+}\ \texttt{length}(l')$ can be reduced to $\texttt{length}(l_1)\ \texttt{+}\ \texttt{length}(l_2)\ \texttt{+}\ \texttt{length}(l')$ because the induction hypothesis for l_1 was assumed to hold for *all* lists l', therefore also for l_2. Q.E.D.

♠

Exercise 112 *Define the function* `reverse` *in the above module and prove*

$$\ldots \vdash \texttt{reverse}(\texttt{reverse}(l)) = l$$

for all ground constructor terms l of sort `List`.

Hint: *This exercise becomes rather trivial if you define the function* `reverse` *with one equation for each constructor.*

4.2.4 * Soundness of Induction over Data Types

□ Not written. □

Part II

Specification and Analysis of Distributed Systems in Maude

Chapter 5

Modeling Dynamic and Concurrent Systems in Rewriting Logic

This chapter gives a brief introduction to *rewriting logic*, which we use for modeling dynamic systems and for reasoning about concurrency in a system. This chapter may well be read together with Chapter 6 which explains how rewriting logic models can be executed in Maude.

5.1 Dynamic Systems

In Part I of this compendium we have specified *static* systems, such as data types, by defining what terms are *equivalent*. There is (mathematically) no *dynamic* behavior in an equational specification. Two expressions are either equivalent or there is no relation between them in equational logic. Due to *symmetry* of the equivalence relation, both

$$\texttt{length(2 5 7)} = \texttt{3}$$

and

$$\texttt{3} = \texttt{length(2 5 7)}$$

hold.

For *dynamic* systems, where the system *evolves* (or *changes*) with time, this equational setting may not make much mathematical or logical sense. To exemplify a dynamic system, let us look at a simplified model of the life of a human being.

In this example, a term

```
person("Peter", 38)
```

denotes a person with name `"Peter"` who is 38 years old. To simulate the life (or at least the aging) of this gentleman, it seems obvious that the next state should be

```
person("Peter", 39).
```

How should such a person-simulating system be modeled? An immediate first thought could be to use an equation

```
person(X, N) = person(X, N + 1).
```

In such a system it would be logically true that

$$\texttt{person("Peter", 38)} = \texttt{person("Peter", 39)},$$

but this change would be *reversible*—due to the *symmetry property* of equational logic—so that also

$$\texttt{person("Peter", 39)} = \texttt{person("Peter", 38)},$$

and (by transitivity)

$$\texttt{person("Peter", 38)} = \texttt{person("Peter", 20)}$$

would hold mathematically. However, in the system we are modeling, these properties intuitively should *not* hold, no matter how much one may wish that they did.

Change is usually not reversible in dynamic systems. One cannot undo the bizarre incident where three men in a mountainous country were bombed after an unmanned airplane "saw" that one of the men was fairly tall[1]. Likewise, if you today have \$100 in your bank account, and tomorrow only \$50 after celebrating the Super Bowl or a finished homework assignment, it means that your bank account *can* evolve from a state in which it contains \$100 to a state in which it contains only \$50. But a bank account with \$100 is *not* the *same* as a bank account with \$50.

In other words, it seems that we cannot use just equational specification/logic for specifying a dynamic (or *changing* or *evolving*) system. Instead, we use *rewriting logic* [69, 73, 6], which extends algebraic equational specification techniques, for modeling dynamic systems.

5.1.1 Reactive Systems

A dynamic computer system is quite often a *reactive* system, which interacts with its *environment* by *reacting* to input from the environment by changing its state and/or by providing some output.

The prototypical example of a reactive system/program is an operating system. Such a system reacts to input, say commands such as `ls` and `rm` from the terminal, by providing output (such as the list of all files) and/or by changing the state of the system (e.g. by removing some files). Most other larger computer systems we have mentioned in the introduction, such as airplane and nuclear power plant controllers, are also reactive systems, which react to e.g. reported low values of flight fuel throughput by temporarily shutting down an engine(?).

Reactive systems are normally nonterminating and nondeterministic, and their properties of interest are "the current state" of the system and their response to stimulus from the environment.

5.2 Rewrite Rules

The problem with using equations to model dynamic behavior is, as indicated above, that equality is symmetric, so that all "change" is reversible. In rewriting logic, the dynamic

[1] The first version of this chapter was written in March 2002.

behavior is instead modeled by *rewrite rules*. Rewrite rules are essentially "one-way equations" that define the local atomic transition patterns of a system. The "birthday" action in our simple example could be modeled by a rewrite rule

$$\texttt{person(X, N)} \longrightarrow \texttt{person(X, N + 1)}$$

for appropriate variables `X` and `N`.

Given the local atomic transitions in the form of rewrite rules, the *deduction rules of rewriting logic* (see page 163) define how the state of a system *may* evolve. A rewriting logic *sequent* (or *formula*)

$$t \longrightarrow t'$$

is intended to mean that the *state* t *can* change to (or *evolve* to, or *reach*) the *state* t' using the rewrite rules zero or more times. We will use "term" and "state" interchangeably since the state of a system is given by a term.

Exercise 113 *Which of the following sequents should* intuitively *hold, given a rewrite rule*

$$\texttt{person(X, N)} \longrightarrow \texttt{person(X, N + 1)}$$

(for variables `X` *and* `N`*)?*

1. `person("Peter", 38)` $\longrightarrow$ `person("Peter", 39)`
2. `person("Peter", 38)` $\longrightarrow$ `person("Peter", 998)`
3. `person("Ollie", 38)` $\longrightarrow$ `person("Ollie", 39)`
4. `person("Peter", 39)` $\longrightarrow$ `person("Peter", 38)`
5. `person("Peter", 38)` $\longrightarrow$ `person("Ollie", 39)`

5.2.1 Rule Labels

A rewrite rule can be equipped with a *label* which names the *action* or *event* that causes the state change. In our example, the labeled rule could be

$$\mathit{birthday}: \ \texttt{person(X, N)} \longrightarrow \texttt{person(X, N + 1)}$$

or

$$\mathit{getting\_older}: \ \texttt{person(X, N)} \longrightarrow \texttt{person(X, N + 1)}.$$

The label does not influence computation/deduction in rewriting logic.

5.2.2 What is the Difference?

What is *really* the difference between the *equation*

$$\texttt{person(X, N)} = \texttt{person(X, N + 1)}$$

and the *rewrite rule*

$$\textit{birthday} : \texttt{person(X, N)} \longrightarrow \texttt{person(X, N + 1)}?$$

When executing in Maude, the *equation* is only used "from left to right," and it was mentioned that a rewrite rule is just a "one-directional equation." That's true, and in the sense of execution of one equation/rule in Maude, the differences are not *that* great. However, we are not only concerned about execution but also about *modeling* and *reasoning* about systems. (These are indeed the main goals of this course.) In the two specifications above (equation vs. rule) we specify two completely different systems. The *equation* specifies an "ageless" zombie who is both 38, and 112, and 2 years old at the same time. The *rewrite rule* above specifies a "person" who is initially k years old, and then *becomes/changes to* $k + 1$ years old, then becomes/changes to $k + 2$ years old. The latter specification is probably more what we want.

Finally, as we will see, since a rewrite rule logically means something different than an equation, Maude will *execute* a *rewrite* specification differently than an *equational* specification. Maude can execute an equational specification without worrying about which equation to apply. This is sufficient since in equational specifications, each expression has only *one* result. A rewrite specification may model a system which is both nondeterministic and nonterminating, and therefore the above approach is not sufficient as some "solutions" would be lost by applying the rewrite rules in that way. We will discuss the execution of rewrite rules in Chapter 6.

5.2.3 Rewriting is "Modulo" the Equational Theory

The data of a dynamic system are defined as data types with equations as usual. In our person example we had the rule

$$\textit{birthday} : \texttt{person(X, N)} \longrightarrow \texttt{person(X, N + 1)}$$

where '`+`' is a function defined equationally (or, equivalently, built-in in the case of natural numbers). We expect that this specification means that `person("Lizzie", 35)` can change to `person("Lizzie", 36)` in *one* step, since this state is equivalent to `person("Lizzie", 35 + 1)`. Therefore, rewriting is *modulo* the equations E in the specification in the sense that if $t \longrightarrow t'$ and $E \vdash t = u$ and $E \vdash t' = u'$ then $u \longrightarrow u'$.

5.3 Rewriting Logic Specifications

A *rewriting logic specification* is a membership equational logic specification extended with labeled rewrite rules which describe the elementary local transitions (or *state changes*) of a system.[2]

[2]Although we use membership equational logic as the underlying equational logic in the definition below, rewriting logic is actually *parametric* in the underlying equational logic, which could be unsorted, order-sorted, membership, or some other kind of equational logic.

Definition 30 (Rewriting logic specification) *A* rewriting logic specification *(also called a* rewrite theory*) is a tuple* $\mathcal{R} = (\Omega, E, L, R)$ *where* Ω *is an algebraic signature (which in the order-sorted case would have the form* $\Omega = (S, \leq, \Sigma)$*), E is a set of equations and membership axioms, L is a set of labels, and R is a set of unconditional labeled rewrite rules written*

$$l : t \longrightarrow t'$$

and conditional labeled rewrite rules of the form

$$l : t \longrightarrow t' \textbf{ if } cond$$

where $l \in L$*,* t *and* t' *are terms in* $\mathcal{T}_\Omega(X)$*, and* cond *is a conjunction of* rewrite conditions *of the form* $u \longrightarrow u'$*,* equational conditions *of the form* $v = v'$ *and* membership conditions $w : s$*, for* u, u', v, v', w *terms in* $\mathcal{T}_\Omega(X)$ *and* s *a sort in* Ω*.*

5.3.1 Rewriting Logic Specifications in Maude

A rewriting logic specification is represented in Maude as a *system module*. A system module is declared as a functional module (described in Part I) with the exception that the keywords `fmod` and `endfm` are replaced by `mod` and `endm`.[3] Rewrite rules are written

```
rl [l] : t => t' .
```

and

```
crl [l] : t => t' if cond .
```

in the conditional case. A conjunct in the condition *cond* may be a term of sort `Bool`, an equality, a membership test, or a *rewrite condition* which is written with syntax u `=>` u'.

5.4 Nondeterminism

Nondeterminism means that a system may exhibit different behaviors—or give different results—from the same start state. Most dynamic systems are nondeterministic, due to race conditions and other (external) factors.

Example 60 Another example (borrowed from [69]) is a nondeterministic choice operator which returns nondeterministically one of its arguments:

```
mod CHOICE-INT is
  including INT .
  op _?_ : Int Int -> Int .
  vars I J : Int .
  rl [choose_first] : I ? J => I .
  rl [choose_second] : I ? J => J .
endm
```

[3] This is to indicate that we are no longer in a *functional* world.

A term `3 ? 5` could change into both `3` and `5`. ♠

Nondeterministic behavior is reflected in the fact that the set of rewrite rules is not confluent.

5.5 Nontermination

Reactive systems such as operating systems, nuclear power plant controllers, airport flight controller systems, etc., are—or should be!—nonterminating. Therefore, while an equational specification, including the "equational specification component" of a rewriting logic specification, is required to be both confluent and terminating in Maude, the set of rewrite rules may well be both nonconfluent and nonterminating.

Remark: We have not yet presented the notion of "applying a rewrite rule" formally. Informally, it can be seen as using the rule "from left to right" modulo the underlying equational theory.

The reader could at this point do worse than to browse through Chapter 6 to get an idea of the different ways in which Maude can execute a rewriting logic specification.

5.6 Examples

This section presents some specifications which will be executed in Chapter 6.

5.6.1 Simulating a Football Game

The following rewriting logic specification is supposed to "simulate" an ("American") football game. For the European reader it may also serve the purpose of a *specification* explaining *how* the score of a game changes as the result of various actions such as a "touchdown" or a "safety." Note that a football game exhibits a highly nondeterministic behavior, as illustrated by the surprising win of the New York Giants over longtime cheaters New England Patriots in Super Bowl 2008.

A *state* of a game is a term of the form

```
"49ers" vs "Cowboys" 49 : 0
```

where the first string (`"49ers"`) denotes the home team, the second string (`"Cowboys"`) the away team, and the rest is the current score. The Maude specification of the possible "behaviors" of a football game is given by the following module:

```
mod ONE-FOOTBALL-GAME is
  protecting NAT .
  protecting STRING .
  sort Game .
  op _vs_ _:_ : String String Nat Nat -> Game [ctor] .
```

```
  vars HOME AWAY : String .     *** Name of home and away team
  vars M N : Nat .

  *** The following rules model the home team scoring:

  rl [touchdown-home] : HOME vs AWAY M : N  =>  HOME vs AWAY (M + 6) : N .
  rl [field-goal-home] : HOME vs AWAY M : N  =>  HOME vs AWAY (M + 3) : N .
  rl [extra-point-kick-home] :
     HOME vs AWAY M : N  =>  HOME vs AWAY (M + 1) : N .
  rl [two-point-conversion-home] :
     HOME vs AWAY M : N  =>  HOME vs AWAY (M + 2) : N .
  *** Away team tackled in their own end zone:
  rl [safety-home] :  HOME vs AWAY M : N  =>  HOME vs AWAY (M + 2) : N .

  *** The following rules model the scoring possibilities for the away team:

  rl [touchdown-away] : HOME vs AWAY M : N  =>  HOME vs AWAY M : (N + 6) .
  rl [field-goal-away] : HOME vs AWAY M : N  =>  HOME vs AWAY M : (N + 3) .
  rl [extra-point-kick-away] :
     HOME vs AWAY M : N  =>  HOME vs AWAY M : (N + 1) .
  rl [two-point-conversion-away] :
     HOME vs AWAY M : N  =>  HOME vs AWAY M : (N + 2) .
  rl [safety-away] :  HOME vs AWAY M : N  =>  HOME vs AWAY M : (N + 2) .
endm
```

Exercise 114 *Show a sequence of rule applications which leads from the horrible state*

```
  "49ers" vs "Giants" 14 : 38
```

to the wonderful state

```
  "49ers" vs "Giants" 39 : 38.
```

5.6.2 Modeling the Life of a Person

The following example models the possible lives of a person. A state in the life of a person is represented by a term

```
  person(name, age, status)
```

where *status* could be either `single`, `engaged`, `married`, `separated`, `divorced`, `widow`, `widower`, or `deceased`, and where *age* denotes the age of the person. A typical state could be

```
  person("Metuzalem", 998, married).
```

My Maude specification looks like

```
mod ONE-PERSON is
  protecting NAT .
  protecting STRING .
  sorts Person Status .
  op person : String Nat Status -> Person [ctor] .
  ops single engaged married separated divorced deceased
      widow widower : -> Status [ctor] .

  var X : String .  var N : Nat .  var S : Status .

  crl [birthday] :
      person(X, N, S) => person(X, N + 1, S) if N <= 1000 /\ S =/= deceased .

  crl [successful-proposal] :
      person(X, N, S) => person(X, N, engaged)
      if N >= 15 /\ (S == single or S == divorced) .

  rl [marriage] : person(X, N, engaged) => person(X, N, married) .
  ...
endm
```

Exercise 115 *Complete the above module as you find appropriate with rules for e.g. broken engagement, separation, divorce, death, death of spouse, and maybe other possible happenings.*

5.6.3 A Coffee Bean Game

The *coffee bean game* (which I found in [55]) is a one-person game in which one is given a *sequence* (or *list*) of coffee beans, where a coffee bean may be either white or black. The rules of the game are simple: Two black beans next to each other may be replaced by one white bean, while a white bean which lies next to a black bean may be removed. The goal of the game is to be left with the least amount of beans possible.

Exercise 116

1. *Specify the coffee bean game in Maude.*
2. *Explain why you used* rewrite rules *instead of* equations *to describe this game.*
3. *Explain why the game is terminating.*
4. *Show that the coffee bean game is not confluent by using the techniques of Section 3.4. (If the game were confluent it would be a very boring game with only* one *possible outcome.)*
5. *Show that from a starting state*

   ○ ○ ● ● ○ ● ○ ●

   one may reach a final state with one white bean, and another final state with five white beans.

6. *Can you use the techniques from Section 3.4 to make the specification confluent by adding* one *rule to the specification. Prove that the resulting specification is confluent and therefore boring.*

7. *In the resulting boring specification, can you determine the final state of every game (as a function of the start state)?*

5.7 Concurrency

In a (distributed) system with more than one processor, many different actions may take place in the system *concurrently*, i.e., at the same time.

Rewriting logic is a logic of change in which the statements of the logic say that

> "state t *may* evolve to a state t'."

In addition, rewriting logic is a logic for *reasoning* about possible *concurrent* computation steps. That is, in rewriting logic one can reason about properties of the form

> "the system in state t may perform actions concurrently in *one concurrent step* to reach a state t'."

The way of thinking about "possible concurrent computation steps" is: assume that we have as many processors as we want and a way of delegating jobs to different processors. What actions could under this scenario be performed concurrently (in one step)? Note that although we use rewriting logic to *reason* at a high level about *possible* concurrent steps, the *execution* in Maude is done *sequentially*.

In the rest of this section we reason about what actions could naturally be performed concurrently.

5.7.1 Concurrent steps

Assume that from a state t_1 a system may evolve in *one* step to a state u_1. (If it helps your intuition, you could well assume that each action takes, say, 10 minutes to perform.) Assume furthermore that a state t_2 could evolve to a state u_2 in one step (which may also take 10 minutes). It then seems reasonable that a state $f(t_1, t_2)$ could evolve to the state $f(u_1, u_2)$ in one *concurrent* step, in which the steps $t_1 \longrightarrow u_1$ and $t_2 \longrightarrow u_2$ have been computed in parallel. (I again emphasize that this is abstract reasoning about *possible* concurrent computations. A concrete implementation on a distributed architecture would have to take care of the task of distributing the two computation tasks to two processors, of synchronizing the results, and so on.)

Example 61 The computation of an expression

```
squareroot(9762385199087) + findPrime(13852379)
```

could obviously be distributed so that one processor could spend, say, 15 minutes on computing `squareroot(9762385199087)`, and another processor could be assigned to compute `findPrime(13852379)` in the same time. That is, if `squareroot(9762385199087)` $\longrightarrow m$ and `findPrime(13852379)` $\longrightarrow n$, for some numbers m and n, can be computed in one step each, then

```
squareroot(9762385199087) + findPrime(13852379)
```

could evolve to a state m `+` n in one concurrent step. ♠

This observation/idea can be generalized. Assume that t_1 can be computed to u_1 in one step, that t_2 can be computed to u_2 in one step, ..., and that t_n can be computed to u_n in one step. Then, there should be a possibility to have one concurrent step taking a state $f(t_1, \ldots, t_n)$ to the state $f(u_1, \ldots, u_n)$.

We can think of this as getting a term $f(t_1, \ldots, t_n)$, and then having the *possibility* of letting one processor "compute" t_1, another processor t_2, etc., and that they then report back their respective values $u_1, \ldots, u_n$. The processor getting the task of dealing with t_i may itself use other processors to compute subparts of t_i concurrently. That is, the step $t_i \longrightarrow u_i$ may itself be a concurrent step.

Example 62 Consider the following specification:

```
mod CONC-1 is
  sort s .
  ops a a' b b' c c' d d' e e' f f' : -> s [ctor] .
  op g : s s -> s [ctor] .
  op h : s s s -> s [ctor] .
  rl [l1] : a => a' .
  rl [l2] : b => b' .
  rl [l3] : c => c' .
  rl [l4] : d => d' .
  rl [l5] : e => e' .
  rl [l6] : f => f' .
endm
```

In this specification there are six atomic steps (or "computations") `a` $\longrightarrow$ `a'`, `b` $\longrightarrow$ `b'`, ..., and `f` $\longrightarrow$ `f'`. A concurrent step takes `g(a,b)` to `g(a',b')` (just let one processor compute `a` $\longrightarrow$ `a'` and another processor compute `b` $\longrightarrow$ `b'`). Similarly there is a concurrent step `g(c,d)` to `g(c',d')` and another step `g(e,f)` to `g(e',f')`.

Furthermore, there is a concurrent step

`h(g(a,b),g(c,d),g(e,f))` $\longrightarrow$ `h(g(a',b'),g(c',d'),g(e',f'))`

in which six actions are performed concurrently. ♠

Exercise 117 *In the above specification, explain what concurrent steps are possible from the states* `g(g(a,a),g(b,c))` *and* `h(a,b',g(c,d))`.

A Population

Our previous specification ONE-PERSON was fairly boring in that the state could only contain one person. In this example we instead consider a *population*, which may be modeled as a multiset of persons:

```
mod POPULATION is
  protecting STRING .
  protecting NAT .
  sorts Person Population Status .
  subsort Person < Population .
  op empty : -> Population [ctor] .
  op __ : Population Population -> Population [ctor assoc comm id: empty] .
  op person : String Nat Status -> Person [ctor] .
  ops single divorced : -> Status [ctor] .
  ops engaged separated married : String -> Status [ctor] .

  vars X X' : String .   vars M N : Nat .
  vars S S' : Status .

  crl [birthday] : person(X, N, S) => person(X, N + 1, S)  if N <= 1000 .

  crl [engagement] :
      person(X, N, S) person(X', M, S') =>
      person(X, N, engaged(X')) person(X', M, engaged(X))
         if (S == single or S == divorced) /\ N >= 16
            /\ (S' == single or S' == divorced) /\ M >= 16 .

  rl [wedding] :
     person(X, N, engaged(X')) person(X', M, engaged(X))  =>
     person(X, N, married(X')) person(X', M, married(X)) .
  ...
endm
```

A population could be for example

```
person("Claudius", 60, married("Gertrude"))
person("Gertrude", 50, married("Claudius"))
person("Hamlet", 28, single)
person("Ophelia", 19, single)
person("Old Norway", 67, married("don_t_know"))
person("Fortinbras", 40, single)
person("Laertes", 57, married("??"))
```

Exercise 118 *Complete the above specification by giving appropriate rules for* separation, divorce, *and* death. *(Don't worry about marriage being something between a male and a*

female, as that is a constraint of the past.) As there is no status `deceased`*, death should result in removal from the population. For example, the state of the above system after the death of* `"Old Norway"` *should be*

```
person("Claudius", 60, married("Gertrude"))
person("Gertrude", 50, married("Claudius"))
person("Hamlet", 28, single)
person("Ophelia", 19, single)
person("Fortinbras", 40, single)
person("Laertes", 57, married("??"))
```

Note that there is a concurrent step from

```
person("Hamlet", 28, single)  person("Ophelia", 19, single)
```

to a state

```
person("Hamlet", 29, single)  person("Ophelia", 20, single)
```

in which *two* `birthday` steps have been performed at the same time. (The above state has the "form" $f(a, b)$, where $a \longrightarrow a'$ and $b \longrightarrow b'$ can be seen as the two birthday steps and f as the multiset union operator `__` .)

From a state

```
person("Hamlet", 28, single)  person("Ophelia", 19, single)
person("Rosenkrantz", 38, single)  person("Helena", 30, single)
```

it should be possible to arrange two engagements concurrently, e.g., to the state

```
person("Hamlet", 28, engaged("Rosenkrantz"))
person("Ophelia", 19, engaged("Helena"))
person("Rosenkrantz", 38, engaged("Hamlet"))
person("Helena", 30, engaged("Ophelia")).
```

However, it should not be possible for one person (say, `"Ophelia"`) to concurrently be involved in two engagements (which reception should she attend?). It is of course also possible to go from a state

```
person("Hamlet", 28, single)  person("Ophelia", 19, single)
person("Rosenkrantz", 38, single)  person("Helena", 30, single)
```

to a state

```
person("Hamlet", 28, engaged("Ophelia"))
person("Ophelia", 19, engaged("Hamlet"))
person("Rosenkrantz", 38, single)  person("Helena", 30, single).
```

That is, not everyone *has to* engage in festivities in one step.

Exercise 119

1. *Is it possible to reach a state in which* `"Ophelia"` *is older than* `"Hamlet"` *starting from either of the states given above?*
2. *What is the largest number of "actions" that can be performed concurrently starting from the state*

```
person("Claudius", 60, married("Gertrude"))
person("Gertrude", 50, married("Claudius"))
person("Hamlet", 28, single)
person("Ophelia", 19, single)
person("Old Norway" 67, married("don_t_know"))
person("Fortinbras", 40, single)
person("Laertes", 57, married("??"))
```

3. *What is the largest number of concurrent actions possible in one step from the above state if we don't count birthdays and deaths?*

□ The specification `POPULATION` does not take the passage of time seriously. We have seen that it is indeed possible for `"Ophelia"` to become older than both `"Hamlet"` and even `"Gertrude"`. A system where events may have quantitative *durations* and where the duration of events influences the functionality of the system is called a *real-time system*.

For example, a text editor responds in the same way no matter how long time it takes you to press a key, while in a cell-phone system you should pay less if you talk for one minute than if you hang up after ten minutes.

The references [79, 80] deal with the specification, execution, and analysis of real-time system in rewriting logic and (Real-Time) Maude. We will not treat real-time systems in this course. □

Many Football Games

As all of us who have been in a busy Las Vegas casino on a Sunday morning know, you can watch (and wager on!) not only *one* football game, but quite a lot of them at the same time. Furthermore, field goals—even touchdowns—may be scored concurrently in different games. We can model a non-empty multiset of games as follows:

```
mod NFL-ROUND is
  protecting ONE-FOOTBALL-GAME .
  sort Games .
  subsort Game < Games .
  op _;_ : Games Games -> Games [ctor assoc comm] .
endm
```

Exercise 120

1. *Give a state (i.e., a ground term) of the specification with at least three games.*

2. *Show one possible result, starting from the state you gave above, of applying one concurrent rewrite step in which points are scored in all three games.*

5.7.2 Rule Applications Inside Rule Applications

Let

$$l : f(x) \longrightarrow g(x)$$

be a rewrite rule. Then, an action takes "f" to "g" no matter what is x. Therefore, one could think that it is possible to let a processor "work on" x while another processor takes f to g. For example, if $l' : a \longrightarrow b$ is another rewrite rule, then a term $f(a)$ should be able to rewrite to $g(b)$ in one concurrent step. That is, we can think of this as a processor seeing $f(\ldots)$, and knows that it can take f to g in one step. It can "delegate" to another processor the task of working on the "interior," which is done concurrently.

Example 63 An airplane may fly from Chicago to Oslo in one (long) rewrite step. A person may write Chapter 6 of these lecture notes in another long rewrite step. Then, a person sitting in an airplane flying from Chicago to Oslo may write Chapter 6 of this compendium in one rewrite step. (This happened in 2003!) ♠

Exercise 121 *How many actions (rule applications) do you think can be performed in one step from a state $f(f(f(a)))$ in the specification $\{l_1 : f(x) \longrightarrow g(x),\ l_2 : a \longrightarrow b\}$?*

Example 64 The following specification reverses a list of natural numbers using rules instead of equations:

```
mod LIST-REV is
  sorts Nat List .
  subsort Nat < List .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat [ctor] .
  op nil : -> List [ctor] .
  op __ : List List -> List [ctor assoc id: nil] .
  op reverse : List -> List .

  vars M N : Nat .   var L : List .

  rl [rev_nil] :  reverse(nil) => nil .
  rl [rev_list] : reverse(N L) => reverse(L) N .
  rl [add_0] :    0 + N => N .
  rl [add_s] :    s(M) + N => s(M + N) .
endm
```

Here, a list `(0 + s(0))  s(s(0))` could be reversed by putting `0 + s(0)` at the end, but it seems reasonable to also allow rewrite `0 + s(0)` to `s(0)` *while* putting it at the end of the list. That is, the computation `(0 + s(0))  s(s(0))` $\longrightarrow$ `s(s(0))  s(0)` could be performed in one concurrent step. ♠

5.8 Deduction in Rewriting Logic

This section defines formally the rewrite relation and the notion of concurrent rewrite steps, which have only been discussed and motivated informally so far in this chapter. For simplicity of exposition, we consider *one-sorted specifications* without *conditional* rewrite rules.

Given a rewriting logic specification $\mathcal{R} = (\Omega, E, L, R)$ the *sequents* ("logical formulas") of rewriting logic have the form

$$\mathcal{R} \vdash t \longrightarrow u$$

for t, u terms in $\mathcal{T}_\Omega(X)$. This sequent intuitively means that it is *possible* to reach the state u from the state t using the rules in $\mathcal{R}$ (zero or more times). Formally, $\mathcal{R} \vdash t \longrightarrow u$ holds if and only if $t \longrightarrow u$ can be obtained by a finite number of applications of the deduction rules of rewriting logic.

Notation. We sometimes write $t(x_1, \ldots, x_n)$ for a term t to emphasize that all the variables in t are in the list $x_1, \ldots, x_n$. We write $t(u_1/x_1, \ldots, u_n/x_n)$ for the term t where each occurrence of x_i has been replaced by the term u_i. For example, if $t(x, y)$ denotes the term $f(g(x), h(a, y))$, then $t(g(y)/x, a/y)$ denotes the term $f(g(g(y)), h(a, a))$.

Definition 31 (Deduction rules of rewriting logic) *Given a rewriting logic specification $\mathcal{R} = (\Omega, E, L, R)$ (which we for simplicity assume is one-sorted and has only unconditional rules), the sequent*

$$\mathcal{R} \vdash t \longrightarrow u$$

holds if and only if $t \longrightarrow u$ can be obtained by finite application of the following rules of deduction:

Reflexivity: *For each term t in $\mathcal{T}_\Omega(X)$*

$$t \longrightarrow t$$

holds.

Equality: *If $t \longrightarrow t'$ holds, and $E \vdash t = u$ and $E \vdash t' = u'$ both hold, then*

$$u \longrightarrow u'$$

holds.

Congruence: *For each function symbol f in Ω, if $t_1 \longrightarrow u_1$, ..., and $t_n \longrightarrow u_n$ all hold, then*

$$f(t_1, \ldots, t_n) \longrightarrow f(u_1, \ldots, u_n)$$

holds.

Replacement: *For each rewrite rule* $l : t(x_1, \ldots, x_n) \longrightarrow u(x_1, \ldots, x_n)$ *in* R*, if* $t_1 \longrightarrow u_1$*, . . . , and* $t_n \longrightarrow u_n$ *all hold, then*

$$t(t_1/x_1, \ldots, t_n/x_n) \longrightarrow u(u_1/x_1, \ldots, u_n/x_n)$$

holds.

Transitivity: *If* $t_1 \longrightarrow t_2$ *and* $t_2 \longrightarrow t_3$ *both hold, then*

$$t_1 \longrightarrow t_3$$

holds.

These deduction rules look very similar to the deduction rules of *equational logic* (with **Replacement** corresponding to **Substitutivity**). Indeed, only the **Symmetry** property of equational logic is missing.

The rewrite relation $\longrightarrow$ corresponds to applying rewrite rules from left to right zero or more times, and to equational reduction in the following sense:

Proposition 5 *Given an equational specification* (Ω, E)*, we have that*

$$t \overset{*}{\rightsquigarrow}_E u \text{ if and only if } (\Omega, \emptyset, \{l\}, rules(E)) \vdash t \longrightarrow u$$

(where $rules(E)$ *changes each equation into a rule by just changing* $=$ *to* $\longrightarrow$ *and adding a label* l*).*

The following fact follows trivially Theorem 10 and Proposition 5:

Corollary 1 *It is in general undecidable whether a given term* t *rewrites to a given term* u *in a given rewriting logic specification* $\mathcal{R}$*.*

5.8.1 Concurrent Steps

This section formalizes the notion of (possible) concurrent rewrite steps. We see that the **Congruence** rule corresponds to the kind of concurrency mentioned in Section 5.7.1. If $a \longrightarrow b$ and $c \longrightarrow d$ hold and can be performed in "one step", then $f(a, c) \longrightarrow f(b, d)$ also holds and can be performed in one step. The **Replacement** rule models the other kind of concurrency, where an "outer" step applies a rule, and an "inner" step performs actions on the variables of the rule. If $l : f(x, y) \longrightarrow g(x, y)$ is a rule, and $a \longrightarrow b$ and $c \longrightarrow d$ hold and can be performed in "one step", then it is reasonable to assume that $f(a, c) \longrightarrow g(b, d)$ can be performed in one step. These observations motivate the formal definition of concurrent steps:

Definition 32 (Concurrent steps)

- *A sequent* $t \longrightarrow u$ *is called a* one-step concurrent rewrite *if it can be obtained using only the rules* ***Reflexivity***, ***Equality***, ***Congruence***, *and* ***Replacement*** *of rewriting logic. (That is, the* ***Transitivity*** *rule cannot be used in a one-step concurrent rewrite.)*

- *A one-step concurrent rewrite is called a (one-step)* sequential rewrite *if the* ***Replacement*** *rule (which is where a rule is "applied") is used* once *in the deduction.*

In a one-step *sequential* rewrite, some rule is applied once, which means that a one-step sequential rewrite corresponds to one-step equational reduction:

$t \rightsquigarrow_E u$ if and only if $(\ldots, \emptyset, \{l\}, rules(E)) \vdash t \longrightarrow u$ is a one-step sequential rewrite.

Example 65 Given the rewriting logic specification

$$\{\, l_1 : f(x) \longrightarrow g(x),\ l_2 :\ a \longrightarrow b\}$$

(we follow the notational conventions for one-sorted equational specifications when writing one-sorted rewriting logic specifications), there is a one-step concurrent rewrite

$$f(f(f(a))) \longrightarrow g(g(g(b))).$$

Proof: Using the **Replacement** rule on rewrite rule l_2 gives that $a \longrightarrow b$ is deducible. By the **Replacement** rule (now using l_1 and $a \longrightarrow b$ for $t_1 \longrightarrow u_1$) we have that $f(a) \longrightarrow g(b)$ is deducible. Using the **Replacement** rule again on l_1, but now with $f(a) \longrightarrow g(b)$ for $t_1 \longrightarrow u_1$, gives $f(f(a)) \longrightarrow g(g(b))$. Another application of the **Replacement** rule gives the sequent

$$f(f(f(a))) \longrightarrow g(g(g(b))).$$

This is a one-step concurrent rewrite since the **Transitivity** rule was not used in its deduction. ♠

Exercise 122 *Show a deduction of $f(f(f(a))) \longrightarrow g(g(g(b)))$ in the specification of Example 65 which involves only* sequential *rewrite steps (and the use of the* ***Transitivity*** *rule).*

Example 66 In the module `CONC-1` in Example 62 there is a one-step concurrent rewrite

$$h(g(a,b), g(c,d), g(e,f)) \longrightarrow h(g(a',b'), g(c',d'), g(e',f'))$$

because $a \longrightarrow a'$ holds by the **Replacement** rule, since $l :\ a \longrightarrow a'$ is a rewrite rule. The sequent $b \longrightarrow b'$ holds for the same reason. The **Congruence** rule then gives that $g(a,b) \longrightarrow g(a',b')$ holds. In the same way, one can deduce $g(c,d) \longrightarrow g(c',d')$ and $g(e,f) \longrightarrow g(e',f')$. Applying again the **Congruence** rule on these last three sequents gives the desired

$$h(g(a,b), g(c,d), g(e,f)) \longrightarrow h(g(a',b'), g(c',d'), g(e',f')).$$

One can also deduce a sequent

$$h(g(a,b), g(c,d), g(e,f)) \longrightarrow h(g(a',b), g(c,d), g(e,f))$$

because $b \longrightarrow b$, $c \longrightarrow c$, etc. hold because of the **Reflexivity** rule. ♠

The examples above indicate that a concurrent step may be decomposed into a number of sequential steps:

Proposition 6 (Sequentializability [69]) *For each concurrent rewrite $t \longrightarrow t'$, either $t = t'$ or there is an $n \in \mathcal{N}$ and a chain of one-step (concurrent) rewrites*

$$t \longrightarrow t_1 \longrightarrow \cdots \longrightarrow t_n \longrightarrow t'.$$

In addition, we can choose all the steps to be sequential *rewrite steps.*

Therefore, each concurrent rewrite step can be simulated by a number of sequential steps.

We will call a (finite or infinite) sequence of one-step rewrites a *behavior* or *run* of a system.

Exercise 123 *We recall the coffee bean game described in Section 5.6.3 and your Maude specification of it which solved Exercise 116.*

1. *Prove formally that the state (representing) ∘ • • ∘ rewrites in one sequential step to the state (representing) ∘ ∘ ∘.*
2. *What is the highest number of "rule applications" that can be performed in one concurrent step from the state (representing) ∘ ∘ • • ∘ • ∘ •. What is the resulting state? Does it make sense? That is, do you think that it is natural to delegate the job to different processors in this way (you can again think that every step takes 10 minutes to perform!)?*

5.8.2 Termination and Confluence

The notions of *termination* and *confluence* carry over to rewrite systems as expected: a rewriting logic specification is terminating if (the underlying equational specification is terminating and) there is no infinite sequence of one-step rewrites. Likewise, a rewriting logic specification is confluent if and only if $t \longrightarrow t_1$ and $t \longrightarrow t_2$ imply that there is a term u such that both $t_1 \longrightarrow u$ and $t_2 \longrightarrow u$ hold.

5.9 Example: A Sorting Program

Consider the following specification of a "swap" operation on a list of integers.

```
mod SORT is
  protecting INT .

  sort List .
  subsort Int < List .
  op nil : -> List [ctor] .
  op __ : List List -> List [assoc id: nil ctor] .

  vars I J : Int .
  var L : List .

  crl [swap] : I L J  =>  J L I if J < I .
endm
```

Repeated use of the `swap` rule (for example by using Maude's `rew` command explained in Chapter 6) will swap integers until the list is sorted:

```
Maude> rew 8 4 0 -3 76 54 21 0 -9 3 23 .

result List: -9 -3 0 0 3 4 8 21 23 54 76
```

We can formally prove the sequent $\texttt{SORT} \vdash \texttt{0 3 2 1} \longrightarrow \texttt{0 1 2 3}$ in rewriting logic as follows:

1. $\texttt{0} \longrightarrow \texttt{0}$, $\texttt{3} \longrightarrow \texttt{3}$, $\texttt{2} \longrightarrow \texttt{2}$, and $\texttt{1} \longrightarrow \texttt{1}$ are all deducible in rewriting logic because of **Reflexivity**.
2. The **Replacement** deduction rule used on the rule `swap` with $\texttt{3} \longrightarrow \texttt{3}$ "for" `I`, $\texttt{2} \longrightarrow \texttt{2}$ for `L`, and $\texttt{1} \longrightarrow \texttt{1}$ for `J` gives that $\texttt{3 2 1} \longrightarrow \texttt{1 2 3}$ is deducible.
3. **Congruence** (w.r.t. the function symbol `__`) together with the proven assumptions $\texttt{0} \longrightarrow \texttt{0}$ and $\texttt{3 2 1} \longrightarrow \texttt{1 2 3}$ gives that $\texttt{0 (3 2 1)} \longrightarrow \texttt{0 (1 2 3)}$ is deducible, which due to the `assoc` attribute of `__` can be written $\texttt{0 3 2 1} \longrightarrow \texttt{0 1 2 3}$.

Exercise 124

1. *Prove formally using the deduction rules of rewriting logic that the sequent*

   $$\texttt{SORT} \vdash \texttt{4 8 5 0 1} \longrightarrow \texttt{0 1 4 5 8}$$

   holds.

2. *It may not be obvious that the specification* `SORT` *is terminating. Can you give a convincing argument for its termination?*

3. *Is it possible to sort the list*

   ```
   8 4 0 -3 76 54 21
   ```

   in one *concurrent step? How about the list* `1 3 2 0`*?*

4. *An even shorter "sorting program" would replace the* `swap` *rule by the following rule:*

   ```
   crl [swap] : I J  =>  J I  if J < I .
   ```

   (a) *Explain why the repeated application of the* `swap` *rule until it can no longer be used will result in a sorted list.*

   (b) *What is the minimum number of concurrent rewrite steps required to sort the lists*

   `8 4 0 -3 76 54 21` *and* `1 3 2 0`

   in the modified specification?

   (c) *Is there any list which can be sorted by fewer steps in the modified specification than in the original one?*

5. *What would be the undesired consequence of adding a function such as e.g.* `op sorted : List -> Bool .` *to the module* `SORT`*? (This function is supposed to be defined equationally so that* `sorted(`l`)` *is* `true` *if* l *is a sorted list, and is* `false` *otherwise.)* (Hint: *think about the* ***Congruence*** *rule of rewriting logic. See also Section 5.10.)*

5.10 * Frozen Operators

□ Consider extending the module `SORT` in Section 5.9 with an operator[4]

```
op first : List ~> Int .
```

which returns the first element in a list. This is a function and should be defined equationally as we have done earlier in the course. In the (extended) module `SORT` there is a rewrite

$$5\ 2 \longrightarrow 2\ 5.$$

By the **Congruence** rule of rewriting logic we can deduce that

$$\texttt{first(5 2)} \longrightarrow \texttt{first(2 5)},$$

which by the **Equality** rule of rewriting logic gives

$$5 \longrightarrow 2.$$

This latter rewrite is probably nothing we want. To avoid such undesired rewrites caused by "projecting" a dynamic domain onto a "static" domain, one can declare operators to be *frozen*:

```
op first : List ~> Int [frozen] .
```

Semantically, this frozenness means that a rewrite $t \longrightarrow t'$ does not allow us to deduce $\texttt{first}(t) \longrightarrow \texttt{first}(t')$. (This also applies if $\texttt{first}(t)$ is a subterm of a larger term.) In a further refinement one may specify that only *some* of the argument places (e.g., the second argument) of an operator is frozen. □

5.11 * Denotational Semantics

Not written.

5.12 Concluding Remarks

Rewriting logic is a logic of change where data types are specified by equational specifications and where the dynamic parts of a system are specified by labeled rewrite rules.

Among the main ideas (or "observations") upon which rewriting logic is based are:

- Rewrite rules can naturally model change, not just equality, in a system.
- Rewriting is intrinsically concurrent. No new constructs are necessary to specify and reason about concurrency.

Attractive features of rewriting logic compared to other models of concurrency include:

[4] Remember from page 63 that `~>` denotes a partial function, which is appropriate here since `first(nil)` may not be an integer.

- Static and dynamic properties are defined using more or less the same language. This could be compared to specification formalisms such as LOTOS [50], colored Petri nets, etc., where the data types may be specified by equational specifications and the dynamic behavior in a totally different formalism such as process algebra or place/transition net graphs.
- The formalism and its semantics are fairly intuitive and easy to understand also for non-formal-methods-experts. Again, we could compare to process algebras and even to very restrictive automaton models.
- We will later see that rewriting logic—again in contrast to many models of concurrency—has a fairly natural model of concurrent objects.

Chapter 6

Executing Rewriting Logic Specifications in Maude

The main reason—apart from obtaining a precise description of a complex system—for developing a *formal* (i.e., "mathematical") model of a system is to be able to *analyze* the system through its model. This chapter shows how a rewriting logic model can be analyzed by executing it in Maude.

Maude executes an *equational specification* without worrying about *which* of the equations to apply at a certain time, and without worrying about *where* to apply an equation. This is sufficient since an equational specification is assumed to be both confluent and terminating, ensuring that each expression has exactly one "result." A *rewriting logic* (or just *rewrite*) specification defines a model of *all* possible behaviors of a dynamic system. Since a dynamic system may exhibit different and possibly infinite behaviors, a rewrite specification may be both nonconfluent and nonterminating, and the notion of "result" may not make much sense.

Maude provides the following ways of executing a specification:

1. The simplest version is to execute (or "simulate") *one* out of the many possible behaviors from a given initial state. The Maude commands `rew` (or `rewrite`) and `frew` (or `frewrite` for "fair rewrite") execute *one* (pseudo-arbitrarily chosen) behavior of the system by "arbitrarily" applying rewrite rules to the initial state, pretty much like executing an equational specification. Since this process may not terminate if the specification is nonterminating, the user may provide an upper bound on the number of rewrite steps to perform. Example: what is the result of simulating at most 50 steps starting with the *term* `person("Peter", 38, single)` in our example specification `ONE-PERSON`?

2. Maude provides a `search` command for searching through *all possible* behaviors from a given initial state. This is done by asking Maude to search for terms which can be reached from the initial state and which satisfy a given condition. Is, for example, the term `person("Peter", 998, single)` reachable from `person("Peter", 38, single)`? How about `person("Peter", 20, single)`? A search may fail to terminate, such as when searching for the state `person("Peter", 20, single)` from the initial state `person("Peter", 38, single)` in a system *without* age limits.

3. Maude is equipped with a *temporal logic model checker* [32] for checking whether all behaviors from a certain initial state satisfy a given *temporal logic* property. Is it e.g. the case that a person in state `married` will be reached in *all* behaviors from state `person("Peter", 38, single)`? Is an engagement always followed by a marriage? The Maude model checker could also have saved Gilgamesh a lot of trouble by finding out whether a person will *always* eventually reach a `deceased` state.

4. Finally, a user may define her own execution strategies in Maude through the use of Maude's reflective and meta-programming capabilities, a taste of which is given in the follow-up course INF 5130.

This chapter describes executions of the first two kinds.

6.1 Executing One Sequential Rewrite Step

Any rewrite can, according to Proposition 6, be decomposed into a sequence of *one-step sequential* rewrites. The basis of a Maude execution is therefore performing a one-step sequential rewrite, i.e., applying a rewrite rule once.

It is fairly easy to see that applying a rewrite rule when there are no equations in the specification is the same as applying an equation in the "corresponding equational specification." The problem of executing a rewrite rule therefore boils down to dealing with the **Equality** deduction rule of rewriting logic. To check even whether a rule $l \longrightarrow r$ applies to the *root* (or *top*) position of a term t requires to check whether l E-matches t for E the equational part of the rewriting logic specification. Unfortunately, we have seen on page 118 that in general matching modulo an arbitrary equational theory E is not decidable. This means that it is impossible to know, given a term t, whether a certain rule can be used on t.

Example 67 Given a Maude specification

```
mod NON-COHERENT is
  sort s .
  ops a b d : -> s .
  ops c f : -> s [ctor] .
  eq a = b .
  eq b = c .
  eq d = f .
  rl [l] : b => d .
endm
```

The term `a` should rewrite to `d` in one step since `a` $\longrightarrow$ `d` (because `a` and `b` are equal according to the equations in the specification). However, Maude cannot immediately "see" that rule `l` should apply to `a`, but would have to check for all terms which are equal to `a`, i.e., `a`, `b`, and `c`, whether the rule can be applied. Furthermore, given state `c`, Maude would have to search "uphill" to check whether this state can be rewritten. ♠

While the search for E-equivalent forms in the above small example does not seem too disastrous, this search would become quite bad—indeed, unsolvable—in richer equational specifications. Maude therefore does the obvious thing: it assumes that the equational part E of a rewriting logic specification is confluent and terminating, and Maude *first* reduces a term t to its E-normal form $t!$ using the equations in the specification, and *then* checks whether a rewrite rule can be applied to $t!$. If so, the rewrite rule is applied, and the resulting term t' is normalized to $t'!$. Rewriting is still performed *modulo* structural axioms such as `assoc`, `comm`, and `id`.

To avoid "losing" rewrites when applying rewrite rules in this way, the left-hand side of each rewrite rule must be in "E-irreducible" form. That is, the left-hand side t of a rule should be such that any ground instance $t\sigma$ is E-reducible only if some term $x_i\sigma$ is E-reducible. We then say that t is *ground irreducible*, and say that a rewriting logic specification is *coherent* if the left-hand side of each rewrite rule is ground irreducible. Usually, this means that the *left-hand side of a rewrite rule must be a constructor term.*

Example 68 The above specification `NON-COHERENT` is *not* coherent since the left-hand side `b` is reducible using the equations. Therefore, given a state `a`, Maude will reduce it to `c`, and will then check if a rule applies, and finds that it does not. Therefore, Maude "misses" the rewrite `a` $\longrightarrow$ `d`.

The coherent equivalent version of `NON-COHERENT` is

```
mod COHERENT is
  sort s .
  ops a b d : -> s .
  ops c f : -> s [ctor] .
  eq a = b .
  eq b = c .
  eq d = f .
  rl [l] : c => d .
endm
```

The term `a` is reduced to `c` using the equations, and the rule `l` will rewrite `c` to `d`, which is further reduced to its normal form `f` using the equations. ♠

Exercise 125 *Which of the following rules could have been used in the specification* `NON-COHERENT` *in Example 67 instead of the given rule, so that we would get an "equivalent" and coherent specification? Explain!*

1. `rl [l] : a => f .`

2. `rl [l] : b => f .`

3. `rl [l] : c => f .`

4. `rl [l] : c => b .`

Example 69 Applying the rule `birthday` once on `person("Robert Fisk", 39 + 22, married)` is done by first reducing the state to its normal form `person("Robert Fisk", 61, married)`, then applying the rule, giving `person("Robert Fisk", 61 + 1, married)`, which is finally normalized to `person("Robert Fisk", 62, married)` using the (built-in) equations for `+`. ♠

Example 70 A rule

```
rl [gettingYounger] : person(X, N + 1, S) => person(X, N, S) .
```

would never be applied, since a term would first be normalized to a form such as e.g. `person("Gilgamesh", 50, married)`. The problem is that the left-hand side of the rule is not a constructor term since it contains the defined symbol `+`. A better rule would be

```
crl [gettingYounger] : person(X, N, S) => person(X, sd(N, 1), S)  if N > 0 .
```

or (since a number is just an abbreviation for a *constructor* term `s s ... 0`):

```
rl [gettingYounger] : person(X, s N, S) => person(X, N, S) .
```

♠

A rewrite rule—in contrast to an equation—may introduce a new variable in the right-hand side of the rule, such as in

```
rl [newAge] : person(X, N, S) => person(X, M, S) .
```

for `M` a variable of sort `Nat`. While such a rule may be convenient for specification purposes (the above rule says that a person can change her age to *any* natural number!), Maude cannot execute it because Maude has no clue what should be the new age of a person. (The rule could still be executed with an appropriate strategy, for example one which always assigns `15` to the new variable.)

□ A conjunct in the condition in a rule (or in an equation for that matter) may also have the form

$$x \texttt{ := } t$$

where x is a variable which does not appear in the left-hand side of the rule. Logically, this is just an equational condition with the same logical meaning as $x = t$. Operationally, it assigns the value t to the variable x. While it does not make much sense in our simple example, our birthday rule could have been written

```
var X : String .   vars M N : Nat .
crl [birthday] : person(X, N) => person(X, M)  if M := N + 1 .
```

Conjunctions of conditions are evaluated from left to right, and while one can have more than one initialization of new variables, in each conjunct of the condition, all the variables (except those being initialized in the conjunct) must have appeared in the left-hand side of the rule or must have been initialized in earlier conjuncts. A rule

```
crl [two-years-older] :
    person(X, N) => person(X, K)  if K := M + 1 /\ M := N + 1 .
```

can therefore *not* be executed (since `M` has no value when `K` should get its value), while the rule

```
crl [two-years-older] :
    person(X, N) => person(X, K)  if M := N + 1 /\ K := M + 1 .
```

is executable.

The left-hand side of an initialization conjunct of a condition does not need to be just a variable, but could have the form

$$t(x, y) \ := \ t'$$

for any *constructor* term t with introduced variables x and y. The conjunct holds if there are terms t_1 and t_2 such that $t(t_1/x, t_2/y)$ equals the normal form of t'; if that is the case, then x gets assigned the value t_1 and y gets assigned the value t_2.

Finally, the truth of a rewrite condition

```
... if ... /\ u => u' /\ ...
```

cannot be determined by just computing "normal forms." Maude must perform a breadth-first search which searches all computation paths from u to check if u' can be reached. If u' may never be reached from u, Maude can search for ever just to determine whether the rule can be applied! □

Conclusion: For a rewrite rule to be executable

- its left-hand side should be a constructor term
- its right-hand side should not introduce new variables (which are not initialized in the condition).

The equations E of a rewriting logic specification must be confluent and terminating. Maude does *not* check whether the equational part of your specification is confluent and terminating; Maude simply assumes that.

6.2 Executing Single Behaviors

Maude's `rew` and `frew` commands are used to execute *a single* behavior of a system. Either of these commands apply rewrite rules to perform one-step sequential rewrites until no rule can be applied, or until a user-given bound on the number of rewrites has been reached. The execution may not terminate if the user does not provide such an upper bound on the number of rewrites and the specification is nonterminating.

As explained above, each term is reduced to its E-normal form before a rewrite rule is applied, so a finite Maude execution with the `rew` or `frew` command will have the form

$$t \overset{*}{\rightsquigarrow} t! \longrightarrow t_1 \overset{*}{\rightsquigarrow} t_1! \longrightarrow t_2 \overset{*}{\rightsquigarrow} t_2! \longrightarrow \cdots \longrightarrow t_n \overset{*}{\rightsquigarrow} t_n!$$

and will return the term $t_n!$. Such as execution is often referred to as "simulating one behavior" of the system even though only a term is output. One could give the Maude command

```
set trace on .
```

before running the `rew` command to "simulate" a behavior by observing all the intermediate steps in the execution.

The syntax for the `rew` command is

```
rew t .
```

or

```
rew [n] t .
```

where n is the maximal number of rewrites to perform, and t is the term to rewrite. The `frew` command has similar syntax. In case the execution should take place in a module different from the "current" module, one could always specify in which module the rewrite should take place:

```
Maude> rew [100] in ONE-PERSON : person("Peter", 38, single) .
result Person: person("Peter", 136, married)
```

Since the specification is not (necessarily) confluent the choice of *which* rule to apply in each step, and *where* in the term to apply it, is important, as different choices give different results. Both the `rew` and the `frew` commands try to apply the rules in a "round Robin" format. However, the highest priority of `rew` is to apply rules as close to the "top" of the term as possible, and thereafter to apply the rules to the *leftmost* subterms. The `frew` command is more "fair" w.r.t. *where* in the term to apply the rules. To get a slightly more "random" execution one should probably use the `frew` command. (However, both `rew` and `frew` are deterministic in the sense that two `frew` executions of the same module and starting with the same initial term will give the same result.)

□ We will take a look at some examples to illustrate the choice of how the rules are applied when executing with `rew` and `frew`. In the following examples, we will have counters of the form `rule2(`n`)` which says that `rule2` was applied n times. First we notice that both `rew` and `frew` choose the rules in a "fair" way when all rewrites happen at the top:

```
mod TEST-REW1 is
  protecting NAT .
  sort Counter .
  ops rule1 rule2 rule3 : Nat -> Counter [ctor] .
  op f : Counter Counter Counter -> Counter [ctor] .
  vars N M K : Nat .
  rl [rule1] : f(rule1(N), rule2(M), rule3(K))
               => f(rule1(s N), rule2(M), rule3(K)) .
  rl [rule2] : f(rule1(N), rule2(M), rule3(K))
               => f(rule1(N), rule2(s M), rule3(K)) .
  rl [rule3] : f(rule1(N), rule2(M), rule3(K))
               => f(rule1(N), rule2(M), rule3(s K)) .
endm
```

```
Maude> rew [100] f(rule1(0), rule2(0), rule3(0)) .
result Counter: f(rule1(34), rule2(33), rule3(33))
Maude> frew [100] f(rule1(0), rule2(0), rule3(0)) .
result Counter: f(rule1(34), rule2(33), rule3(33))
```

In both cases, `rule1` was applied 34 times and the other rules 33 times each. The situation is not so good when the rewrites happen in a subterm since `rew` always applies rules in a leftmost-outermost way, while `frew` is fair also w.r.t giving each subterm a chance to rewrite:

```
mod TEST-REW2 is
  protecting NAT .
  sort Counter .
  ops rule1 rule2 rule3 : Nat -> Counter [ctor] .
  op f : Counter Counter Counter -> Counter [ctor] .
  var N : Nat .
  rl [rule1] : rule1(N) => rule1(s N) .
  rl [rule2] : rule2(N) => rule2(s N) .
  rl [rule3] : rule3(N) => rule3(s N) .
endm

Maude> rew [100] f(rule1(0), rule2(0), rule3(0)) .
result Counter: f(rule1(100), rule2(0), rule3(0))
Maude> rew [100] f(rule3(0), rule2(0), rule1(0)) .
result Counter: f(rule3(100), rule2(0), rule1(0))

Maude> frew [100] f(rule1(0), rule2(0), rule3(0)) .
result (sort not calculated): f(rule1(34), rule2(33), rule3(33))
Maude> frew [100] f(rule3(0), rule2(0), rule1(0)) .
result (sort not calculated): f(rule3(34), rule2(33), rule1(33))
Maude> frew [100] f(rule1(0), rule1(0), rule1(0)) .
result (sort not calculated): f(rule1(34), rule1(33), rule1(33))
```

Since `rew` always looks at the leftmost subterm of the term, it always rewrites what's applicable there, be it `rule1` or `rule3`, while `frew` tries to apply rules in all subterms. □

6.2.1 Executing Maude Specifications using `rew` and `frew`

In this section we use `rew` and `frew` to execute some of the specifications given in Chapter 5.

Exercise 126 *Declare an associative (*`assoc`*) and commutative (*`comm`*) choice operator* `_?_` *using only* one *rewrite rule so that e.g. the term* `1 ? 2 ? 3 ? 4` *could change to both* `1`*,* `2`*,* `3`*, and* `4`*. Use Maude's* `rew` *and* `frew` *commands to test which element is chosen from the terms* `1 ? 2 ? 3 ? 4` *and* `6 ? 2 ? 3`*.*

Exercise 127 *In this exercise we look at two ways of "choosing" a number in a given interval. That is, we want that* `choose`(m, n) *rewrites to an arbitrary integer* k *with* $m \leq k \leq n$*. In particular, you should ensure that* `choose`(m, n) *can rewrite (theoretically) to* any *integer in the interval.*

1. *Write a specification that introduces a variable in the right-hand side of a rule so that the "arbitrary" value is returned in one step.*

2. *Write a specification that does not introduce a new variable in the right-hand side of a rule, but where you may use more than one rewrite step to return the value.*

Example 71 The specification `ONE-FOOTBALL-GAME` on page 154 is not terminating (why?), but we can let Maude simulate a game with 15 scoring actions as follows:

```
Maude> rew [15] "49ers" vs "Cowboys" 0 : 0 .
result Game: "49ers" vs "Cowboys" 28 : 14
```

which gives a fairly correct result. ♠

Exercise 128 *The module* `ONE-FOOTBALL-GAME` *is a nonterminating specification where games are never stopped.*

1. *Add a rule which gives the possibility of "stopping" the game at any time and displaying the final score as a term of the form*

   ```
   "49ers" vs "Giants" FinalScore: 39 : 38
   ```

2. *Show that the resulting specification is still not terminating.*

3. *Modify your specification so that it becomes terminating by assuming that the total number of points scored in a game never exceeds 130.*

Exercise 129 *The module* `ONE-FOOTBALL-GAME` *is an incorrect model of possible scoring scenarios in a football game, since an application of the rule* `extra-point-kick-home` *may follow not only a* `touchdown-home` *but also a* `two-point-conversion-home` *or a* `field-goal-away`*. Modify your specification so that a* `two-point-conversion-home` *or an* `extra-point-kick-home` *may only follow an application of* `touchdown-home`*, and similarly for the away team.*

Exercise 130 *(If you dare.) Let Maude execute the specification* `ONE-PERSON` *from Exercise 115 to "predict" your and/or a loved one's future.*

Exercise 131 *Play the coffee bean game you specified in Exercise 116 a couple of times against Maude. That is, find a start state, and let Maude and you play the game from the same start state, and see who is left with the fewest beans. Also check who is the fastest :-)*

Exercise 132 *Another version (also taken from [55]) of the coffee bean game has the following rules:*

$$\begin{aligned}
\bullet\,\bullet &\longrightarrow \circ\,\circ\,\circ\,\circ\\
\bullet\,\circ &\longrightarrow \circ\,\circ\,\circ\,\bullet\\
\circ\,\bullet &\longrightarrow \bullet\\
\circ\,\circ &\longrightarrow \circ
\end{aligned}$$

1. *Specify this game in Maude and play it in Maude. Does it always terminate?*
2. *(Tricky?) Prove that the game is nonterminating or prove that it is terminating.*
3. *Is the game confluent? (Remember that a system may be nonterminating and still confluent!)*

6.3 Search in Maude

While using the `rew` and `frew` commands to execute one out of possibly many different behaviors can be very useful for a first prototyping of a specification, such executions may not be sufficient to deeply understand a specification. For example, no matter how many times we execute the module `ONE-FOOTBALL-GAME`, the home team will *never* lose. After many tests one could therefore be tempted to declare that "the away team can never win a football game," which is clearly wrong. (This author has himself once proudly declared—and written about it—that a communication protocol was correct after extensively testing it using Maude's `rew` command. He had to eat his words when his exhaustive investigation of *all* behaviors revealed flaws in the protocol.) The age of a person does not decrease in any of the `rew` executions of the `ONE-PERSON` specification. These tests by themselves do not exclude the possibility that the age of a person could decrease. By "manually" analyzing the trivial `ONE-PERSON` specification we easily "see" that the age can never decrease. It may not be equally simple to "manually analyze" much larger and complex specifications of communication protocols and rocket controllers. We therefore need tool support for further analyzing specifications.

Maude provides a *search* command which searches through all behaviors from a given initial state and returns all—or a user-given number of—states which can be reached from the initial state and which satisfy the given search condition. The search may be restricted to analyze all behaviors up to n rewrite steps.

Maude's search command searches in *breadth-first* way through all behaviors from the initial state. That is, Maude first visits all terms reachable in *one* (sequential) rewrite step from the initial state, then it visits all states reachable in *two* steps from the initial state, and so on. Maude stores the visited states and ignores states which have been visited earlier in the search. This kind of search may not terminate if there is an infinite number of states which are reachable from the initial state.

The basic forms of the search command are

`search` t_0 *arrow* *pattern* .

and

`search` t_0 *arrow* *pattern* `such that` *cond* .

The term t_0 is the initial state, *pattern* is a term which can contain variables, and *cond* is a condition which has the same form as a condition of a rewrite rule. A term t satisfies the search condition if *pattern* matches t and *cond* holds for the matching substitution. The *arrow* is either `=>1`, `=>*`, `=>+`, or `=>!` and indicates in how many (sequential) rewrite steps the desired terms are to be found:

`=>1:` states which can be reached in *exactly one* step from the initial state t_0;

`=>*:` states reachable in *zero or more* steps;

`=>+:` states reachable in *one or more* steps; and

`=>!:` states that cannot be further rewritten.

(The arrows correspond to the arrow kinds on page 81.)

Since Maude cannot do E-matching for checking whether a term is E-matched by the search pattern, all states are reduced to their E-normal forms before matching (possibly modulo associativity and/or commutativity) is checked. The patterns must therefore be *ground irreducible* (see page 173) which should imply that *the pattern should be a constructor term.*

Example 72 The command

```
Maude> search person("Babko", 84, widow) =>1 P:Person .
```

is looking for all states reachable in *one* step from `person("Babko", 84, widow)` that matches the variable `P:Person`. (Remember from Section 2.1.6 that variables of the form *varname*`:`*sort* can be used without being explicitly declared. A search pattern can use such undeclared variables as well as the variables defined in the module in which the search is to take place.) All terms of sort `Person` are matched by the variable `P:Person`, so the above command searches for all states reachable in one step from `person("Babko", 84, widow)`.

The output from a successful search is the matching substitution:

```
Solution 1 (state 1)
P:Person --> person("Babko", 85, widow)

No more solutions.
```

(The search is performed on the module `ONE-PERSON` given in Section 5.6.2 and not on the extended module which is the result of solving Exercise 115.)

To find out what could happen to `"Peter"` when he is 41 years one may give the command

```
Maude> search person("Peter", 38, single) =>* person("Peter", 41, S) .
```

to which Maude answers with the matching substitutions:

```
Solution 1 (state 6)
S --> single

Solution 2 (state 10)
S --> engaged

Solution 3 (state 14)
```

```
S --> married

No more solutions.
```

We can also check whether the above person can become younger:

```
Maude> search person("Peter", 38, single) =>* person("Peter", N, S)
       such that N < 38 .

No solution.
```

Finally, one may be interested in how it may end; that is, what are the possible final states from which nothing more will happen?

```
Maude> search person("Peter", 38, single) =>! person("Peter", N, S) .

Solution 1 (state 2900)
N --> 1001
S --> married

No more solutions.
```

♠

The command

`show path` n `.`

prints the shortest rewrite sequence from the initial state to the state number n in the previous search, and the command

`show path labels` n `.`

prints the sequence of rules (represented by their labels) applied in that rewrite sequence.

Example 73 In Example 72 we searched for all states where the age of `"Peter"` is `41`. The solution in which this person was in state `married` had number 14. One could then give the command `show path 14 .` to let Maude display the path leading to the `married` state:

```
Maude> show path 14 .

state 0, Person: person("Peter", 38, single)
===[crl person(X, N, S) => person(X, N + 1, S) if N <= 1000 = true
    /\ S =/= deceased = true [label birthday] .]===>
state 1, Person: person("Peter", 39, single)
...
```

```
state 6, Person: person("Peter", 41, single)
===[crl person(X, N, S) => person(X, N, engaged) if N >= 15 = true
    /\ S == single or S == divorced = true [label successful-proposal] .]===>
state 10, Person: person("Peter", 41, engaged)
===[rl person(X, N, engaged) => person(X, N, married) [label marriage] .]===>
state 14, Person: person("Peter", 41, married)
```

♠

A search (with an arrow different from `=>1`) will not terminate if there are infinitely many states reachable from the initial state. This is because the search command looks for *all* results. One may therefore put an upper bound on the number of solutions using the syntax

`search [`n`] ...`

and/or put an upper bound d on the number of rewrite steps in the behaviors using the syntax

`search [`n`,`d`] ...`

or

`search [,`d`] ...`

Exercise 133 *Assume that we are in pre-Gilgamesh times and that there is no age limit in the rule* `birthday` *in the module* `ONE-PERSON`*. Explain why the execution of the search command*

```
Maude> search person("Peter", 38, single) =>* person("Peter", 40, S) .
```

would not *terminate.*

Example 74 Consider the specification `ONE-FOOTBALL-GAME` on page 154 in which the away team could not win/lead in `rew` and `frew` simulations. To settle the issue of whether the away team can lead we could try the command

```
Maude> search [1] "Packers" vs "49ers" 0 : 0  =>* "Packers" vs "49ers"  M : N
       such that  M < 7 /\ N > 40 .

Solution 1 (state 691)
M --> 0
N --> 42
```

♠

Exercise 134 *Use Maude to check whether it is possible in American football to reach the state*

```
"49ers" vs "Giants"  39 : 38
```

from the undesired state

```
"49ers" vs "Giants"  14 : 38.
```

If it is possible, check whether Maude's path leading to the desired state corresponds to what happened in the game that glorious January afternoon (touchdown, 2-pt conversion, touchdown, 2-pt conversion, field goal, touchdown).

The execution of a search command may fail to terminate even when we restrict the number of desired solutions. A search for *one* solution will fail if there is no solution and the set of reachable states is infinite.

Example 75 The execution of the search command

```
Maude> search [1] "49ers" vs "Giants" 39 : 41 =>* "49ers" vs "Giants" 39 : 38 .
```

will fail to terminate in `ONE-FOOTBALL-GAME`. Why? ♠

To summarize, because the search is performed in a breadth-first way, n desired solutions will always be found if there exist at least n reachable states satisfying the search condition. If there are not at least n solutions, then the search will first output the existing solutions, and will then either terminate if only a *finite* number of *distinct* states are reachable from the initial state, or will loop forever (desperately searching for the remaining non-existing solutions) otherwise. Of course, if a bound on the number of rewrites in the behaviors is added, a search command will (almost) always terminate.

Exercise 135

1. *Assume that Maude instead would search the rewrite paths from the initial state in a "depth-first" way. Could we still guarantee that searching for n solutions would always be successful if there exist at least n solutions?*
2. *Can you use Maude's search command to prove that it is impossible to go from the state* `person("Gilgamesh", 50, married)` *to a state in which the noble man's age is less than 50, provided the* `birthday` *rule has no age limit?*
3. *Prove why it is* impossible *(no matter how smart you are) to implement a search command which always terminates and which can be used to find whether there exists (at least)* one *reachable state from the initial state that is matched by the search pattern. (You may of course assume that the underlying* equational *theory of the specification is confluent and terminating.)*

Exercise 136 *In this exercise you should use Maude's search command to analyze the coffee bean game described in Section 5.6.3.*

1. *What are the possible results of the game when starting from the initial bean sequence*

   ○ ● ○ ○ ○ ● ● ○ ○ ● ● ● ○ ○

   Ask Maude to display the run which resulted in the fewest remaining beans.

2. *Is it the case that each state reachable from an initial state with an even number of black beans will contain an even number of black beans? Test this on the initial states*

   ○ ● ○ ○ ○ ● ● ○ ○ ● ● ● ○ ○

   and

   ○ ○ ● ● ○ ● ○ ●

3. *Search for all the results of playing the game when the initial state contains an odd number of black beans. Try this for a couple of initial states, such as*

   ● ○ ● ○ ○ ○ ● ● ○ ○ ● ● ● ○ ○

   and

   ● ○ ○ ● ● ○ ● ○ ●

   Do you see any pattern in the answers?

4. *Check some more examples and suggest whether it is* always *possible to end of with* one *coffee bean no matter what is the initial state.*

Exercise 137 *Use Maude to prove that each nonextensible rewrite sequence starting with the list*

```
8 4 0 -3 76 54 21
```

ends with the sorted list

```
-3 0 4 8 21 54 76
```

in the module `SORT` *on page 166.*

Note finally that since a search analyzes the behaviors starting from *one* given initial state, a search can never be used to prove something for *all* possible inputs. While search could be used to prove that the specification `SORT` would always sort the list

```
8 4 0 -3 76 54 21 0 -9 3 23
```

correctly, we cannot claim that it will sort all possible inputs correctly. For proving properties about all possible inputs one would have to use inductive techniques, just as we used inductive techniques in Section 4.2.3 to prove properties like

$$\texttt{length}(\texttt{concat}(l, l')) = \texttt{length}(l) \ \texttt{+} \ \texttt{length}(l')$$

for *all* lists l and l'.

Search is nevertheless a quick and useful way of analyzing specifications and eliminating errors before the cumbersome task of proving correctness is undertaken.

6.4 Further Analyzing Rewriting Logic Specifications

Search is not always sufficient to analyze a specification from a single state. One could for instance be interested in checking more complex properties than whether a state is reachable from the initial state, and/or one would like to guide the application of the rules so that only "relevant" behaviors are explored. (For example, while the possibility of losing *any* packet in a data network implies that *all* packets could *in principle* be lost in some behavior, we would normally only analyze behaviors in which not more than, say, 40% of the packets are lost.)

6.4.1 Temporal Logic Model Checking

Maude's useful search command analyzes all behaviors from an initial state with respect to the property

> is a state such-and-such reachable from the initial state.

It is often desirable to analyze a reactive system with respect to more sophisticated properties such as:

1. will the nuclear power plant controller *always* reach a `shutDown` state after some dangerously bad values are reported from the sensors?
2. does there exist a behavior of the power plant in which the plant will explode?
3. will a state be reached in which `"Ifi"` is 22 years old in *each* possible behavior starting from the state `person("Ifi", 20, single)`?
4. is, in each possible behavior starting with some person, each state in which a person is `married` preceded by a state in which the person is `engaged`?
5. starting with the state `person("Lizzie", 35, married)`, is it the case for each behavior that the age stays 35 until it reaches 36?
6. in each behavior from the state `person("Oreg", 55, married)` the age of `"Oreg"` is ≥ 55.

Exercise 138 *Which of the above properties can be checked by using Maude's search command? Explain for each property why it can/can not be checked for a given initial state.*

The properties above are properties about behaviors ("should hold for each state in the behavior"; "must eventually happen in the behavior"; "each X state must preceded by a Y state in the behavior"; "X must hold for all states in the behavior until Y holds" etc.) which should hold for *all* or *some* behaviors and may talk about relative order of events in a behavior ("preceded by"). Such properties about behaviors can often be expressed in various forms of *temporal logic* (see e.g. [34, 65]). Maude is equipped with a (linear) temporal logic *model checker* [32] in which temporal logic properties can be specified and checked, provided that the

set of states reachable from the initial state is finite. The Maude model checker is comparable with state-of-the-art model checkers such as SPIN [49] in terms of efficiency.

We will neither use temporal logic nor Maude's model checker in this course, but we will study some classes of temporal properties such as *invariants* in Chapter 10.

6.4.2 User-Defined Analysis Strategies

In addition to the execution capabilities mentioned above, the user could write her own *execution strategies* for executing a Maude module in a user-defined way. Most simulation and analysis tools have a certain set of assumed useful analysis strategies hardwired into the tools. While that approach has certain advantages such as ease of first use and the possibility of optimizing the code, it also has some drawbacks. It restricts the possible strategies that can be applied (the setting in which every tenth packet in a network is lost can be difficult to execute) and a new language is needed to write more extensive strategies. The Maude approach is different.

> □ Maude uses the fact that rewriting logic is *reflective* [10, 15, 16] and that Maude modules therefore can be represented as *terms* in other Maude modules (at the so-called *meta-level*), and that the object module and its rewrite steps can be analyzed/simulated in the meta-level Maude module. □

Execution strategies are defined in Maude itself by equations and rewrite rules. The user can write his own strategies in Maude starting with a basic built-in building block of "applying a given rule once." Apart from allowing the user the flexibility in defining his strategies, the strategy is no longer an extra-logical feature, but is written as a rewrite specification and can be reasoned about as such. The advantages of "predefined" strategies can be retained by saving a set of useful strategies in libraries.

Chapter 7

Concurrent Objects in Maude

A distributed system can naturally be modeled as an object system in which each component of the distributed system is represented by an object. The whole system—being a collection of (communicating) components/subsystems—can then be represented by a *multiset* of objects. The different components *may* communicate with each other through message passing, so the global state of a system may also contain messages being sent between objects.

This chapter starts by explaining how concurrent objects may be modeled directly in rewriting logic. Section 7.2 presents *Full Maude*, a Maude "interface" which supports object-oriented specification by adding "syntactic sugar" for defining classes, messages, objects, and rules in a more "object-oriented notation." Finally, this chapter shows how the classical *dining philosophers problem* can be specified in Maude in an object-oriented way.

The theoretical foundations of how various object-oriented concepts can be represented in rewriting logic are thoroughly discussed in [70].

7.1 Modeling Concurrent Objects in Maude

One way of modeling an object in Maude is to let the term

< o : C | att_1: val_1, ..., att_n: val_n >

denote an *object* of class C which has the *name* (or *identifier* or "address") o and *attributes* att_1 to att_n whose "current" values are val_1 to val_n, respectively. Continuing our ongoing example from Chapter 5, a `Person` object in a certain state could be e.g.,

```
< "Peter" : Person | age: 38, status: single > .
```

Letting a sort `Object` denote objects, a class C could be declared with a constructor

op <_: C | att_1:_, ..., att_n:_> : Oid s_1 ... s_n -> Object [ctor] .

where `Oid` is some sort denoting object identifiers, and s_1 to s_n are the sorts of the attributes att_1 to att_n. A class `Person` may therefore be declared

```
sorts Object Oid .
subsort String < Oid .
op <_: Person | age:_, status:_> : Oid Nat Status -> Object [ctor] .
```

where the data type `Status` is the same as declared in Chapter 5. The objects of the above form are then just ordinary terms of sort `Object`.

A system may also contain messages, and we may therefore have a sort `Msg` whose values are messages. A distributed system may then be seen as a multiset of objects and of messages traveling between objects. The sort `Configuration` is often used for such multisets and is defined as expected:

```
sorts Object Msg Configuration .
subsorts Object Msg < Configuration .
op none : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration [ctor assoc comm id: none] .
```

A term of sort `Configuration` could be for example

```
< "Peter" : Person | age: 38, status: single >
< "Mette" : Person | age: 39, status: married("Rich") >
< "Lizzie" : Person | age: 35, status: single > .
```

7.1.1 Rewrite Rules

Rewrite rules define the behavior of objects and their treatment of messages. There are no restrictions on the shape of the rules. The left-hand side is a multiset of objects and messages, and so is the right-hand side. A rule may involve zero, one, or many objects, and zero or more messages. The objects need not be the same on both sides: objects may be created and/or destroyed by a rule, and so may messages.

The "simplest" rules are those with only one (and the same) object on either side, such as

```
vars X X' : String .  vars N N' : Nat .  vars S S' : Status .

rl [birthday] : < X : Person | age: N, status: S >  =>
                < X : Person | age: N + 1, status: S > .
```

The rule defines the local state change for an object. For example, with the above rule we have that

```
< "Mette" : Person | age: 21, status: single >  C  ⟶
< "Mette" : Person | age: 22, status: single >  C
```

holds for any configuration C. This is because of the **Congruence** rule in rewriting logic, since

```
< "Mette" : Person | age: 21, status: single >   ⟶
< "Mette" : Person | age: 22, status: single >
```

holds by **Replacement** and **Equality**, and $C \longrightarrow C$ holds by **Reflexivity**, and by **Congruence** with respect to the operator `__` we have the above sequent.

Objects can perform independent actions concurrently. If $o_1 \longrightarrow o_1'$, ..., $o_n \longrightarrow o_n'$ are rewrite steps for objects $o_1, \ldots o_n$, then, again by the **Congruence** rule of rewriting logic with respect to the operator `__` (which due to associativity can be seen as a n-ary operator `__...__` for any n), there is a concurrent step

$$o_1 \ \ldots \ o_n \longrightarrow o_1' \ \ldots \ o_n'.$$

There is for example a concurrent one-step rewrite

```
< "Peter" : Person | age: 20, status: single >
< "Mette" : Person | age: 21, status: single >
< "Lizzie" : Person | age: 17, status: single >
                        ⟶
< "Peter" : Person | age: 21, status: single >
< "Mette" : Person | age: 22, status: single >
< "Lizzie" : Person | age: 18, status: single >
```

in which three birthdays are celebrated concurrently in one step.

Multiple Objects in a Rule

More than one object may be involved in a rewrite step. An engagement in our running example may be modeled by the following rule:

```
crl [engagement] : < X : Person | age: N, status: single >
                   < X' : Person | age: N', status: single >
                      =>
                   < X : Person | age: N, status: engaged(X') >
                   < X' : Person | age: N', status: engaged(X) >
                       if N > 15 /\ N' > 15 .
```

Such a rule models a kind of *synchronous communication* where two (or more) objects meet and perform an action together. Two objects may always meet in this way, due to commutativity and associativity of the `Configuration` constructor `__`. For example there is a (one-step sequential) rewrite

```
< "Imtiaz" : Person | age: 29, status: single >
< "Peter" : Person | age: 27, status: single >
< "Maiken" : Person | age: 27, status: single >
                     ⟶
< "Imtiaz" : Person | age: 29, status: engaged("Maiken") >
< "Peter" : Person | age: 27, status: single >
< "Maiken" : Person | age: 27, status: engaged("Imtiaz") >
```

since the state

```
< "Imtiaz" : Person | age: 29, status: single >
< "Peter" : Person | age: 27, status: single >
< "Maiken" : Person | age: 27, status: single >
```

is the same as

```
< "Imtiaz" : Person | age: 29, status: single >
< "Maiken" : Person | age: 27, status: single >
< "Peter" : Person | age: 27, status: single >
```

because `__` is declared `assoc` and `comm`, and terms which are equivalent modulo these attributes of a function symbol are always considered to be "the same" term.

Exercise 139

1. *Is there a one-step concurrent rewrite*

   ```
   < "Imtiaz" : Person | age: 29, status: single >
   < "Peter" : Person | age: 27, status: single >
   < "Maiken" : Person | age: 27, status: single >
                        ⟶
   < "Imtiaz" : Person | age: 29, status: engaged("Maiken") >
   < "Peter" : Person | age: 28, status: single >
   < "Maiken" : Person | age: 27, status: engaged("Imtiaz") >
   ```

   in which `"Peter"` *at least gets to celebrate his birthday while the others are becoming engaged?*

2. *Is there a one-step concurrent rewrite*

   ```
   < "Imtiaz" : Person | age: 29, status: single >
   < "Peter" : Person | age: 28, status: single >
   < "Maiken" : Person | age: 27, status: single >
                        ⟶
   < "Imtiaz" : Person | age: 29, status: engaged("Maiken") >
   < "Peter" : Person | age: 28, status: single >
   < "Maiken" : Person | age: 28, status: engaged("Imtiaz") >
   ```

   in which `"Maiken"` *celebrates both her birthday* and *her engagement in the same step? Is this natural?*

3. *Declare the rule for marriage.*

4. *Use Maude's search command to prove that there is a behavior from the above state to a state in which the* `age` *of* `"Imtiaz"` *is* `50`*. (Try to avoid mentioning the other objects explicitly in the search pattern.)*

Creation and Deletion of Objects

It is not necessary that the *same* objects occur on both sides of a rule. Some objects may be "removed" from the right-hand side, as in the following rule modeling the death of a `Person`:

```
rl [death] : < X : Person | age: N, status: S >  =>  none .
```

The application of this rewrite rule could take the state

```
< "Hamlet" : Person | age: 28, status: single >
< "Old Norway" : Person | age: 67, status: married("Queen of Norway") >
< "Fortinbras" : Person | age: 40, status: single >
```

to the state

```
< "Hamlet" : Person | age: 28, status: single >
< "Fortinbras" : Person | age: 40, status: single > .
```

The right-hand side of a rule may contain objects not present in the left-hand side, in which case these additional objects are "created" by the rule and added to the state. For example, the birth of a child to a couple may be modeled by the following (fairly old-fashioned) rule:

```
crl [birth] : < X : Person | age: N, status: married(X') >
              < X' : Person | age: N', status: married(X) >
                 =>
              < X : Person | age: N, status: married(X') >
              < X' : Person | age: N', status: married(X) >
              < X'' : Person | age: 0, status: single >  if N < 60 or N' < 60 .
```

Then there is for example a rewrite step in which `"Aphrodite"` is born:[1]

```
< "Uranus" : Person | age: 999, status: married("?") >
< "Chronos" : Person | age: 900, status: single >
< "Zeus" : Person | age: 800, status: married("Dione") >
< "Dione" : Person | age: 21, status: married("Zeus") >
                        ⟶
< "Uranus" : Person | age: 999, status: married("?") >
< "Chronos" : Person | age: 900, status: single >
< "Zeus" : Person | age: 800, status: married("Dione") >
< "Dione" : Person | age: 21, status: married("Zeus") >
< "Aphrodite" : Person | age: 0, status: single >.
```

Exercise 140 *Prove that the above rewrite step follows from the rule* `birth` *and the deduction rules of rewriting logic.*

[1]The author must admit that he is not overly confident in the correctness of the age of the characters in this example.

There is a slight problem with the above rule `birth`: the variable `X''` is introduced in the right-hand side of the rule, which means that the new object could have any name, both desirable names such as `"Aphrodite"` and `"Lizzie"` and undesired names such as `"Pol Pot"`, `"Stalin"`, `"Hitler"`, `"Kissinger"`, `"Mao"`, and `"Perle"`. Furthermore, Maude's `rew` and `search` commands ignore such rules since Maude does not really know which value to give to `X''`. One idea to solve this problem could be to have an extra object containing a list of attractive names from which one could choose. Using the module

```
fmod STRING-LIST is
  protecting STRING .
  sort StringList .
  subsort String < StringList .
  op nil : -> StringList [ctor] .
  op __ : StringList StringList -> StringList [ctor assoc id: nil] .
endfm
```

which defines lists of strings, a class containing attractive names could be declared

```
  op <_: ListOfNames | OKnames:_> : Oid StringList -> Object [ctor] .
```

A state could then contain persons and a `ListOfNames` object containing a list of desired names:

```
  < "Wonderful Names" : ListOfNames | OKnames: "Mette" "Aphrodite" "Lizzie"
                                               "Fisk" "Chomsky"  >
  < "Uranus" : Person | age: 999, status: married("Gaia") >
  < "Chronos" : Person | age: 900, status: single >
  < "Zeus" : Person | age: 800, status: married("Dione") >
  < "Dione" : Person | age: 21, status: married("Zeus") >
```

The following rule is executable in Maude and chooses (pseudo-) nondeterministically among the favored names:

```
  vars L L' : StringList .
  crl [birth2] : < X : Person | age: N, status: married(X') >
                 < X' : Person | age: N', status: married(X) >
                 < X'' : ListOfNames | OKnames: L X''' L' >
                     =>
                 < X : Person | age: N, status: married(X') >
                 < X' : Person | age: N', status: married(X) >
                 < X'' : ListOfNames | OKnames: L X''' L' >
                 < X''' : Person | age: 0, status: single >  if N < 60 or N' < 60 .
```

Exercise 141

1. *Use the above approach and define a rule* `twinBirth` *for the birth of twins in* one *step. You may assume that the list of good names contains at least two distinct names if you want to give the twins different names.*

2. *Modify the rule* `birth2` *so that no person born later can have the same name as the current newborn.*

3. *Can more than one person be born at the same time using rule* `birth2` *(under the assumption that there is only one* `ListOfNames` *object)? Can two* `birth`*-actions take place concurrently?*

Exercise 142 *Give rules for separation and divorce.*

Exercise 143 *As the specification now stands, an engaged, married, or separated person cannot remarry if her/his spouse/fiancé(e) dies. Change the specification so that a married/engaged person becomes single when the spouse/fiancé(e) dies.*

Exercise 144 *Execute your specification, which should now be complete, in Maude. It is usually a good idea to use the* fair *rewrite command* `frew` *when executing object-oriented specifications to ensure that each object get its chance to rewrite.*

Communication Through Message Passing

The above specification is not particularly realistic, since a separation or a divorce these days does not necessarily happen when the parties involved meet. More often a letter (from the lawyer?), an e-mail, or a message on the answering machine is used to express the desire to separate or divorce. Therefore we declare two messages

```
ops separate divorce : Oid -> Msg [ctor] .
```

where `separate(X)` is a message *to* `X` intended to mean that `X`'s spouse wants to separate. In the rule

```
rl [separationInit] :  < X : Person | age: N, status: married(X') >  =>
                       < X : Person | age: N, status: separated(X') >
                       separate(X') .
```

which models the setting where `X` initiates a separation, the message `separate(X')` is created. In the rule

```
rl [acceptSeparation] :  separate(X)
                         < X : Person | age: N, status: married(X') >  =>
                         < X : Person | age: N, status: separated(X') > .
```

the message `separate(X)` is consumed by the unsuspecting spouse, who can just accept that they are separated.

The message passing is modeled "abstractly" in that the "traveling" of the message is due to the fact that `__` is associative and commutative. For example, from the state

```
< "Chronos" : Person | age: 900, status: single >
< "Zeus" : Person | age: 800, status: married("Dione") >
< "Hera" : Person | age: 19, status: single >
< "Dione" : Person | age: 21, status: married("Zeus") >
```

`"Zeus"` may want a separation (and later a divorce) so that he can marry his sister `"Hera"`. In one `separationInit` the above state rewrites to

```
< "Chronos" : Person | age: 900, status: single >
< "Zeus" : Person | age: 800, status: separated("Dione") >
separate("Dione")
< "Hera" : Person | age: 19, status: single >
< "Dione" : Person | age: 21, status: married("Zeus") >
```

which, again due to associativity and commutativity of `__`, is the same as

```
< "Chronos" : Person | age: 900, status: single >
< "Zeus" : Person | age: 800, status: separated("Dione") >
< "Hera" : Person | age: 19, status: single >
separate("Dione")
< "Dione" : Person | age: 21, status: married("Zeus") >
```

which rewrites by the use of rule `acceptSeparation` to

```
< "Chronos" : Person | age: 900, status: single >
< "Zeus" : Person | age: 800, status: separated("Dione") >
< "Hera" : Person | age: 19, status: single >
< "Dione" : Person | age: 21, status: separated("Zeus") >.
```

We often call communication by message passing *asynchronous communication* because the objects do not synchronize in performing the action. Quite a lot of time (e.g., some `birthday` events) may elapse between the `separationInit` and the corresponding `acceptSeparation` event. It may also happen that a party initiates a divorce before the spouse has consumed the `separate` message.

Exercise 145

1. *What happens if both parties in a couple initiate a separation at the same time?*
2. *Define rules for initiating and accepting a divorce and show that* `"Zeus"` *and* `"Dione"` *can indeed be divorced, even if* `"Dione"` *initiates the divorce.*
3. *Show that there is a scenario in which a* `divorce` *message may be consumed before a* `separate` *message to the same object is consumed. (If there is no such scenario, explain why!)*

4. *Use Maude's search command to check whether it is possible to reach a state in your modified specification in which a person x is married to a person y, while person y is married to someone else. Start with one of the initial states mentioned earlier in this chapter.*

5. *A separation should not immediately lead to a divorce: there should be a (usually microscopic) possibility that the parties can get back together. That is, model a setting in which a separated person can send a* `backAgain?` *message to the spouse, to which the spouse should respond with a* `yes` *or a* `no` *message. (Try to design your system so that a person who sends a* `backAgain?` *message waits for an answer before sending a* `divorce` *message. Also note that a spouse could send a* `backAgain?` *message while the other sends a* `divorce` *message at the same time.)*

The Specification

The executable Maude specification—with the exercise parts omitted and the `death` rule commented away!—can be given as follows:

```
mod OO-POPULATION is
  protecting STRING-LIST .
  protecting NAT .

  *** Objects, messages, object names, and configurations:
  sorts Oid Object Msg Configuration .
  subsorts Object Msg < Configuration .
  op none : -> Configuration [ctor] .
  op __ : Configuration Configuration -> Configuration
                                         [ctor assoc comm id: none] .

  subsort String < Oid .          *** Object names are String's

  *** Classes:
  op <_: ListOfNames | OKnames:_> : Oid StringList -> Object [ctor] .
  op <_: Person | age:_, status:_> : Oid Nat Status -> Object [ctor] .

  *** Messages:
  ops separate divorce : Oid -> Msg [ctor] .

  sort Status .
  op single : -> Status [ctor] .
  ops engaged married separated : Oid -> Status [ctor] .

  vars X X' X'' X''' : String .  vars N N' : Nat .  var S : Status .
  vars L L' : StringList .

  rl [birthday] : < X : Person | age: N, status: S >  =>
                  < X : Person | age: N + 1, status: S > .

  crl [engagement] : < X : Person | age: N, status: single >
                     < X' : Person | age: N', status: single >
                        =>
```

```
                    < X : Person | age: N, status: engaged(X') >
                    < X' : Person | age: N', status: engaged(X) >
                        if N > 15 /\ N' > 15 .

  *** rl [death] : < X : Person | age: N, status: S >  =>  none .

  crl [birth2] : < X : Person | age: N, status: married(X') >
                 < X' : Person | age: N', status: married(X) >
                 < X'' : ListOfNames | OKnames: L X''' L' >
                     =>
                 < X : Person | age: N, status: married(X') >
                 < X' : Person | age: N', status: married(X) >
                 < X'' : ListOfNames | OKnames: L X''' L' >
                 < X''' : Person | age: 0, status: single >
                     if N < 60 or N' < 60 .

  rl [separationInit] :  < X : Person | age: N, status: married(X') >  =>
                         < X : Person | age: N, status: separated(X') >
                         separate(X') .

  rl [acceptSeparation] :  separate(X)
                           < X : Person | age: N, status: married(X') >  =>
                           < X : Person | age: N, status: separated(X') > .

  *** Some rules are exercises and are therefore missing
endm
```

Since there is no rule for marriage in this specification, the best we can hope for is an engagement when executing the specification in Maude:

```
Maude> frew [10]  < "Peter" : Person | age: 38, status: single >
                  < "Lizzie" : Person | age: 35, status: single >
                  < "Sam The Snake" : Person | age: 41, status: single >
                  < "Names" : ListOfNames | OKnames: "Mette" "Fisk" > .

result (sort ...): < "Names" : ListOfNames | OKnames: "Mette" "Fisk" >
                   < "Lizzie" : Person | age: 38, status: engaged("Peter") >
                   < "Peter" : Person | age: 40, status: engaged("Lizzie") >
                   < "Sam The Snake" : Person | age: 43, status: single >
```

One final remark: The states can be quite large in object-oriented specifications. To avoid having to type large states each time you execute your specification it can be useful to define "abbreviations" for initial states such as

```
  op greeks : -> Configuration .
  eq greeks =
    < "Wonderful Names" : ListOfNames | OKnames: "Mette" "Aphrodite" "Lizzie" >
    < "Uranus" : Person | age: 999, status: single >
    < "Chronos" : Person | age: 900, status: single >
    < "Zeus" : Person | age: 800, status: married("Dione") >
```

```
    < "Dione" : Person | age: 21, status: married("Zeus") > .
```

inside the module, which can then be executed by giving the command

```
Maude> frew [30] greeks .
```

7.2 Concurrent Objects in Full Maude

We have just seen that concurrent objects can quite naturally be specified directly in rewriting logic. Apart from "meta-language" use of Maude as specification and execution environment for other languages, most "real" Maude specifications are object-oriented specifications (see e.g. [19, 82, 83, 77]).

Although concurrent objects can be naturally specified directly in Maude, one could wish to provide syntactic support for object-oriented concepts such as classes, messages, etc. Furthermore, it is slightly trickier to achieve *inheritance* through subclasses directly in Maude. Maude is supposed to provide exactly this kind of support for object-oriented specification in the future. In the meantime we use *Full Maude* [28, 13] to specify and execute object-oriented systems until that version of Maude becomes available.

7.2.1 Full Maude

Full Maude is a *prototype* of Maude's support for object-oriented specification and for its operations on modules and module parameterization [27]. Full Maude is a Maude specification/program written by Francisco Durán and is given in the file `full-maude.maude`.

Full Maude provides support for object-oriented specifications in *object-oriented modules*, which give syntactic support for declaring *classes*, *subclasses*, and *messages*, and for allowing to write shorter rewrite rules by omitting attributes that do not affect, and are not affected by, the application of the rule. Likewise, Full Maude extends Maude's search command to the object-oriented case by also taking subclasses into account in searches, and by allowing us to only mention relevant attributes in the search pattern. (These extensions of the search command seem not to work too well in the current version 2.3 of Full Maude.) Full Maude works by translating an object-oriented module into an ordinary Maude module (along the lines presented in the previous section), which can then be executed by Maude.

7.2.2 Using Full Maude

Full Maude is an ordinary Maude specification given in the file `full-maude.maude` and is started as an ordinary Maude module, that is, by starting Maude with

```
UNIX> maude full-maude.maude
```

or by giving the Maude command

```
Maude> load full-maude.maude
```

to which Maude answers

```
      Full Maude 2.3 `(February 12th`, 2007`)
```

Input is given to Full Maude by enclosing it between a pair of parentheses. Full Maude accepts the modules and commands of Maude with some exceptions[2]:

```
Maude> (fmod NAT-ADD is
>         sort Nat .
>         op 0 : -> Nat .
>         op s : Nat -> Nat [ctor] .
>         op _+_ : Nat Nat -> Nat .
>
>         vars M N : Nat .
>         eq 0 + M = M .
>         eq s(M) + N = s(M + N) .
>       endfm)
rewrites: 1065 in 20ms cpu (70ms real) (53250 rewrites/second)
Introduced module NAT-ADD

Maude> (show module .)
rewrites: 129 in 10ms cpu (10ms real) (12900 rewrites/second)

fmod NAT-ADD is
  protecting BOOL .
  sorts Nat .
  op _+_ : Nat Nat -> Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  eq 0 + M:Nat = M:Nat .
  eq s(M:Nat) + N:Nat = s(M:Nat + N:Nat) .
endfm

Maude> (red s(s(0)) + s(0) .)
rewrites: 224 in 0ms cpu (0ms real) (~ rewrites/second)
reduce in NAT-ADD :
  s(s(0)) + s(0)
result Nat :
  s(s(s(0)))
```

One Full Maude command worth mentioning here is `(show all .)` which displays the Maude module which results from Full Maude's translation. Full Maude is a huge program and cannot be completely without errors. If you encounter some unexpected problems it may be worth using this command to see how Full Maude translated your specification into Maude.

The Maude command `trace exclude FULL-MAUDE .` should be given (without parentheses) after the command `set trace on .` to trace a Full Maude execution.

[2]The rewrites below refer to the rewrites Full Maude performs in order to transform its input into a Maude module, and to other computations done by Full Maude.

7.2.3 Object-Oriented Modules in Full Maude

Object-oriented modules are declared with syntax

```
(omod M is ... endom)
```

The sorts `Oid`, `Object`, `Msg`, and `Configuration` with the constructors described above are defined in the following module `CONFIGURATION` (given in the file `prelude.maude`) which is automatically imported in any object-oriented module[3].

```
mod CONFIGURATION is
  sorts Attribute AttributeSet .
  subsort Attribute < AttributeSet .
  op none : -> AttributeSet .
  op _,_ : AttributeSet AttributeSet -> AttributeSet
                          [format (o m so o) ctor assoc comm id: none] .

  sorts Oid Cid Object Msg Portal Configuration .
  subsort Object Msg Portal < Configuration .
  op <_:_|_> : Oid Cid AttributeSet -> Object
                                       [ctor object format (b r b g b o b o)] .
  op none : -> Configuration .
  op __ : Configuration Configuration -> Configuration
                        [format (o n o) ctor config assoc comm id: none] .
  op <> : -> Portal [ctor] .
endm
```

The sort `Cid` denotes *class identifiers*, and the sort `AttributeSet` denotes *multisets* of attribute-value-pairs. Since attributes are multisets, the order in which the attributes to an object are written does not matter.

Classes are declared with syntax

```
class C | att_1 : s_1, ..., att_n : s_n .
```

In our ongoing example we could therefore write

```
class Person | age : Nat, status : Status .
```

Remark: Note the blank everywhere, especially *before* and after the colon ':'.

Although the *sort* `Oid` is predefined, no *values* are predefined in this sort, so we could declare

```
subsort String < Oid .
```

[3]The module below is not the "standard" module, but has been slightly changed by the author to get better formatted output.

if strings are to serve as object identifiers. Objects are written as before, with the difference that each colon should be preceded and followed by a blank, and that the order of attributes does not matter:

```
< "Peter" : Person | status : single, age : 38 >.
```

An object with no attributes is written with syntax

```
< o : EmptyClass | >.
```

Only a few among the possibly many attributes of an object may affect, or be affected by, the application of a rewrite rule. By convention, only attributes whose values are changed need to be present in the right-hand side of a rule, and only those attributes whose values affect the applicability of a rule, the new values of the attributes changed by the rule, or the messages need to be present in the left-hand side of a rule.

For example, since the status of a person is not changed in the `birthday` rule, and the status does not affect the "next" age of a person, the `status` attribute may be omitted from the birthday rule:

```
crl [birthday] :
    < X : Person | age : N >  =>  < X : Person | age : N + 1 >  if N < 999 .
```

Similarly, the age of a person influences whether the person can be engaged, but is not itself changed by the engagement, so the attribute `age` may be omitted from the right-hand side:

```
crl [engagement] :  < X : Person | age : N, status : single >
                    < X' : Person | age : N', status : single >
                       =>
                    < X : Person | status : engaged(X') >
                    < X' : Person | status : engaged(X) >
                       if N > 15 or N' > 15 .
```

Finally, as we all know from the magnificent Sumeric epic *Gilgamesh*, "When the high gods get together/ And Mammetu is with them, the goddess of fate/ They decide each man's life/ And his death, but say not the day"[4]. That is, death may be modeled by the rule

```
rl [death] :  < X : Person | >  =>  none .
```

The partial specification—including a command for reading in the file `full-maude.maude` at the start of the session and a prototyping command for executing the specification—can be given in a file as follows:

[4]This is a homemade translation from the Norwegian edition [38], which is highly recommended to the Norwegian reader.

```
load full-maude

(omod POPULATION is
  protecting NAT .
  protecting STRING .
  sort Status .
  op single : -> Status [ctor] .
  ops engaged married separated : Oid -> Status [ctor] .
  subsort String < Oid .

  class Person | age : Nat, status : Status .

  vars N N' : Nat .   vars X X' : String .

  crl [birthday] :  < X : Person | age : N >  =>
                    < X : Person | age : N + 1 >  if N < 999 .

  crl [engagement] :  < X : Person | age : N, status : single >
                      < X' : Person | age : N', status : single >
                         =>
                      < X : Person | status : engaged(X') >
                      < X' : Person | status : engaged(X) >
                         if N > 15 or N' > 15 .

  op initState : -> Configuration .  *** Initial state
  eq initState =
         < "Peter" : Person | age : 38, status : single >
         < "Lizzie" : Person | age : 35, status : single >
         < "Sam the Snake" : Person | age : 40, status : single > .
endom)

(frew [10] initState .)
```

Assuming that the above file is called `oo-population.maude` it can be executed as follows:

```
UNIX>maude oo-population.maude
...
Introduced module POPULATION

rewrites: 556 in 10ms cpu (10ms real) (55600 rewrites/second)
frewrite in POPULATION :
  initState
result Configuration :
  < "Lizzie" : Person | age : 35, status : engaged("Peter") >
  < "Peter" : Person | age : 40, status : engaged("Lizzie") >
  < "Sam the Snake" : Person | age : 43, status : single >
```

Exercise 146 *Complete the above object-oriented module* `POPULATION` *with rules for birth, marriage, separation, divorce, and for attempting reconciliation in a separated couple. Avoid superfluous attributes. Execute your specification in Full Maude.*

7.2.4 Subclasses

Full Maude supports *multiple inheritance* with the use of *subclasses*. If C is a class declared

```
class C | att_1 : s_1, ..., att_n : s_n .
```

and B is a class declared

```
class B | att'_1 : s'_1, ..., att'_k : s'_k .
```

we may declare that B is a subclass of C as follows:

```
subclass B < C .
```

Just as for subsorts, where `subsort Ape < Animal` means that every `Ape` is also an `Animal` and "inherits" all properties and functionalities of an animal, so `subclass` B `<` C means that every B-object is also a C-object which inherits all the attributes and all functionality of the class C. That is, just like an equation $f(x) = g(x)$ for `var` x `: Animal` also applies to apes, so will any rule

```
rl [l] : < o : C | ... > => < o : C | ... >
```

also apply to B-objects, whose set of attributes will be $att_1, \ldots, att_n, att'_1, \ldots, att'_k$.

Full Maude supports *multiple inheritance* in which a class may be a subclass of a number of classes:

```
subclass C < C_1 ... C_n .
```

In this case, the attributes of a C-object is the union of the set of attributes in C_1 to C_n and those declared in C. The class C also inherits all rewrite rules of its superclasses. We have *repeated inheritance* in that a superclass C_i may itself be a subclass of some other class.

Example: Muslims and Christians

The baptism is counted as the *real* birth for a Christian and is supposed to be at least as big event as the actual birth. We would be interested in modeling this important event, as well as *confirmation*, in our running example. A sort `ChristianStatus` could have constants `notBapt`, `baptized`, and `confirmed`. Not all persons are Christians; some may be Muslims (whose important event is the *hajj* (the pilgrimage to Mecca)), while others may be neither Christians nor Muslims. Both Christians and Muslims are persons: they celebrate birthdays, engagements, marriages, and they separate, divorce, and die like all persons.

We assume that we have the specification `POPULATION` and would like to extend it with important religious events. There are some different modeling possibilities. In one version, the religion of a person is given at birth, while in another version, one is born without religion but can become Christian or Muslim by performing a baptism or by reading the call-for-prayer prayer to the young infant.

Version 1: Religion Given at Birth. In this version, one's religion is given at birth, or in the initial state if one is not born during the execution of the specification.

Since Christians and Muslims are persons, we define the classes `Christian` and `Muslim` as subclasses of the class `Person`:

```
sort ChristianStatus .
ops notBapt baptized confirmed : -> ChristianStatus [ctor] .

class Christian | chrStatus : ChristianStatus .
class Muslim | al-haji : Bool .
subclass Christian Muslim < Person .
```

(The attribute `al-haji` is `true` iff a Muslim has done a *hajj*.) The rules for baptism and confirmation are fairly trivial:

```
rl [baptism] :   < X : Christian | chrStatus : notBapt >  =>
                 < X : Christian | chrStatus : baptized > .

rl [confirmation] :  < X : Christian | chrStatus : baptized >  =>
                     < X : Christian | chrStatus : confirmed > .
```

The rule for *hajj* implies that a Muslim may perform more than one *hajj* in his lifetime:

```
rl [hajj] :  < X : Muslim | >  =>  < X : Muslim | al-haji : true > .
```

Recall the rule for birth in the module `POPULATION`:

```
crl [birth] :
    < X : ListOfNames | OKnames : L X' L' >
    < X'' : Person | age : N, status : married(X''') >
    < X''' : Person | age : N', status : married(X'') >
       =>
    < X : ListOfNames | OKnames : L L' >
    < X'' : Person | >
    < X''' : Person | >
    < X' : Person | age : 0, status : single >   if N < 60 or N' < 60 .
```

In this rule the newborn is just a `Person` who will not be a follower of either of the main monotheistic religions. Since this rule also applies to Muslims and Christians, a religious couple may still get a non-religious offspring. The following rules allow the *possibility* of a child to have the religion of a parent:

```
crl [birthChristian] :
    < X : ListOfNames | OKnames : L X' L' >
    < X'' : Christian | age : N, status : married(X''') >
    < X''' : Person | age : N', status : married(X'') >
```

```
      =>
    < X : ListOfNames | OKnames : L L' >
    < X'' : Christian | >
    < X''' : Person | >
    < X' : Christian | age : 0, status : single, chrStatus : notBapt >
      if N < 60 or N' < 60 .

crl [birthMuslim] :
    < X : ListOfNames | OKnames : L X' L' >
    < X'' : Muslim | age : N, status : married(X''') >
    < X''' : Person | age : N', status : married(X'') >
      =>
    < X : ListOfNames | OKnames : L L' >
    < X'' : Muslim | >
    < X''' : Person | >
    < X' : Muslim | age : 0, status : single, al-haji : false >
      if N < 60 or N' < 60 .
```

A typical initial state of this system could be

```
< "Nice names" : ListOfNames | OKnames : "Mette" "Lizzie" >
< "Imtiaz" : Muslim | age : 39, status : married("Maiken"), al-haji : false >
< "Maiken" : Christian | age : 38, status : married("Imtiaz"),
                         chrStatus : confirmed >
< "Panchen Lama" : Person | age : 18, status : single >
```

Exercise 147 *According to the specification above, what kind of offspring (Christian? Muslim? neither?) could a couple have if*

1. *one is Muslim and the other one is Buddhist?*
2. *both are Christian?*
3. *one is Muslim and the other one is Christian?*
4. *both are Jewish?*

*** Version 2: We are all Born Equal.**

□ In another version, everyone is born a non-believer and instead has the opportunity to baptize to become Christian or to convert to Islam. There is no need for more birth-rules since only `Persons` are born. Instead we need a rule for baptism so that both non-believers and Muslims can be baptized, while Christians are already baptized and cannot be baptized again. The possibility

```
rl [baptism] : < X : Person | >  =>
               < X : Christian | chrStatus : baptized > .
```

looks promising but cannot be used since it would allow a Christian to be baptized again. How can we modify the rule so that non-believers and Muslims can be baptized while Christians can not? The simplest idea is to define two sorts `ChrObject` and `MuslimObject` and define them as follows:

```
sorts ChrObject MuslimObject .
subsorts ChrObject MuslimObject < Object .
mb (< X : Christian | >)  :  ChrObject .
mb (< X : Muslim | >)  :  MuslimObject .
```

and then define the baptism rule as follows:

```
crl [baptism] : < X : Person | age : N, status : S >  =>
                < X : Christian | age : N, status : S,
                                  chrStatus : baptized >
                   if not (< X : Person | >)  ::  ChrObject .
```

Likewise you cannot convert to Islam if you are already a Muslim:

```
crl [convertIslam] : < X : Person | age : N, status : S >  =>
                     < X : Muslim | age : N, status : S, al-haji : false >
                        if not (< X : Person | >)  ::  MuslimObject .
```

The rules `hajj` and `confirmation` are as before.

Remark: The change of class corresponds to the deletion of an object and the creation of another object with the same name with a different class. *All* the attributes of the new object must therefore be provided in the right-hand sides of class-changing rules.

Exercise 148 *The above specification allows for a* `Christian` *to convert to Islam and vice versa. How would you modify the specification to disallow that?*

□

7.2.5 * Modularity/Encapsulation

□ Object-oriented programming is usually closely connected with *encapsulation* in that the definition of a class and its methods is encapsulated in a module. Maude emphasizes ease and generality of specification and does not impose any condition on modularity. It is possible also in Maude to encapsulate each class and its methods (i.e., its rewrite rules) in a separate module. Most larger Maude specifications are written in this modular style (see e.g. [82, 19]). This style of specification is less suitable when communication is synchronous and a rule may involve objects of different classes. (In that case one can, in contrast to many other languages, encapsulate the specification of these closely cooperating classes in one module!)

Maude does not impose any modularity constraints to allow also for synchronous rules, although it would be trivial to check whether certain encapsulation constraints are satisfied in a specification. □

7.2.6 * Method Specialization

□ We are used to the concept of *method specialization* from object-oriented languages. This means that a method in a class may be redefined for subclasses. In Maude a method roughly corresponds to a rewrite rule, and a remote method call could correspond to the sending of a message.

While one could sometimes think that method specialization is useful (one could e.g. think of restricting the `marriage` rule so that only people of the same faith can get married, or that the child of two Christians should be born a Christian), Maude does not support method specialization. Instead, Maude tries to be flexible enough so that the specifier should be able to easily write specifications which achieve the desired effects. In this section I first try to illustrate some difficulties in finding a suitable semantics (meaning) of "method specialization" in Maude, and then propose some "solutions."

In general it is very difficult to state that a rule such as `birth` should *not* apply to a couple of Christians just because rule `birthChr` could be used instead. This kind of thought would lead to e.g. the rule `birthday` not being applied on young Christians since the rules `baptism` or `confirmation` could be applied instead, and the objects in these rules have lower class than the objects in rule `birthday`. The picture is also more complicated since we have freedom to have synchronous rules with multiple objects involved, which we normally don't have in the object-oriented languages we are familiar with.

It is also well known that method specialization is problematic in the context of *multiple* inheritance (which is not possible in e.g. Java) because if the subclass structure is

```
subclass A < B C < D
```

and `D` has a "method" which is refined/redefined in both the subclasses `B` and `C`, but is not further redefined in class `A`, then *which* version of the method should `A` use? In another scenario, only `B` may redefine the method in `D`. Which version should `A` use?

If a message is seen as a remote method call, one could think of "method specialization" as being that the message-reading rule with the lowest appropriate class takes care of the message. While this approach may sound fairly natural, there are some cases where it seems unnecessarily restrictive. Consider for example a message `m(o)` sent to objects of a class `C` with subclasses `C1` to `Cn`. Consider furthermore that `m` could be ignored *or* could be treated. Then a rule

```
rl [ignore] : m(X) < X : C | >  => < X : C | > .
```

should intuitively also be applicable to subclasses even though they have other rules dealing with the message when it is not ignored. Again this example tried to illustrate that it is not always natural to assume that only "lower class" rules should treat messages, and adding such a constraint would unduly restrict the specification language. Furthermore, all kinds of artificial restrictions would not only restrict the specification language but would also make it more complicated as one needs to know what constraints are assumed and exactly how Maude works. In Maude's view, the purer and simpler the semantics of a specification, the better.

Meseguer argued in [70] that method specialization which entails completely redefining the method in "subclasses" should be seen as *module* inheritance instead of *class* inheritance. That is, the new class is not really a subclass but a *redefinition* of another class. Consider

```
(omod C is
  class C | ... .
  msg M ... .
endom)
```

and assume that we want a class `B` which behaves as `C` but redefines its behavior on the treatment of `M` and may add some new rules and attributes. Such a class `B` can be obtained by letting it be some renaming (and something else?) of `C`, and by removing the rules treating the message `M`:

```
(omod B is
  including C[rename(C) and fix to B, remove M-rules] .
  ...
  rl M < X : B | ... > => ...
endom)
```

Full Maude does not provide primitives for removing rules from an imported module, as suggested in [70].

Another way of achieving specialization has been indicated above and consists of manually checking the class of an object and not applying a rule if it has a certain class. For example, in the setting

```
class C | ...
class B | ...
class A | ...
subclass A B < C .

rl [l1] :  < X : C | ... >  =>  < X : C | ... > .
rl [l2] :  < X : B | ... >  =>  < X : B | ... > .
```

if one never wants rule `l1` to apply to `B`-objects one could define a sort `Bobject` of `B`-objects as follows:

```
sort Bobject .
subsort Bobject < Object .
mb (< X : B | >)  :  Bobject .
```

and add a condition

```
if not (< X : C | >) :: Bobject
```

to rule `l1`. This setting has the disadvantage that the *superclass* must know and define for what subclasses the rule should not apply.

A similar solution which avoids the problem of the superclass needing to know about the desires of the subclass is a generalized version, in which for each rule `l` there is a sort `Overrides-l-object` which denotes all the objects for whom the rule `l` should not be used. Each rule should then have a condition checking that the object participating in the rewrite is not of this sort, such as for our example:

```
sort Overrides-l1-object .
subsort Overrides-l1-object < Object .
crl [l1] :  < X : C | ... >  =>  < X : C | ... >
                if not (< X : C | >) :: Overrides-l1-object .
```

A subclass `B` of `C` can later decide that rule `l1` should not apply to a `B`-object by declaring

```
mb (< X : B | >) : Overrides-l1-object .
```

While a logically clean solution, it requires too many conditions and sorts, since each rule in the specification must have its own sort and condition. The solution could be to introduce language constructs such as

```
rl [l2] :  < X : B | ... >  =>  < X : B | ... >  overrides l1 .
```

with the semantics described above. (Alternatively, the semantics could be that `overrides` means that, on a certain term, first rule `l2` is tried, and if doesn't rewrite, then rule `l1` is tried.)

To summarize our discussion, it is not obvious what "method specialization" is supposed to mean in the very general framework of Maude. Instead of introducing strange restrictions and language constructs with possibly dubious semantics, Maude ignores the whole problem to retain maximal generality and clarity of the specification language. □

7.2.7 Search in Full Maude

After a recent update of Full Maude, a search pattern < *o* : *C* | *att* : *pattern* > in an object-oriented system will match any object of class *C* *or of a proper subclass of C* whose attribute *att* is matched by *pattern*. Therefore, we need not worry about subclasses and about explicitly mentioning all the attributes in the search pattern. This can be seen from the "echo" of the search command:

```
Maude> (search [1]
>            < "JR" : Person | age : 50, status : married("Sue Ellen") >
>            < "Sue Ellen" : Person | age : 46, status : married("JR") >
>          =>*
>            < "JR" : Person | status : single >
>            < "Sue Ellen" : Person | status : single > .)

rewrites: 3511 in 0ms cpu (43ms real) (~ rewrites/second)
search [1] in GOOD-SEPARATION :
  < "JR" : Person | age : 50, status : married("Sue Ellen") >
  < "Sue Ellen" : Person | age : 46, status : married("JR") >
 =>*
  < "JR" : V#0:Person | status : single, V#1:AttributeSet >
  < "Sue Ellen" : V#2:Person | status : single, V#3:AttributeSet > .

Solution 1
V#0:Person --> Person ; V#1:AttributeSet --> age : 50 ;
V#2:Person --> Person ; V#3:AttributeSet --> age : 46
```

The command echo shows that Full Maude replaces the class names with variables (`V#0` and `V#2` above) that can be used to capture objects belonging to subclasses of the class *C* (`Person` in the above example). Likewise, the "remaining" attributes of each object are captured by variables (`V#1` and `V#3` above) of the sort `AttributeSet`. The search result shows that the (least) class of the two objects is `Person`.

Warning: When I search in Full Maude, these features sometimes do not work, so that I have to add the variables of sort `AttributeSet` to the search pattern. This is weird as this feature works in most of the searches ...

No Search Paths

The `show path` commands are not implemented in Full Maude. The next section explains how a Full Maude module can be converted into a core Maude module that can be executed in core Maude, which of course supports the `show path` commands.

7.2.8 Overcoming Full Maude Problems

Full Maude is a *prototype* of Maude's future object-oriented and parameterization capabilities and does not work nearly as well as (core) Maude itself. Minor Full Maude deficiencies include:

- Syntactic errors are usually difficult to locate since Full Maude's error messages are not too informative—an error is often indicated not by an error message but rather by the lack of output from Full Maude. Not even an `Introduced module ...` receipt is a guarantee that the module parsed. This makes the specification effort somewhat frustrating.
- Variables in search commands *with* `such that`-conditions need to be written in their "explicit" form *var* : *sort*:

```
(search [2] < "Lizzie" : Person | age : 35, status : single > =>*
            < "Lizzie" : Person | age : N:Nat, status : single >
            such that N:Nat > 35 .)
```

- Maude's `show path` and debugging facilities don't work in Full Maude.
- As mentioned above, I have experienced that sometimes Full Maude does not automatically add the variables of sort `AttributeSet` to the objects in the search pattern.

This section presents some basic techniques for making the specification and analysis of object-oriented modules less frustrating.

Obtaining Syntactically Correct Modules

The straight-forward way of locating syntax errors in a specification is just to use `***(` and `)` to comment out large chunks of the code to see if the rest parses. Use the command `(show all .)` to make sure that Full Maude parsed your module. Repeated use of this low-level technique will (hopefully) help you localize the problems.

Obtaining the Search Path

To obtain the path leading to a state found during a search, we must transform the Full Maude specification into an equivalent (core) Maude specification. This is done by the Full Maude command

```
(show all .)
```

which outputs the (core) Maude version of the current module. One can then cut-and-paste the output from this command into (core) Maude and perform the search in (core) Maude:

```
Maude> (show all .)

mod POPULATION is
  sorts Attribute AttributeSet Bool Char Cid Configuration FindResult
    Msg Nat NzNat Object Oid Person Status String Zero .
  subsort Attribute < AttributeSet .
   ...
  crl < X:String : V#0:Person | status : V#1:Status,age : N:Nat,none,
    V#2:AttributeSet >
```

```
    => < X:String : V#0:Person | age :(N:Nat + 1),status : V#1:Status,
    V#2:AttributeSet >
    if N:Nat < 999 = true [label birthday] .
endm

Maude> mod POPULATION is
>  sorts Attribute AttributeSet Bool Char Cid Configuration FindResult
>    Msg Nat NzNat Object Oid Person Status String Zero .
>  subsort Attribute < AttributeSet .
>      ...
>  crl < X:String : V#0:Person | status : V#1:Status,age : N:Nat,none,
>    V#2:AttributeSet >
>    => < X:String : V#0:Person | age :(N:Nat + 1),status : V#1:Status,
>    V#2:AttributeSet >
>    if N:Nat < 999 = true [label birthday] .
> endm
Maude> search [2] < "Lizzie" : Person | age : 30, status : single > =>*
>                 < "Lizzie" : Person | age : N:Nat, status : single >
>                 such that N:Nat > 35 .

Solution 1 (state 6)
N:Nat --> 36

Solution 2 (state 7)
N:Nat --> 37

Maude> show path 7 .
...
```

Repeated cutting-and-pasting may be hard on your wrist so you may want to specify your Full Maude module in a file, say `file.maude`, which ends with the lines

```
(show all .)
q
```

The UNIX command

```
UNIX> maude file.maude > core-maude-file.maude
```

will then write the equivalent (core) Maude module to the file `core-maude-file.maude`. Remove the welcome and farewell greetings from this file and enter it into core Maude. The specification can be analyzed using all of Maude's features.

7.2.9 Using Full Maude: Repetition

Some things to remember when using Full Maude:

- Use Full Maude for specifying object-oriented systems.

- When Full Maude is active, input to Full Maude is enclosed by a pair of parentheses. Input that is not enclosed in such a way is input to (core) Maude.
- *Each* module given to Full Maude must be enclosed by a pair of parentheses, regardless of whether the module is given in a file or directly on the Maude command line.
- Commands such as `red`, `rew`, and `search` should likewise be enclosed by a pair of parentheses.
- The commands `in` and `load` should be treated by (core) Maude and should *not* be enclosed by parentheses. However, the file which is read may of course contain input to Full Maude; each of these should be enclosed by parentheses in the file.
- Many Maude commands and features—such as the debugger and the `show path` command—are not available in Full Maude. See the Maude manual for details.
- Load the file `full-maude.maude` to activate Full Maude.

7.3 Example: The Dining Philosophers

The *dining philosophers problem* [25] is a classic example due to Dijkstra which is used to illustrate some concepts in distributed systems where the components need to access shared resources such as, say, printers or shared memory. While this example is usually analyzed "manually," we will analyze it entirely by executing it in Maude.

7.3.1 Problem Description

Five philosophers sit around a round table with an enormous bowl filled with delicious dumplings in the middle of the table. Each philosopher spends her life alternating between thinking, then being hungry, then eating, then thinking again, and so on in a never-ending cycle. However, not even this seemingly idyllic setting is perfect: By a quirk of fate there are only *five* chopsticks on the table, one chopstick between each neighboring pair of philosophers, as seen in Fig. 7.1. We all know that dumplings are delicious but hot and slick, so a philosopher needs *both* her left and her right chopstick to eat.

A hungry philosopher will first try to grab a (left or right) chopstick when one is available, and will then hold on to this stick until she can grab her other chopstick and begin to eat. No philosopher can eat forever, so after a finite time of eating, an eating philosopher must put both chopsticks back on their places, and start thinking.

There are some intriguing questions about this world. Is it e.g. possible that all philosophers will starve to death due to lack of available chopsticks? Is it possible that *one* philosopher will starve to death while the others are feasting until the end of time?

7.3.2 Modeling the Dining Philosophers

This section presents an object-oriented model which specifies all possible behaviors of the philosophers system.

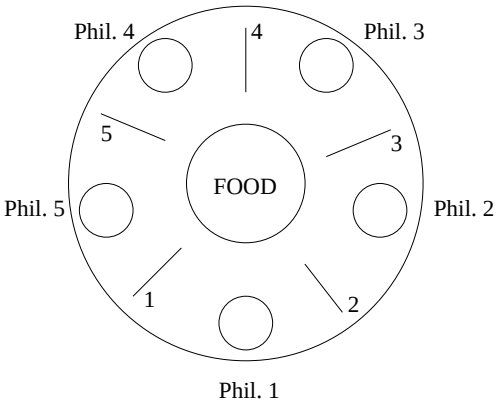

Figure 7.1: The table setting for the dining philosophers.

Modeling the Chopsticks

Chopstick number i could be represented either by an *object*

```
< i : Chopstick | >,
```

by a *message* `chopstick(`i`)`, or by a special entity `chopstick(`i`)` declared

```
sort Chopstick .
op chopstick : Nat -> Chopstick [ctor] .
subsort Chopstick < Configuration .
```

The subsort declaration declares that a `Configuration` is a multiset of objects, messages, *and chopsticks*. While this author may be inclined to believe that this last setting is the most elegant and logical, we will stick to the chopstick-as-message setting.

A "message" `chopstick(`i`)` means that chopstick i is available, and can be seen as a message which can be read and consumed by a philosopher, which then "has" the chopstick. When the philosopher stops eating she sends two `chopstick` messages into the configuration, making the chopsticks available again. The only difference between this use of messages and the normal use of messages is that normally the recipient of each message is a unique object, while in our case both philosopher i and her left neighbor can read the message `chopstick(`i`)`. Chopsticks are defined as follows:

```
msg chopstick : Nat -> Msg .
```

Philosophers

Each philosopher should be an object with an attribute denoting the current state (thinking? hungry? eating?) of the philosopher and an attribute storing the number of chopsticks currently in the philosopher's hands. Although it should *not* really be part of the specification of the philosophers, we add for analysis purposes a counter `noOfEats` to record how many times the philosopher has eaten. A philosopher object is therefore a term

```
< i : Philosopher | state : s, noOfSticks : j, noOfEats : k >
```

where i denotes the number of the philosopher. The philosopher class is declared as follows:

```
class Philosopher | state : State, noOfSticks : Nat, noOfEats : Nat .

subsort Nat < Oid .    *** Object names are numbers!

sort State .
ops thinking hungry eating : -> State [ctor] .
```

The Rewrite Rules

Each philosopher starts in a thinking state without a chopstick in hand. The first rule models the philosopher becoming hungry:

```
vars I J K : Nat .

rl [hungry] :  < I : Philosopher | state : thinking >  =>
               < I : Philosopher | state : hungry > .
```

The next rule models the philosopher grabbing her first chopstick, which could be either her left or her right chopstick:

```
crl [grabFirst] : chopstick(J)
                  < I : Philosopher | state : hungry, noOfSticks : 0 >  =>
                  < I : Philosopher | state : hungry, noOfSticks : 1 >
                     if I can use stick J .

op right : Nat -> Nat . *** The ``right'' chopstick index.
eq right(I) = if I == 5 then 1 else I + 1 fi .
op _can`use`stick_ : Nat Nat -> Bool .
eq I can use stick J = (I == J) or (J == right(I)) .
```

A philosopher can start eating when she grabs her second chopstick:

```
crl [grabSecond] : chopstick(J)
                   < I : Philosopher | noOfSticks : 1, noOfEats : K >  =>
                   < I : Philosopher | state : eating, noOfSticks : 2,
                                       noOfEats : K + 1 >
                      if I can use stick J .
```

The final rule stops the eating and puts the chopsticks back on the table:

```
rl [stopEating] :  < I : Philosopher | state : eating >  =>
                   < I : Philosopher | state : thinking, noOfSticks : 0 >
                   chopstick(I)  chopstick(right(I)) .
```

The initial state is always the same and can be declared as follows:

```
op initState : -> Configuration .
eq initState =
    chopstick(1)  chopstick(2)  chopstick(3)  chopstick(4)  chopstick(5)
    < 1 : Philosopher | state : thinking, noOfSticks : 0, noOfEats : 0 >
    < 2 : Philosopher | state : thinking, noOfSticks : 0, noOfEats : 0 >
    < 3 : Philosopher | state : thinking, noOfSticks : 0, noOfEats : 0 >
    < 4 : Philosopher | state : thinking, noOfSticks : 0, noOfEats : 0 >
    < 5 : Philosopher | state : thinking, noOfSticks : 0, noOfEats : 0 > .
```

7.3.3 Deadlock and Livelock

A distributed system where processes need exclusive access to shared resources may *deadlock*. This means that the system is stuck and nothing can happen in the system because no process can proceed until it gets a shared resource which is controlled by another process (which is equally stuck, since it may need e.g. some resource controlled by the first process). A possible deadlock state here could be a state where each philosopher has one chopstick each, and cannot do anything because there are no chopsticks available.

Livelock (also known as *starvation*) is a trickier property which means that *one* philosopher could starve to death because she can never get hold of both chopsticks, while at the same time the other philosophers could feast merrily.

Exercise 149

1. *Execute the dining philosophers system using Full Maude's* `rew` *and* `frew` *commands. Do all philosophers get to eat sufficiently often?*
2. *Use Full Maude's search command to show that there could be a deadlock in the system.*
3. *Show a scenario (a "run") which results in a* deadlock.
4. *Use Maude's search command to check whether there is a state in which* each *philosopher has eaten at least* twice. *Repeat the search for* tree *meals per philosopher.*
5. *Show a desired scenario in which each philosopher gets to eat infinitely many times.*
6. *Does the system allow* starvation*? That is, use Maude's (or Full Maude's) search command to check whether there is a behavior in which one philosopher has yet to eat, while at least three other philosophers have eaten at least two times each.*
7. *What is the maximum number of events (i.e., rule applications) that could happen in a concurrent rewrite step?*
8. *Can a philosopher grab both "her" chopsticks in one concurrent rewrite step? Why/why not? Does this make sense?*
9. *Can two philosophers grab the same chopstick in a concurrent step?*

10. *Sometimes the dining philosophers setting is described so that each philosopher grabs her* left *chopstick first. Can you modify the above specification to accommodate for that setting? Can the system still contain deadlocks?*

7.3.4 * The Specification is Incorrect

□ The Full Maude specification given above is an incorrect model of the philosophers setting, since the informal description says that each eating philosopher must *eventually* put down her chopstick. Similarly, no philosopher can be in a thinking state forever. These *fairness* constraints are not captured in the Maude model where one philosopher may never stop eating, and another philosopher may never stop thinking.

However, each *finite* behavior (simulated with a `rew [`n`]` command) is "correct" in that it is always a *prefix* of a behavior in which no philosopher eats continuously. Therefore, this deficiency doesn't affect the reasoning about deadlocks. However, a livelock (starvation) is an infinite scenario, so that one must take care that the livelock behaviors allowed by a specification also satisfy additional fairness constraints.

Fairness criteria which say that *eventually* (i.e., "some time in the future") something must happen cannot be "implemented" in full generality, since there is no bound on *when* that "something" must happen. Instead, many nonexecutable models of concurrency such as *fair transition systems* [65] add a fairness constraint such as "no rule can be applicable continuously (or infinitely often) without being applied." This constraint is then used when *reasoning* about "global properties" such as "eventually each philosopher will eat." All infinite runs executed by Maude are fair in this sense since a rule which is continuously applicable will eventually be applied. However, sample executions indicate that Maude's `frew` command is not very fair w.r.t. which objects get to execute. □

7.3.5 A Deadlock-Free Solution

A solution which has been proposed to avoid deadlocks is to let each philosopher grab both chopsticks at the same time (and not allow them to grab only one).

Exercise 150

1. *Specify and execute this version of the dining philosophers in Full Maude.*
2. *Use Full Maude's search command to search for a deadlock in the new specification. Explain the "result" of the search.*
3. *Explain why there cannot be a deadlock in the specification. That is, explain why some rule can always be applied.*
4. *Show that the specification is not livelock-free. That is, show that there is an infinite behavior of the system (in which all fairness criteria are satisfied) in which there is some philosopher who never has the* possibility *of grabbing chopsticks.*

7.3.6 A Deadlock- and Livelock-Free Solution

The dining philosophers could get stuck in a deadlock situation where each philosopher proudly holds her, say, right chopstick and waits for the other chopstick, which will never become

available. The solution where each philosopher grabs both chopsticks removed the possibility of deadlock, but not the possibility of livelocks.

The following solution has been proposed to avoid also livelocks: Philosophers should not meditate on the deep questions of life in the dining room, but in the adjacent library! Furthermore, there is now a doorman (or a sophisticated turnstile system) allowing no more than *four* philosophers to be in the dining room at any time. Of course, each philosopher can only sit at her designated place while in the dining room.

A state in this new setting can be an object of the form

```
< GlobalSystem : DinPhilHouse | diningRoom : philsAndSticks, noInDinRoom : n,
                                library : philosophers > .
```

where *philsAndSticks* is a `Configuration` consisting of all available chopsticks and of those philosophers who are currently in the dining room, the number n denotes the number of philosophers currently in the dining room, and *philosophers* is a `Configuration` consisting of the philosophers in the library.

In this system, we have configurations, that is, object-oriented systems, inside an object. The "global" class `DinPhilHouse` could be declared as follows:

```
class DinPhilHouse | diningRoom : Configuration, noInDinRoom : Nat,
                     library : Configuration .

op GlobalSystem : -> Oid [ctor] .
```

The philosophers and chopsticks are modeled as before, and so is the rule `hungry` which lets a thinking philosopher become a hungry philosopher (although now this transformation takes place in the library). A new rule lets a hungry philosopher enter the dining room if there are less than four philosophers in the dining room:

```
var O : Oid .  vars C C' : Configuration .

crl [enterDinRoom] :
    < O : DinPhilHouse | diningRoom : C, noInDinRoom : K,
                         library : (< I : Philosopher | state : hungry >  C') >
       =>
    < O : DinPhilHouse | diningRoom : (< I : Philosopher | >  C),
                         noInDinRoom : K + 1, library : C' >
       if K < 4 .
```

Here, the variable `C` holds the configuration consisting of the philosophers and chopsticks already in the dining room, and `C'` are the philosophers left in the library.

The rules `grabFirst` and `grabSecond` apply as before. We could be harsh and modify the `stopEating` rule so that a philosopher must leave the dining room at the moment she stops eating. A gentler versions keeps the rule `stopEating` and adds the rule

```
rl [enterLibrary] :
   < O : DinPhilHouse | diningRoom : (< I : Philosopher | state : thinking >  C),
                        noInDinRoom : s K, library : C' >
      =>
   < O : DinPhilHouse | diningRoom : C, noInDinRoom : K,
                        library : (< I : Philosopher | > C') > .
```

in which a philosopher who has started thinking leaves the dining room.

In the initial state all philosophers should be in the library thinking, while the delicious dumplings and the chopsticks are in the dining room:

```
< GlobalSystem : DinPhilHouse |
    diningRoom : chopstick(1)  chopstick(2)  chopstick(3)
                 chopstick(4)  chopstick(5),
    noInDinRoom : 0,
    library :
      (< 1 : Philosopher | state : thinking, noOfSticks : 0, noOfEats : 0 >
       < 2 : Philosopher | state : thinking, noOfSticks : 0, noOfEats : 0 >
       < 3 : Philosopher | state : thinking, noOfSticks : 0, noOfEats : 0 >
       < 4 : Philosopher | state : thinking, noOfSticks : 0, noOfEats : 0 >
       < 5 : Philosopher | state : thinking, noOfSticks : 0, noOfEats : 0 >)
>
```

Exercise 151

1. *Specify this version of the dining philosophers and execute your specification.*
2. *Show the behavior leading to the state returned by your above execution of the specification.*
3. *Can a Full Maude search find a deadlock in the specification?*
4. *Explain why there cannot be a deadlock in this specification.*
5. *Explain why there cannot be a livelock in the specification, when we assume the additional fairness constraint that each eating philosopher will leave the dining room in finite time. That is, explain that there is no scenario in which a hungry philosopher* can *never grab both chopsticks in the future.*
6. *Can two philosophers exit the dining room at the same time?*
7. *Can one philosopher enter the dining room while another philosopher is exiting it? (In both of these cases there could then be a problem with the counting.)*
8. *Can two philosophers stop eating at the same time?*
9. *Can two philosophers grab a chopstick each, another philosopher becoming hungry, and yet another philosopher leave the dining room, all in one concurrent step?*
10. *Change the specification to the harsher version where a philosopher must exit the dining room at the moment he stops eating.*

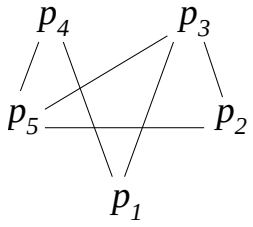

Figure 7.2: A "conflict graph" between the processes p_1 to p_5.

7.4 * The Generalized Dining Philosophers Problem

□ The goal of the dining philosophers setting is to ensure that no pair of neighbors access a shared resource (chopstick) at the same time.

In the *generalized dining philosophers problem* we are given a set of processes (philosophers) and an arbitrary set of edges between pairs of processes (shared chopsticks), where each edge represents some kind of "conflict." The ultimate goal is to let the processes operate such that two processes never access a shared resource at the same time if there is an edge between them. (We say that a process is in its "critical region" when it is using a shared resource.)

When the conflict graph is a complete graph, this goal is the same as the *mutual exclusion* which says that, at any time, at most *one* process can access the shared resource.

A process may be in either of three "states":

1. It doesn't need the shared resource at the moment.
2. It needs the shared resource to continue its work and waits for the shared resource to become available.
3. It is in its "critical region" accessing the shared resource.

For example, in Fig. 7.2 processes p_1 and p_2 may access the shared resource at the same time, while e.g. p_5 and p_2 should never access the shared resource at the same time.

Exercise 152 *Specify the generalized dining philosophers problem in Full Maude and make sure that conflicting pairs of processes never access the shared resource at the same time.*

This setting can be modeled using object-oriented techniques, by letting each process be modeled by an object which can be in either of the states $computingElsewhere$, $waitingForResource$, *and* $accessingSharedResource$, *so that no conflicting pair of processes are in the state* $accessingSharedResource$ *at the same time.*

□

Chapter 8

Modeling Communication and Networks in Maude

A distributed system can naturally be modeled as a system of *communicating* concurrent objects. This chapter suggests how various forms of communication and communication networks can be modeled in Maude in an object-oriented style.

We need to be able to appropriately model a wide range of different forms and features of communication, because

1. there are different kinds of communication devices with different communication capabilities;
2. we need to able to model systems at different *levels of abstraction* so that unnecessary details are omitted;
3. of generality: a protocol may be applicable not only to one kind of system, but to all classes of systems that satisfy given assumptions about their communication features.

Examples of different communication devices include an "ordinary" computer communicating using TCP/IP, a satellite broadcasting TV shows to a large number of households, and a sensor node in a wireless sensor network. These three devices have very different communication capabilities. In addition, a single system may contain several different kinds of communication devices.

To make any modeling and analysis task of communicating systems feasible, it is imperative to omit as much detail as possible from the parts which do not influence the *functionality* of the system. When analyzing "high-level" communication protocols it is not necessary to know how messages are divided into packets (or "frames"), to know what are the "header" fields in a packet, to know exactly through which links a packet is routed, etc. Including these aspects into the model just clutters the protocol, and would make its analysis much harder—for no gain. Instead, the model should *abstract* from those details and model only the essential aspects of the protocol. The desired *level of abstraction* changes from application to application. Some protocols are more "high level" where we only need to be concerned about whether a packet is lost, while more details must be modeled in e.g. "lower level"

routing protocols. In these lecture notes we consider communication at a fairly high level of abstraction, and abstract from details about how communication is actually achieved. On the other hand, in the modeling of the AER/NCA multicast protocol in (Real-Time) Maude [82], details, such as the speed of the individual communication links, their transmission delays, and their capacities, were crucial for the functionality of the protocol, and thus had to be carefully modeled.

Finally, a protocol for distributed systems may be "general" in that it is intended not only for systems with a certain kind of communication, but may be intended for *all* systems where communication satisfies certain criteria, such as, e.g., "messages are not lost."

There are many forms of communication. Communication may be *synchronous* in that objects *synchronize* in the communication event, such as when two people talk to each other. Communication may be *asynchronous* in that the parties do *not* synchronize to communicate. Examples of asynchronous communication include communication by sending letters using the postal service, by email, by leaving messages on the voice mail, and by writing on and reading a shared message board.

Asynchronous communication may be *ordered*, so that the recipient of a set of messages/letters from the same sender reads the messages in the order in which they were sent, or asynchronous communication may be *unordered*, in which the recipient may read a sequence of messages in a different order than they were sent. Examples of ordered delivery include messages left on an answering machine and communication by messages sent along the same cable or link. Examples of unordered delivery include email sent on the Internet, letters sent in the mail, and a communication setting with different forms of communication, such as in the communication between a bank and a customer in which the customer may send both ATM "purchase" messages *and* checks in the mail to pay bills.

A communication event may involve *two* parties, such as a person writing a letter to a loved one (*unicast*), or may involve *many* parties, such as a party sending junk emails to millions of computer users (*multicast*), or a satellite sending the same pictures to all the households with an appropriate satellite dish (*broadcast*).

We also need to model the communication network (or *topology*). Communication "channels" may go "both ways," such as a phone conversation between two persons, or may go only "one way," such as the communication between a broadcasting satellite and your TV set, or such as the communication between a monstrous B-52 dropping propaganda leaflets and the people on the ground who are supposed to be fooled by such propaganda. The communication topology may be *static*, or it may *dynamic*, in that new "channels" may be added and older channels may be deleted. Examples of channel additions may include a person starting a subscription to pay-TV (which I had to do in order to watch the 2002 soccer World Cup on TV) and the addition of a computer to the Internet. An example of deletion of communication channels includes unsubscribing to the pay-TV after the World Cup.

Another aspect of communication is whether it is *reliable*. Communication may be unreliable, such as when email or letters are lost or misplaced, or when the content of a data packet is corrupted. Other forms of communication are reliable, such as (hopefully) the co-called hot line between "Washington" and "Kreml" durinf the cold war. Communication may also be both reliable and unreliable, such as a *data link* which delivers data reliably unless it is full.

This proliferation of forms and aspects of communication presents a challenge for modeling

formalisms. Most of the high-level modeling formalisms for distributed systems, such as Actors [1], process algebras [74, 48], I/O automata [64], and Petri nets [87], provide a *fixed* communication model. For instance, most process algebras provide primitives for synchronous communication, while the Actor model supports asynchronous communication. If the user needs a different kind of communication, this must be encoded using the given communication primitive(s). Maude, in contrast, does *not* provide any fixed communication primitive. Instead, many forms of communication can be easily and naturally modeled directly in rewriting logic. This gives the specifier the flexibility to define all kinds of communication in Maude—at the desired level of abstraction—without having to trickily encode the appropriate form of communication using some fixed communication construct.

This chapter gives some *suggestions* of how different forms and features of communication can be modeled in Maude, so that the reader gets an idea of how she can specify her own communicating system. Section 8.1 discusses high-level modeling of *synchronous* communication. The rest of the chapter deals with *asynchronous* communication, mostly through message passing. We separate between the cases where the underlying communication medium can be seen as providing, respectively, *unordered* and *ordered* message delivery. In Section 8.2 we present the fundamental message passing model for unicast unordered message transmission, and show how even the seemingly trivial task of getting a *separation* from your spouse is complicated when done by message exchange. This example is supposed to give some flavor of the intrinsic difficulty in understanding asynchronously communicating systems—underscoring the necessity of being able to analyze such systems. Section 8.3 suggests modeling *ordered* asynchronous communication by using explicit *link* (or *channel*) objects, through which messages between pairs of nodes are transmitted. For each of these fundamental models for ordered and unordered unicast, we show how to extend it to *multicast*, i.e., sending a message to a *set* of receivers, and (for unordered communication only) *broadcast*, i.e., sending a message to all other nodes in the system. We then add features to our basic models for modeling *loss* and *duplication* of messages, and to model links with limited capacities.

Section **??** proposes a way of modeling asynchronous communication through *shared variables*.

protocols/examples

8.1 Synchronous Communication

Synchronous ("handshake") communication, in which objects (which we use to model the "nodes" in a communication network) *synchronize* (or "meet") to perform a communication event together, is modeled in Maude by having all the objects in the communication event in the left-hand side of a rule, such as in the rule

```
crl [engagement] :  < X : Person | age : N, status : single >
                    < X' : Person | age : N', status : single >
                   =>
                    < X : Person | status : engaged(X') >
                    < X' : Person | status : engaged(X) >
                   if N > 15 /\ N' > 15 .
```

in which two parties communicate their mutual desire to marry. Objects can be seen as "swimming" in the global "soup" which makes up the state, and can meet to perform the rule, and can then drift away, as explained on page 189. More than two objects may of course participate in a communication event, in which case all the objects must be in the left-hand side of the corresponding rule.

Exercise 165 *A marriage needs two witnesses to be legal. Modify the* `marriage` *rule so that also two witnesses "are present" in a marriage. Neither the age nor the civil status of a witness is changed. Is the resulting specification* really *different from the original one, assuming that the initial state always has at least four objects? (*Hint: *look at the possible concurrent steps.)*

8.2 Asynchronous Communication through Unordered Message Passing

In this section we describe different ways of modeling *unordered asynchronous communication* between objects by message passing. We will consider reliable and unreliable communication, object-to-object communication (*unicast*) as well as various forms of *multicast* (one-to-many communication) and *broadcast* (one-to-all communication).

8.2.1 Unordered Unicast Message Passing

Unordered delivery is a natural model of many forms of asynchronous communication, including email transmission and communication by sending letters in the mail. An email sent from A to B may well arrive later than another email from A to B which was sent later. The same is the case with letters sent through the postal service. One difficulty in designing distributed systems is that one has to allow for *any* possible order of message delivery, and that one may not know whether a message is delayed or is lost.[1]

The usual way of modeling unordered asynchronous communication in Maude is by message passing. A message m is sent by adding it to the global state, i.e., the message m appears only in the *right-hand* side of a rewrite rule. A rewrite rule in which a message m occurs in the left-hand side of the rule models the *consumption* of the message m.

A system in which an object `A` wants to send two messages `m1` and `m2` to an object `B` could have the form

```
rl [send-m1] :    < A ... >  =>  < A ... > m1 .
rl [send-m2] :    < A ... >  =>  < A ... > m2 .
rl [read_m1] : m1 < B ... >  =>  < B ... > .
rl [read_m2] : m1 < B ... >  =>  < B ... > .
```

[1]This author is still waiting for 20 kg of books delivered to the Palo Alto post office in December 2000. They were supposed to arrive within about 3 months. Were they ever sent from the post office? Will they arrive soon, or are they lost forever? Furthermore, he has already received 40 kg of books sent from Urbana, Ill., in July 2004.

We observe that, since a configuration is a *multiset* of objects and messages, the order of the elements does not matter. Therefore, the message `m2` may be consumed before message `m1` is consumed, even if `m2` was sent after `m1`:

```
< A ... > < B ... >  ⟶  < A ... > m1 < B ... >  ⟶
< A ... > m2 m1 < B ... >  ≡ (AC)  < A ... > m1 m2 < B ... >  ⟶
< A ... > m1 < B ... >  ⟶  < A ... > < B ... > .
```

Example: Arranging a Separation Through Messages

To illustrate asynchronous communication—and its difficulties—we will look at the seemingly trivial task of arranging a *separation*, and then a divorce, through asynchronous message passing. I guess that quite often a separation is initiated by a letter (from the lawyer?). To make the example as simple as possible, there is no possibility of reconciliation. It's all over if one party wants to separate.

First we define the message

```
msg separate : Oid -> Msg .
```

which models an "I want to separate"-letter. A message `separate(X)` is a message to `X` stating that `X`'s spouse wants a separation. A first, intuitive model of an attempt to separate could be the rule

```
rl [initiateSeparation] :  < X : Person | status : married(X') >
                          =>
                           < X : Person | status : separated(X') >
                           separate(X') .
```

in which the message `separate(X')` is added to the global state. The treatment of a separation request by the unsuspecting spouse is equally straight-forward:

```
rl [acceptSeparation] :   separate(X)
                          < X : Person | status : married(X') >
                         =>
                          < X : Person | status : separated(X') > .
```

A behavior of the system could then be

```
< "JR" : Person | age : 50, status : married("Sue Ellen") >
< "Cathy" : Person | age : 25, status : single >
< "Sue Ellen" : Person | age : 45, status : married("JR") >
< "Cliff" : Person | age : 46, status : single >
            ⟶
< "JR" : Person | age : 50, status : married("Sue Ellen") >
< "Cathy" : Person | age : 25, status : single >
< "Sue Ellen" : Person | age : 45, status : separated("JR") >
separate("JR")
< "Cliff" : Person | age : 46, status : single >
```

which, due to the `assoc` and `comm` attributes of the `Configuration` constructor `__`, is equivalent to

```
separate("JR")
< "JR" : Person | age : 50, status : married("Sue Ellen") >
< "Cathy" : Person | age : 25, status : single >
< "Sue Ellen" : Person | age : 45, status : separated("JR") >
< "Cliff" : Person | age : 46, status : single >
```

which can be rewritten to

```
< "JR" : Person | age : 50, status : separated("Sue Ellen") >
< "Cathy" : Person | age : 25, status : single >
< "Sue Ellen" : Person | age : 45, status : separated("JR") >
< "Cliff" : Person | age : 46, status : single >
```

In this case everything went well so that `"JR"` and `"Sue Ellen"` separated successfully.

Notice that other actions may happen between the initiation and the end of the separation process:

```
separate("JR")
< "JR" : Person | age : 50, status : married("Sue Ellen") >
< "Cathy" : Person | age : 25, status : single >
< "Sue Ellen" : Person | age : 45, status : separated("JR") >
< "Cliff" : Person | age : 46, status : single >
          birthday
            ⟶
separate("JR")
< "JR" : Person | age : 51, status : married("Sue Ellen") >
< "Cathy" : Person | age : 26, status : single >
< "Sue Ellen" : Person | age : 46, status : separated("JR") >
< "Cliff" : Person | age : 46, status : single >
           ⟶
< "JR" : Person | age : 51, status : separated("Sue Ellen") >
< "Cathy" : Person | age : 26, status : single >
< "Sue Ellen" : Person | age : 46, status : separated("JR") >
< "Cliff" : Person | age : 46, status : single >
```

This behavior models a process in which it takes so long for the `separation` message to arrive so that three of the persons could celebrate their birthdays in-between (possibly in one concurrent step). This is expected and desired from the model, given both my experience with postal delivery and the fact that the `separate` message could have been mailed the day before the three birthdays.

Unfortunately, the specification is incorrect since an *old* separation may destroy a *new* marriage! Assume namely that *both* `"JR"` *and* `"Sue Ellen"` figure out more or less at the same time that they want to separate:

```
< "JR" : Person | age : 50, status : married("Sue Ellen") >
< "Cathy" : Person | age : 25, status : single >
< "Sue Ellen" : Person | age : 45, status : married("JR") >
< "Cliff" : Person | age : 46, status : single >
```

```
                ⟶
< "JR" : Person | age : 50, status : married("Sue Ellen") >
< "Cathy" : Person | age : 25, status : single >
< "Sue Ellen" : Person | age : 45, status : separated("JR") >
separate("JR")
< "Cliff" : Person | age : 46, status : single >
                ⟶
< "JR" : Person | age : 50, status : separated("Sue Ellen") >
separate("Sue Ellen")
< "Cathy" : Person | age : 25, status : single >
< "Sue Ellen" : Person | age : 45, status : separated("JR") >
separate("JR")
< "Cliff" : Person | age : 46, status : single >
```

In this behavior, both `"JR"` and `"Sue Ellen"` felt the bad vibes, sent the separation requests, and went into state `separated`. Two separated people can then divorce (the rules for "asynchronous divorce" are not yet given, but we could use the synchronous divorce rule[2]):

```
< "JR" : Person | age : 50, status : separated("Sue Ellen") >
separate("Sue Ellen")
< "Cathy" : Person | age : 25, status : single >
< "Sue Ellen" : Person | age : 45, status : separated("JR") >
separate("JR")
< "Cliff" : Person | age : 46, status : single >
                ⟶
< "JR" : Person | age : 50, status : single >
separated("Sue Ellen")
< "Cathy" : Person | age : 25, status : single >
< "Sue Ellen" : Person | age : 45, status : single >
separate("JR")
< "Cliff" : Person | age : 46, status : single >
```

Now, `"JR"` is again single and does not waste his time and starts courting `"Cathy"`, and eventually marries her. Likewise, `"Sue Ellen"` goes on and marries `"Cliff"`, leading us to the state

```
< "JR" : Person | age : 50, status : married("Cathy") >
separate("Sue Ellen")
< "Cathy" : Person | age : 25, status : married("JR") >
< "Sue Ellen" : Person | age : 45, status : married("Cliff") >
separate("JR")
< "Cliff" : Person | age : 46, status : married("Sue Ellen") >
```

Now disaster strikes ... `"JR"` reads the `separate("JR")` message (sent by `"Sue Ellen"`!) which has been lying around, and thinks that `"Cathy"` wants a separation:

```
< "JR" : Person | age : 50, status : separated("Cathy") >
separate("Sue Ellen")
< "Cathy" : Person | age : 25, status : married("JR") >
< "Sue Ellen" : Person | age : 45, status : married("Cliff") >
< "Cliff" : Person | age : 46, status : married("Sue Ellen") >
```

[2]There is no contradiction in using the synchronous divorce rule, since the parties don't talk to each other and are therefore not aware of the fact that *both* of them have initiated a separation when they meet in court.

Finally, "Sue Ellen"—now happily married to "Cliff"—reads the old separation message from "JR" and separates:

```
< "JR" : Person | age : 50, status : separated("Cathy") >
< "Cathy" : Person | age : 25, status : married("JR") >
< "Sue Ellen" : Person | age : 45, status : separated("Cliff") >
< "Cliff" : Person | age : 46, status : married("Sue Ellen") >
```

Two happy marriages have been broken up by old `separate` messages! The problems are that

1. a `separate` message from the spouse is not treated if you are in a state `separated` (you don't check your mailbox for `separate` messages once you feel separated), and
2. an old `separate` message can arrive a couple of years later, destroying a new and happy marriage (this could well happen, considering my still-delayed packet sent in 2000).

The first of the above problems could be amended by adding a rule

```
rl [sep2] :  separate(X)
              < X : Person | status : separated(X') >
             =>
              < X : Person | > .
```

Adding this rule does not solve the second problem, since the unfortunate behavior outlined above is still possible also in the extended system. (The reader may suggest to add a sender-parameter to the `separate` message, so that `separate("JR", "Sue Ellen")` would be a message from "JR" to "Sue Ellen". This would *not* solve all our problems, since it could happen that "JR" and "Sue Ellen" would remarry after their first divorce (as I think happened in the excellent TV show *Dallas*). In that case, the *old* `separate` message would be devastating to their *new* marriage. (One could of course add a *time* component and stamp the message with the current date, but then we are dealing with *real-time* systems which is beyond the scope of this course.))

The intention in this course is to use Maude as much as possible to analyze communicating systems, so, instead of manually analyzing the above specification, we find these errors using Maude's search function:

Exercise 166 *Extend your specification of a population with the above rules for asynchronous separation (including the rule* `sep2`*), so that it includes (synchronous rules) for divorce, engagement, marriage etc. (You may also download the specification from the course web site.)*

1. *Use Maude's search capabilities to show that a married couple can turn into a couple in which one of the person is married to the same spouse, while the other spouse is separated, and there is no pending (unread)* `separate` *message in the system. (This case corresponds to the case of* "JR" *and* "Sue Ellen" *remarrying and then one of them discovers the old* `separation` *message.)*

2. *Use Maude's search capabilities to show that, starting from a normal state in which* `"JR"` *and* `"Sue Ellen"` *are married and* `"Cathy"` *is single, it is possible to reach a state in which* `"JR"` *is separated from* `"Cathy"`, `"Cathy"` *is married to* `"JR"`, *and there is no message pending. (This case corresponds to the first bad case described above.)*

Is it *impossible* to separate? I think that the following solution is the simplest. We first add a new status

```
op waitSep : Oid -> Status [ctor] .
```

which states that a separation has been initiated and that the person is waiting for the answer. The rules defining the *protocol* for how to separate are

```
rl [initSep] :  < X : Person | status : married(X') >
               =>
                < X : Person | status : waitSep(X') >
                separate(X') .

rl [acceptSep] :  separate(X)
                   < X : Person | status : married(X') >
                  =>
                   < X : Person | status : separated(X') >
                   separate(X') .

rl [acceptSep2] :  separate(X)
                    < X : Person | status : waitSep(X') >
                   =>
                    < X : Person | status : separated(X') > .
```

While I *think* that the above solution is the least complicated correct solution to the separation problem, it is not entirely intuitive. To further analyze the separation protocol one could start with a state

```
< "JR" : Person | age : 50, status : married("Sue Ellen") >
< "Sue Ellen" : Person | age : 46, status : married("JR") >
< "Cathy" : Person | age : 25, status : single >
```

and analyze all possible states reachable from that state to check whether they "look OK."

Exercise 167

1. *Repeat the searches for the bad states described in Exercise 166 in the new and hopefully correct version of the separation protocol. (*Hint: *You may want to set a lower maximal age to speed up the search.) Can you state that the protocol is correct based on these executions?*

2. *Set the age limit in the* `birthDay` *rule to, e.g.,* `25` *to reduce the search space, and find* all *states* without messages *that are reachable from the above initial state. Do they all look OK?*[3]

The specification above is often called a *protocol* which describes how the spouses should behave for a separation to be successful. Indeed, the "programs" for distributed systems are often protocols which define how the distributed components should interact.

To argue for the correctness of the separation protocol we see that *each* party must send exactly one `separate` message, and must consume exactly one `separate` message, in the separation process.

Exercise 168 *Define a protocol which describes how a separated couple should behave to divorce through asynchronous communication. Analyze the specifications by performing various searches in Maude.*

Exercise 169 *Another possible solution to the separation problem is for a person to wait for an "acknowledgment" message* `sepOK` *saying that the (ex-) partner has received the separation request:*

```
msg sepOK : Oid -> Msg .

rl [sep] :  < X : Person | status : married(X') >
             =>
            < X : Person | status : waitSep(X') >
            separate(X') .

rl [ackSep] :  separate(X)
               < X : Person | status : married(X') >
                =>
               < X : Person | status : separated(X') >
               sepOK(X') .

rl [sepOK] :  sepOK(X)
              < X : Person | status : waitSep(X') >
               =>
              < X : Person | status : separated(X') > .
```

Explain why this is an "incorrect" protocol for separation.

This example illustrates the difficulties with asynchronously communicating systems. It seems almost impossible to find a simpler example: only one communication event (a separation) should take place, and there is no loss or corruption of messages. If this extremely simple problem has such an unintuitive solution, one can imagine how difficult more complex communication protocols can be.

[3] It is of course still a job to check these 9 states manually. It is much easier to check whether a bad state is reachable by using Maude's *LTL model checker* [13].

Example: A Protocol for Ordered Asynchronous Communication using Unordered Message Passing

This section presents a fairly simple *protocol* for achieving *ordered* communication between pairs of objects when the underlying transmission medium only provides *unordered* message transmission. This task should not be confused with *modeling ordered communication*, which is treated in Section 8.3.

A protocol that ensures that messages from an object o_1 to an object o_2 are *read* in the sending order can be described as follows:

1. A *sequence number* is attached to each message.
2. The first message from o_1 to o_2 is assigned sequence number 1, the second message from o_1 to o_2 is assigned sequence 2, and so on.
3. The object o_2 must read the messages from each sender in ascending sequence number order. That is, it can only read the message from o_1 which has a sequence number that is one greater than the sequence number of the last message received from o_1.

Each object must know, for each object to which it can send messages, the sequence number of the last message sent to that object, and must know, for each object from which it can receive messages, the sequence number of the last received message from that object. Such knowledge can be modeled by sets of pairs of the form `(`o`,`n`)`, where o is an object identifier and n is a natural number:

```
(omod PAIRS is protecting NAT .
  sort OidNatPair .
  op _‘,_ : Oid Nat -> OidNatPair [ctor] .

  sort SetOfPairs .
  subsort OidNatPair < SetOfPairs .
  op noPair : -> SetOfPairs [ctor] .
  op __ : SetOfPairs SetOfPairs -> SetOfPairs [ctor assoc comm id: noPair] .
endom)
```

An object can then have two attributes, call them `sentNos` and `rcvdNos`, which keep track of the sequence number of the last message sent to/received from each object. For purposes of reusability, we can define a class

```
  class SeqNoNode | sentNos : SetOfPairs, rcvdNos : SetOfPairs .
```

so that objects which communicate by sequence numbers can be declared to be objects of subclasses of this class.

Let now `m(A, B, N)` be a message with "body" `m`, sender `A`, receiver `B`, and sequence number `N`. The rule for sending such a message should have the form

```
  var S : SetOfPairs .   var N : Nat .

  rl [send-m] :
     < A : ... | sentNos : (B, N) S, ... >
    =>
     < A : ... | sentNos : (B, s N) S, ... >
     m(A, B, s N) .
```

which ensures that the sequence number of this message is one greater than the sequence number of the previous message sent to `B`. The receiver `B` may only read the message from `A` with the expected sequence number:

```
  rl [read-m] :
     m(A, B, s N)
     < B : ... | rcvdNos : (A, N) S, ... >
    =>
     < B : ... | rcvdNos : (A, s N) S, ... > .
```

Before we continue, we present a data type of *lists of messages*, where we use the symbol `::` for list concatenation:

```
(omod MSG-LIST is
  sort MsgList .
  subsort Msg < MsgList .
  op nil : -> MsgList [ctor] .
  op _::_ : MsgList MsgList -> MsgList [ctor assoc id: nil] .
endom)
```

Exercise 170 *In this exercise we implement the sequence number protocol described above. Assume three kinds of messages:*

```
  msgs m1 m2 m3 : Oid Oid -> Msg .
```

where `m1`(o_1, o_2) *is a message from* o_1 *to* o_2*. Each object has a* list *of messages it wants to send, and a list which stores the messages it has received:*

```
  class Node | msgsToSend : MsgList, msgsRcvd : MsgList .
  subclass Node < SeqNoNode .
```

If the initial state is

```
  < "A" : Node | msgsToSend : m2("A", "B") :: m3("A", "B") :: m1("A", "B"),
                 msgsRcvd : nil,
                 sentNos : ("B", 0),  rcvdNos : ("B", 0) >
  < "B" : Node | msgsToSend :  m1("B", "A") :: m3("B", "A"),
                 msgsRcvd : nil,
                 sentNos : ("A", 0),  rcvdNos : ("A", 0) >
```

then `"A"` *wants to send* `m2`, `m3`, *and then* `m1`, *in that order to object* `"B"`, *and* `"B"` *wants to send* `m1` *and* `m3` *to* `"A"`. *The specification/protocol should send and receive the messages in order, and store the messages received in the attribute* `msgsRcvd`, *so that the final state resulting from rewriting the above initial state should be*

```
< "A" : Node | rcvdNos : ("B",2), sentNos : ("B",3),
               msgsRcvd : m1("B","A") :: m3("B","A"), msgsToSend : nil >
< "B" : Node | rcvdNos : ("A",3), sentNos : ("A",2),
               msgsRcvd : m2("A","B") :: m3("A","B") :: m1("A","B"),
               msgsToSend : nil >
```

We may of course have more than two objects, so another initial state could be

```
< "A" : Node | msgsToSend : m1("A", "B") :: m2("A", "C") ::
                            m2("A", "B") :: m3("A", "B"),
               msgsRcvd : nil,
               sentNos : ("B", 0) ("C", 0),
               rcvdNos : ("B", 0) ("C", 0) >
< "B" : Node | msgsToSend : m3("B", "C") :: m1("B", "A") ::
                            m1("B", "C") :: m2("B", "A"),
               msgsRcvd : nil,
               sentNos : ("A", 0) ("C", 0),
               rcvdNos : ("A", 0) ("C", 0) >
< "C" : Node | msgsToSend : m3("C", "B") :: m3("C", "A") ::
                            m2("C", "B"),
               msgsRcvd : nil,
               sentNos : ("A", 0) ("B", 0),
               rcvdNos : ("A", 0) ("B", 0) >
```

Node `"B"` *should receive all messages from* `"A"` *in the order they are sent, so that* `m1("A", "B")` *should be read before* `m2("A", "B")`. *However, the message* `m3("C", "B")` *could be received by* `"B"` *both before and after it receives* `m1("A", "B")` *(and* `m2("A", "B")`*), as long as* `m3("C", "B")` *is received before* `m2("C", "B")`.

1. *Specify the message exchange using sequence numbers to achieve ordered communication between each pair of nodes.* Hint: *You may want to use a message "wrapper"*

   ```
   op _withSeqNo_ : Msg Nat -> Msg [ctor] .
   ```

   which adds a (sequence) number to a message.

2. *Use Maude's* `rew` *and* `frew` *commands to execute your specification starting with the initial states given above. Was your result as expected?*

3. *Use Maude's* `search` *command to search for a (bad) state, reachable from either of the above initial states, in which node* `"B"` *has received the message* `m3("A","B")` before *the message* `m2("A","B")`.

4. *Compute "by hand" the expected number of different final states, reachable from the second of the above initial states, in which the messages have been stored in the correct order.*

5. *Use Maude's* `search` *command to find* all *states, reachable from each of the the initial states given above, that cannot be further rewritten. Are the results satisfactory?*

8.2.2 Multicast in the Unordered Setting

Multicast means that a sender sends a message not only to *one* recipient, but to a whole group of recipients at once. The prototypical example of multicast is of course the sending of a spam email to all the email accounts on some kind of list. More benign examples include the sending of useful information, such as stock quotes or conference announcements, to a group of recipients who subscribe to such services.

A sender needs to know all the members in the multicast group. We can model this an address list by a set of object identifiers:

```
class Sender | multicast-group : OidSet, ...

sort OidSet .
subsort Oid < OidSet .
op none : -> OidSet [ctor] .
op _;_ : OidSet OidSet -> OidSet [ctor assoc comm id: none] .
```

The idea behing our model of multicast is that a message to a multicast group is considered to be equivalent to *a separate message to each recipient* in the group. If `m(A, B, ...)` is a message from object `A` to object `B`, then the "multicast message" `m(A, B ; C ; E, ...)` should be the same as the set of messages

```
m(A, B, ...)  m(A, C, ...)  m(A, E, ...)
```

Assume that we have a message declaration

```
msg m : Oid Oid ... -> Msg .
```

To be able to send a "multicast message" we must add the declaration

```
op m : Oid OidSet ... -> Configuration .
```

The following equations define the equivalence whereby a multicast message becomes a message to each member in the group:

```
vars O O' : Oid .    var OS : OidSet .

eq m(O, none, ...) = none .
ceq m(O, O' ; OS, ...) = m(O, O', ...)  m(O, OS, ...)
    if OS =/= none .
```

Exercise 171 *Explain why the condition is needed in the last equation above.*

Sending a message `m` to a multicast group can then be modeled by a rule having the form

```
rl [multicast] :
   < O : Sender | multicast-group : OS, ... >
  =>
   < O : Sender | ... >
   m(O, OS, ...) .
```

The receiving object just sees the separate message to itself and treats it as a message in the usual way, just like someone is supposed to believe that she is the only lucky recipient of an email from Nigeria promising millions of dollars for providing a bank account number and an initial payment.

A disadvantage with this approach is that one has to write the above equations for each kind of message that is multicast. One improvement is therefore to introduce "message wrappers" around messages. If `m` is the message to be broadcast, we may have a message wrapper `msg_from_to_`, so that `msg m(...) from A to B` models a message from `A` to `B` with content `m(...)`. A multicast message may then look like `multimsg m(...) from A to B ; C ; E`. The following module defines these wrappers:

```
(omod MESSAGE-WRAPPERS is protecting OID-SET .

  op msg_from_to_ : Msg Oid Oid -> Msg [ctor] .
  op multimsg_from_to_ : Msg Oid OidSet -> Msg [ctor] .

  var M : Msg .   vars SENDER ARECEIVER : Oid .
  var OTHER-RECEIVERS : OidSet .

  eq multimsg M from SENDER to none = none .
  eq multimsg M from SENDER to ARECEIVER ; OTHER-RECEIVERS =
        (msg M from SENDER to ARECEIVER)
        (multimsg M from SENDER to OTHER-RECEIVERS) .
endom)
```

In this way, *any* kind of message can be multicast by just enclosing it within the `multimsg_from_to` wrapper.

Example: A Protocol to Broadcast a Message

Given a network where each node knows its set of neighbors, some node may want to distribute a very important message to all the other nodes in the network (that are reachable from the sender). There is only *one* important message to transmit. A very simple protocol which achieves just that can be described as follows:

1. The sender multicasts the very important message to its neighbors.

2. When a node reads an important message for the first time, it stores the content of the message, and multicasts the message to its neighbors *except* the node from which it just received the message.

3. When a node receives an important message but has already received some important message (hopefully the same message!), it just ignores the message.

Again, this is not a *model* of broadcast, but a *protocol* to achieve broadcast using multicast.

In the following module, the important message to broadcast is either `m1`, `m2`, `m3`, `m4`, or `m5`. The network topology is modeled by each node having a `neighbors` attribute denoting its neighbors. The `msgRead` is `none` before an object has received the important message, and contains the important message otherwise. We use a message of the form `broadcast(m4, "b")` to start the protocol, so that object `"b"` is responsible to broadcast the important message (in this case `m4`) to all the other nodes. The constant `initState` defines a suitable initial state:

```
(omod BROADCAST is
  protecting OID-SET .                --- Sort OidSet
  including MESSAGE-WRAPPERS .
  protecting STRING .
  subsort String < Oid .              --- Object names are strings

  msgs m1 m2 m3 m4 m5 : -> Msg .      --- Important messages
  msg broadcast : Msg Oid -> Msg .

  class Node | neighbors : OidSet, msgRead : Configuration .

  var O O' : Oid .
  var OS : OidSet .
  var M : Msg .

  rl [startBroadcast] :
     broadcast(M, O)
     < O : Node | neighbors : OS, msgRead : none >
    =>
     < O : Node | msgRead : M >
     multimsg M from O to OS .

  --- Rules for receiving and forwarding messages are exercises and
  --- are therefore omitted here!

  op initState : -> Configuration .
  eq initState =
       broadcast(m4, "b")
       < "a" : Node | neighbors : "b" ; "e", msgRead : none >
       < "b" : Node | neighbors : "a" ; "d", msgRead : none >
       < "c" : Node | neighbors : "d", msgRead : none >
       < "d" : Node | neighbors : "b" ; "c" ; "e", msgRead : none >
```

```
      < "e" : Node | neighbors : "a" ; "d", msgRead : none > .
endom)
```

Exercise 172

1. *Complete the specification of the module* `BROADCAST` *with rules for receiving and forwarding messages.*
2. *Draw the network topology defined by the initial state* `initState`*.*
3. *What should the final state reachable from* `initState` *look like?*
4. *Execute the protocol using Full Maude's* `frew` *command.*
5. *Use Full Maude's capabilities to check that* each *final state reachable from* `initState` *is as expected.*

8.2.3 Broadcast in the Unordered Setting

Broadcast means that a node sends that a node sends a message to all the (other) nodes in the system. A trivial example is a television satellite which broadcasts TV signals to all the households in the the world, or a part thereof, that have certain kinds of reception equipment. Unlike for multicast, a broadcasting node is usually not aware of the identities of all the other nodes in the system. This makes modeling broadcast slightly trickier than modeling multicast.

The broadcast model presented here is influenced by discussions with José Meseguer and by Real-Time Maude [81]. The idea is to regard a broadcast message to be equivalent to a single message to all the other nodes in the system. To have "control" over all the nodes in the system, we introduce an operator

```
sort GlobalSystem .
op {_} : Configuration -> GlobalSystem [ctor] .
```

and require that the whole state has the form `{`*conf*`}`, for some configuration *conf*. A broadcast message wrapper can be declared

```
op broadcast : Msg Oid -> Configuration .
```

A broadcast message `broadcast m from` O is then supposed to be equivalent to

`(msg m from` O `to` O_1`)` ... `(msg m from` O `to` O_n`)`

if O, O_1, ..., O_n are *all* the objects in the system. Assuming that the nodes in the system are objects of a class `Node`[4], and knowing that all the objects in systems are enclosed within the pair of curly braces, the equations defining a `broadcast` message to be equal to single messages to all other nodes in the system are:

[4]This is not a serious restriction, since any class can be declared to be a subclass of the class `Node`.

```
var REST : Configuration .
vars O O' : Oid .
vars M M' : Msg .

eq {< O : Node | >  (broadcast M from O) REST}  =
   {< O : Node | >  (broadcast M from O to REST)} .
```

This equation essentially transforms a `broadcast` message into some kind of "multicast" message to the rest of the configuration, which can of course also contain other messages. The following equation states that a broadcast message to an object and some configuration is equivalent to a single message to the object and, recursively, a broadcast message to the remaining parts of the system:

```
eq broadcast M from O to (< O' : Node | > REST)  =
      < O' : Node | >
      (msg M from O to O')
      (broadcast M from O to REST) .
```

A broadcast to a message and the rest of the system is just a broadcast message to the rest of the system:

```
eq broadcast M from O to (M' REST) = M' (broadcast M from O to REST) .
```

The following equation ends the recursion:

```
eq broadcast M from O to none = none .
```

Finally, since we have declared `broadcast` messages to have the sort `Configuration` instead of the sort `Msg`, we need the following equation in case the state contains multiple `broadcast` messages:

```
eq broadcast M from O to ((broadcast M' from O') REST) =
     (broadcast M from O to REST)  (broadcast M' from O') .
```

This broadcast model is summarized in the following module:

```
(omod BROADCAST-MODEL is
 class Node .           --- All classes must be subclasses of this one.

 op msg_from_to_ : Msg Oid Oid -> Msg [ctor] .
 op broadcast_from_ : Msg Oid -> Configuration .

 sort GlobalSystem .
 op `{_`} : Configuration -> GlobalSystem .

 var REST : Configuration .
```

```
  vars O O' : Oid .
  vars M M' : Msg .

  eq {< O : Node | >  (broadcast M from O) REST}  =
     {< O : Node | >  (broadcast M from O to REST)} .

  --- An auxiliary function:
  op broadcast_from_to_ : Msg Oid Configuration -> Configuration .

  eq broadcast M from O to (< O' : Node | > REST)  =
       < O' : Node | >
       (msg M from O to O')
       (broadcast M from O to REST) .

  eq broadcast M from O to (M' REST) = M' (broadcast M from O to REST) .

  eq broadcast M from O to none = none .

  eq broadcast M from O to ((broadcast M' from O') REST) =
       (broadcast M from O to REST)  (broadcast M' from O') .
endom)
```

Broadcasting a message is done by a rule of the form

```
  rl [broadcast] :
     < O : ... >
    =>
     < O : ... >
     broadcast M from O .
```

It is worth noticing that

- a sender does not need to know the rest of the system, and
- the model allows for multiple (and *concurrent*) broadcasts at the same time.

Exercise 173 *Use the module* `BROADCAST-MODEL` *to define a system where any node can broadcast a message, and where a node records the messages it has received. Define an appropriate initial state with at least 6 nodes, where a node* `"A"` *wants to broadcast the message* `"Hi"`*, and a node* `"C"` *wants to broadcast a message* `"there"`*. Execute your specification using both the* `frew` *command and a search for all final states.*

Remember *that the initial state has to be a term of sort* `GlobalSystem`*, and that all objects have to be objects of the class* `Node`*.*

Exercise 174 *How would you modify the module* `BROADCAST-MODEL` *so that also the sender receives the* `broadcast` *message?*

Exercise 175 *Assume that we have three classes* `BroadcastingSatellite`, `HouseWithAntenna`, *and* `HouseWithoutAntenna` *that are not related by the subclass relation. Modify the module* `BROADCAST-MODEL` *so that a* `broadcast` *message becomes single messages only to objects of class* `HouseWithAntenna`. *Test your specification.*

8.2.4 Modeling Unreliable Communication

Messages can get lost during transmission. Another source of unreliability that is sometimes considered is that messages can be *duplicated* during transmission. This sounds less intuitive (how often do we receive two copies of the same message?), but is often considered to be an *abstraction* for settings where a message is re-sent with certain intervals. We therefore need to be able to model the possiblity of message loss and duplication.

The simplest solution is to have the following rules:

```
var M : Msg .

rl [lose-msg] :      M  =>  none .
rl [duplicate-msg] : M  =>  M  M .
```

With these rules, any message can be be lost or duplicated. The disadvantage is that it is not only messages in transmission that can get lost, but also messages inside message wrappers, leading to rewrites like `msg m from "A" to "B"` ⟶ `msg none from "A" to "B"` and `msg m from "A" to "B"` ⟶ `msg m m from "A" to "B"`, which was probably not intended. Furthermore, we have seen above that we often use attributes such as `msgRead` to store messages that have been received. With the above rules we could have rewrites such as

```
< O : Rcvr | msgRead : m1 m2 >  ⟶  < O : Rcvr | msgRead : m2 >
```

A sensible solution is to strictly enforce that only messages in transmission are terms of sort `Msg`, and that the actual content of a message is a term of some sort `MsgContent`.

If we use the message wrapper `msg_from_to_` whenever a message is sent, then we can modify the above solution to instead have the rules

```
var M : Msg .    vars O O' : Oid .

rl [lose-msg] :
   msg M from O to O'  =>  none .

rl [duplicate-msg] :
   msg M from O to O'  =>  (msg M from O to O')  (msg M from O to O') .
```

Yet another solution is to have an explicit "destroyer" object which is like a "shark" that swims in the configuration and devours and duplicates messages in the configuration. Having such an object ensures that only messages in transit are lost or duplicated:

```
class Destroyer .

var M : Msg .     var O : Oid .

rl [devour-msg] :
   M  < O : Destroyer | >  =>  < O : Destroyer | > .

rl [duplicate-msg] :
   M  < O : Destroyer | >  =>  < O : Destroyer | >  M  M .
```

The advantage of this last solution is that we can easily accomodate for test cases where messages are not lost or duplicated by just not having a `Destroyer` object in the initial state, and that the solution can be easily modified to model a setting where, say, at most 20 messages are lost or duplicated in a single execution. Its disadvantage is the lack of concurrency, and that it defines a less elegant model.

Exercise 176 *Explain more carefully the difference in concurrency between the last solution and the second solution above.*

Exercise 177 *Define a class* `LimitedDestroyer`*, whose objects can perform at most 20 losses or duplications during an execution. Such objects may be conceptually ugly, but can be handy for analysis purposes.*

8.3 Asynchronous Communication through Ordered Message Passing in Links

By *ordered message delivery* one could mean either of the following:

1. A sequence of messages sent from an object A to an object B are received in the order in which they are sent. Sending packets through a data link is such an example.
2. All messages to an object A are received in the order they were sent, no matter who is the sender. Communication by leaving messages on an answering machine is an example of this kind of communication.
3. A message m is read before a message m' if m was sent before m', no matter who are the senders and receivers. I do not have any good real-life example of such communication.

We will focus almost exclusively on the first of these kinds of communication.

An underlying communication medium that provides ordered communication can often be seen as a *link* (or a *channel* or a *buffer*) between two communicating "objects." Ordered communication can therefore be modeled using link objects to model the actual link and the message transmission through the link.

Recalling the definition of the sort `MsgList` on page 230, a (unidirectional) link between two objects can be represented by an object of the following class `Link`:

```
class Link | content : MsgList .
```

where `content` is the *list* of messages traveling in the link from the sender object to the receiver object. But which is the sender (or *source*) object and which is the receiver (or *destination*) object of the link? The trivial solution is to declare the class `Link` as follows:

```
class Link | source : Oid, destination : Oid, content : MsgList .
```

It is a matter of taste, but I prefer to instead have the name of the source node and of the destination node in the name of the link object, so that an object

< o to o' : Link | content : $m1$ :: $m2$ >

models a link with the object named o as its source node, and the object named o' as its destination node. (The message $m1$ is the next message which can be read by o'.)

The following module defines the class `Link` and the names intended to be used as link names:

```
(omod LINK is protecting MSG-LIST .
  class Link | content : MsgList .
  op _to_ : Oid Oid -> Oid [ctor] .
endom)
```

A *bidirectional* communication link can be modeled by a new class

```
class BiLink | contentDown : MsgList, contentUp : MsgList .
```

where `contentDown` contains the messages traveling in one direction in the link and `contentUp` contains the messages traveling the other way (the direction may be related to the name of the link). However, I prefer to model a bidirectional link by two unidirectional ones:

```
< "o1" to "o2" : Link | content : m1 :: m2 >
< "o2" to "o1" : Link | content : m3 >
```

because one need not worry about whether messages to a certain object is in the `contentDown` or `contentUp` attribute.

The global state should contain one `Link` object (two for bidirectional communication channels) between each pair of nodes which are connected. The network in Fig. 8.1 can be represented by the state

```
    < "a" : Node | ... >          < "b" : Node | ... >
    < "c" : Node | ... >          < "d" : Node | ... >
< "a" to "b" : Link | ... >   < "b" to "a" : Link | ... >
< "a" to "c" : Link | ... >   < "c" to "a" : Link | ... >
< "a" to "d" : Link | ... >   < "d" to "a" : Link | ... >
< "b" to "d" : Link | ... >   < "d" to "b" : Link | ... >
```

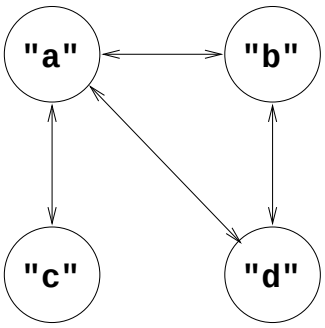

Figure 8.1: A communication topology.

A message is sent over a link by inserting it at the *back* of the link, so that the sending of a message `m` from an object `O` to an object `O'` is modeled by a rule of the form

```
var ML : MsgList .

rl [send-m] :
   < O : ... | ... >
   < O to O' : Link | content : ML >
  =>
   < O : ... | ... >
   < O to O' : Link | content : ML :: m > .
```

An object `O'` reads the "next" message from an object `O` by removing the *first* element in the link from `O` to `O'`:

```
rl [read-m] :
   < O' : ... | ... >
   < O to O' : Link | content : m :: ML >
  =>
   < O' : ... | ... >
   < O to O' : Link | content : ML > .
```

Exercise 178 *In this exercise we use links to send a sequence of messages from a node to another. The class* `Node` *is defined*

```
class Node | msgsToSend : MsgList, msgsRcvd : MsgList .
```

Each object wants to send the same sequence of messages as in Exercise 170.

1. *What are the two initial states corresponding to the two initial states given in Exercise 170?*
2. *Define the rules for sending and receiving (and storing) messages (one by one) in the example.*

3. *Test your specification on the initial states above. Then search for the error situation we searched for in Exercise 170, namely, where node* `"B"` *has received the message* `m3("A","B")` *before the message* `m2("A","B")`*. Finally, check all possible final states of the protocol to make sure that all messages between pairs of objects were received in the sending order.*

Exercise 179

1. *Define some kind of link object, together with rules (or rule "schemes") for "sending" and "receiving " messages, to model the "answering machine" form of communication where all messages to the same object are read in the order sent, no matter who sent the messages.*

2. *Repeat the exercise above for the "global" form of ordered communication where* all *messages are read in the order they were sent, no matter who is the sender or receiver.*

3. *What can you say about* concurrency *in the two models above. That is, can*

   (a) *two objects send messages at the same time?*

   (b) *two objects send messages to the same object at the same time?*

   (c) *two objects read messages at the same time?*

   (d) *an object o read a message from an object o' at the same time that object o' is sending another message to o?*

   (e) *can an object o read a message while another object o' is sending a message (to some other object)?*

   Do these concurrency results make sense with respect to what would be expected from such communication forms?

Links and Concurrency

The specification should model the world faithfully with respect to possible *concurrent* steps to make reasoning about concurrency meaningful. We have seen that a message or an object can take part in at most *one* rule application at a time. (This follows easily from the definition of concurrent steps.) `"Ophelia"` could not celebrate her birthday *and* her engagement in the same step.

In the "link-less" message passing world, two objects could send messages at the same time, and an object o could read a message from o' while o' sends another message to o. These consequences seem natural. The fact that an object cannot take part in more one rule application, together with the fact that a link is modeled by an object, implies that an object o cannot write to the link "o_1 `to` o_2" at the same time when the object o_2 *reads* a message from the same link. Is this a natural model of concurrency in a link? It depends on the link being modeled. Maybe one cannot both send to and read from the same link at the same time? Maybe it is possible? A link may also model a fiber-optic cable or,e.g., the the cars entering

and exiting the Channel Tunnel between France and the U.K.[5] In the latter case it should indeed be possible for different cars to enter and exit a link at the same time.

One way of achieving concurrency, so that one message may enter a link while another message exits the same link, is to regard the link as consisting of two parts, its "back" and its "front." Messages should enter the "back" part of the link and should exit from its "front" part. These two parts of a link can be declared to be objects of the following classes:

```
class LinkFront | front : MsgList .
class LinkBack | back : MsgList .
```

The attribute `front` contains the messages in the "front" part of the link and `back` the messages in its "back" part. A link can be seen as consisting of its two parts, so we have the crucial equivalence

```
 < O to O' : Link | content : ML :: ML' >
                  =
 < O to O' : LinkFront | front : ML >
 < O to O' : LinkBack | back : ML' >
```

(for all `O`, `O'`, `ML`, and `ML'`). Sending a message `m` from `O` to `O'` is done by appending `m` to the back of the link:

```
rl [sendConcLink-m] :
   < O : ... | ... >
   < O to O' : LinkBack | back : ML >
  =>
   < O : ... | ... >
   < O to O' : LinkBack | back : ML :: m > .
```

Reading is done by removing the first element from the front of the link:

```
rl [readConcLink-m] :
   < O' : ... | ... >
   < O to O' : LinkFront | front : m :: ML >
  =>
   < O' : ... | ... >
   < O to O' : LinkFront | front : ML > .
```

That is, we regard *one* object as being *equivalent* to *two* objects, each of which can participate in a rewrite step at the same time.

Messages are only added to the `LinkBack` part/object, so the following *equation* is needed to move messages from the `LinkBack` (part of the `Link`) object to the `LinkFront` (part of the `Link`) object:

[5] Assuming that there is only one open lane in each direction!

```
eq < O to O' : LinkBack | back : ML >
   < O to O' : LinkFront | front : ML' >
    =
   < O to O' : LinkBack | back : nil >
   < O to O' : LinkFront | front : ML' :: ML > .
```

A skeptic reader may ask whether the above equality does not model an *action* and therefore should be modeled by a *rule* instead of an *equation*. No, the above equality follows from the fundamental equality

```
< O to O' : Link | content : ML :: ML' >
                 =
< O to O' : LinkFront | front : ML >
< O to O' : LinkBack | back : ML' >
```

in equational logic, so that adding the "suspect" equation does not change the mathematical/logical meaning of the specification. The extra equation is necessary for the *operational* meaning of the specification.

Exercise 180 *Show that the "extra" equation follows logically from the "fundamental equality" above.*

Exercise 181 *Given the above declarations and equations for links and their parts, and the rules* `sendConcLink-m` *and* `readConcLink-m` *for sending to and receiving from the link-parts*[6]*, use the deduction rules of rewriting logic to prove that*

```
< "o1" : Node | ... >    < "o2" : Node | ... >
< "o1" to "o2" : LinkFront | front : m3 :: m2 >
< "o1" to "o2" : LinkBack | back : nil >
```

(which "represents" the state

```
< "o1" : Node | ... >    < "o2" : Node | ... >
< "o1" to "o2" : Link | content : m3 :: m2 >)
```

rewrites in one concurrent step *to*

```
< "o1" : Node | ... >    < "o2" : Node | ... >
< "o1" to "o2" : LinkFront | front : m2 :: m1 >
< "o1" to "o2" : LinkBack | back : nil >
```

(which represents the state

```
< "o1" : Node | ... >    < "o2" : Node | ... >
< "o1" to "o2" : Link | content : m2 :: m1 >).
```

[6] Assume that `m` in the rules could be either of `m3` and `m1`.

8.3.1 Unreliable Links

A *lossy* link, i.e., a link in which messages in transit can be lost[7], can be modeled as an object of the following subclass `LossyLink`, where the rule `lose-msg` models the loss of a message:

```
class LossyLink .
subclass LossyLink < Link .

vars ML ML' : MsgList .    var M : Msg .    vars SOURCE DEST : Oid .

rl [lose-msg] :
   < SOURCE to DEST : LossyLink | content : ML :: M :: ML' >
  =>
   < SOURCE to DEST : LossyLink | content : ML :: ML' > .
```

A link which allows for a message in transmission to be duplicated can be modeled as an object of the following class `DuplLink`:

```
class DuplLink .
subclass DuplLink < Link .

rl [duplMsg] :
   < SOURCE to DEST : DuplLink | content : ML :: M :: ML' >
  =>
   < SOURCE to DEST : DuplLink | content : ML :: M :: M :: ML' > .
```

Finally, the following class `UnrelLink` specifies links where messages can get lost as well as getting duplicated during transmission:

```
class UnrelLink .
subclass UnrelLink < LossyLink DuplLink .
```

For full generality, the rewrite rules involving sending and receiving messages shold mention links of the superclass `Link`, so that they apply to all kinds of links. The initial states should then specify exactly what kind of links are used in each case. In this way, we can easily model systems with different kinds of links: some links may be reliable while other links are lossy and/or duplicating.

8.3.2 Links with Limited Capacity

Quite often it is the case that messages are dropped because the link is "full." A link which can transport at most N messages[8] can be modeled by the following class `BoundedLink`:

[7] In communicating systems, a message loss is also often used as an *abstraction* for the case where a message has been found to have been corrupted during transmission.

[8] Such a model may of course be easily refined to a model which also takes the *size* of the messages into account; the reference [82] shows one such model.

```
class BoundedLink | content : MsgList, capacity : NzNat, currentSize : Nat .
```

where `currentSize` is the size of the list in the `content` attribute[9]. The sending of a message `m` through such a link must be modeled by rules of the forms

```
var NZ : NzNat .   var N : Nat .   vars O O' : Oid .   var ML : MsgList .

crl [send-OK] :
    < O : ... >
    < O to O' : BoundedLink | content : ML, capacity : NZ, currentSize : N >
   =>
    < O : ... >
    < O to O' : BoundedLink | content : ML :: m, currentSize : s N >
   if N < NZ .

rl [send-full] :
   < O : ... >
   < O to O' : BoundedLink | capacity : NZ, currentSize : NZ >
  =>
   < O : ... >
   < O to O' : BoundedLink | > .
```

8.3.3 Multicast Through Links

While the the above models for different forms of communication seem fairly intuitive, I have yet to see an intuitive model for multicast via link objects. In this section, I suggest *one* way of achieving multicast through links that seems semantically sound and that performs the multicast in *one* rewrite step. My solution, however, is not as simple or intuitive as the models we have encountered thus far in these notes.

The intuitive approach, namely, to send a message through multiple links by sending the message to each recipient in separate rewrite steps is not desired because it uses many rewrite steps to achieve an action which should be performed in *one* step. The brutal way to reduce the number of rewrite steps is to transform rewrite rules into equations. However, this tends to give models that, while Maude may execute them in the desired way, are semantically suspect.

My idea is to use an additional "spreader" object, which "spreads" (or "sprinkles" or "distributes") the message to multicast into the appropriate links. My spreader class is defined as follows:

```
class Spreader | dest : OidSet, msgToSend : DefMsg, sendTo : OidSet .

op spreader : Oid -> Oid [ctor] .
```

[9]The `currentSize` attribute is not needed, since its value can be computed given the `content` value; however, it is usually more "efficient" to have such an attribute.

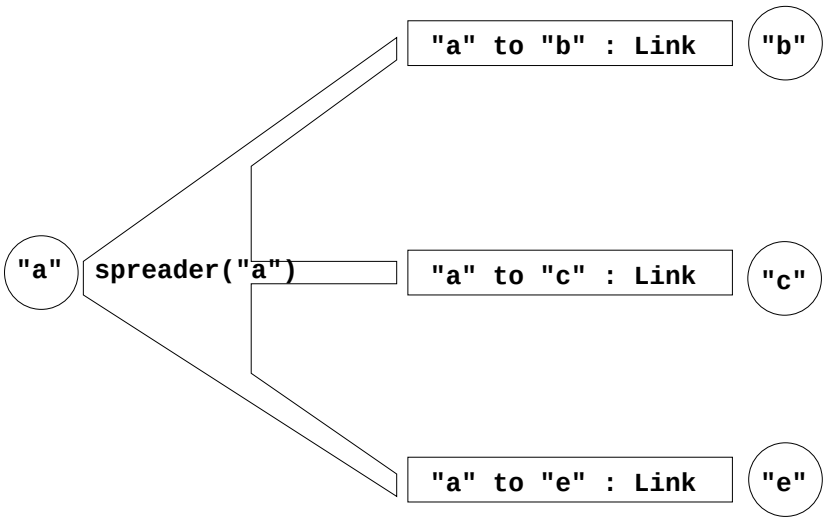

Figure 8.2: A communication topology with "spreaders" and links.

The spreader belonging to an object named O is assumed to have identifier `spreader(`O`)`. Fig. 8.2 shows a setting where the object `"a"` have links *to* its neighbors `"b"`, `"c"` and `"e"`. The spreader from `"a"`, with identifier `spreader("a")`, can then take a message from `"a"` and insert it into each of the links `"a" to "b"`, `"a" to "c"`, and `"a" to "e"`. The `dest` attribute of `spreader(`o`)` is supposed to denote the objects to which the object o has an outgoing link. (In the example above, `spreader("a")`'s `dest` value is `"b" ; "c" ; "e"`.) The `msgToSend` attribute takes as value the message to be multicast, and is `noMsg` when there is no message to distribute to the links. Finally, the `sendTo` attribute contains the destination objects for which the spreader has not yet inserted a message into the link. This attribute looks similar to `dest`, and is used for

- the modeling of the distribution mechanism of the spreader; and
- allows an object to multicast a message to a *subset* of its outgoing links.

A spreader object in a state

```
< spreader("a") : Spreader | dest : "b" ; "c" ; "e",
                             msgToSend : m,
                             sendTo : "c" ; "e" >
```

denotes the spreader of the object `"a"`, where `"a"` is supposed to have outgoing links exactly to the objects `"b"`, `"c"`, and `"e"`, and where the spreader has yet to insert the message `m` into the links `"a" to "c"` and `"a" to "e"`. I assume that `sendTo` has the same value as `dest` whenever `msgToSend` is `noMsg`.

The multicast of a message m from an object O to all its neighbors (i.e., to all the objects to which O has an outgoing link) should therefore be modeled by a rule of the form

```
rl [multicast-m] :
   < O : ... >
   < spreader(O : Spreader | msgToSend : noMsg >
  =>
   < O : ... >
   < spreader(O : Spreader | msgToSend : m > .
```

Note that the sender doesn't need to know its neighbors, since the spreader is assumed to know them all.

The following equations model the transfer of a message from a spreader into the appropriate link objects:

```
vars SENDER O : Oid .  var M : Msg .  var ML : MsgList .  var OS : OidSet .

--- Insert the message recursively into all links:
eq < spreder(SENDER) : Spreder | msgToSend : M, sendTo : O ; OS >
   < SENDER to O : Link | content : ML >
     =
   < spreder(SENDER) : Spreder | msgToSend : M, sendTo : OS >
   < SENDER to O : Link | content : ML :: M > .

--- Insertion into links of message M completed:
eq  < spreder(SENDER) : Spreder | msgToSend : M, dest : OS,
                                  sendTo : none >
      =
    < spreder(SENDER) : Spreder | msgToSend : noMsg, dest : OS,
                                  sendTo : OS > .
```

We use *equations* because we can consider a system with a message in the spreader to be equivalent to a system where all the messages are already in the links. The careful reader may observe that the resulting equational theory is *not* confluent when there are more than one spreader from a sender; this setting should of course not occur in any reachable state.

Exercise 182 *Show that the equations defining a spreader are not confluent if an object can have multiple spreaders.*

The `sendTo` attribute can, as mentioned, be used to multicast messages to a subset of the neighbors by rules of the following form:

```
rl [multicast-m-to-some-friends] :
   < o : ... goodFriends : os >
   < spreader(o : Spreader | msgToSend : noMsg >
  =>
   < o : ... >
   < spreader(o : Spreader | msgToSend : m, sendTo : os > .
```

The initial states should be defined as before for systems with links, with an additional `Spreader` object for each node that has outgoing links. The initial values of the attributes `dest`, `msgToSend`, and `sendTo` should be, respectively, the outgoing neighbors, `noMsg`, and the outgoing neighbors again.

Exercise 183 *Define an initial state with spreader objects and links corresponding to the topology in Fig. 8.1. You may assume that the nodes in the system are instances of a class* `Node` *with no attributes. Assume that the links to and from object* `"b"` *are reliable, and that the other links are lossy.*

The spreaders provide *additional* multicast capabilities to links, which of otherwise work as before. That is, we can still have different kinds of links in the initial state. Unicast communication works as before.[10] The creation and deletion of new links can also be easily accomodated.

BROADCAST PROTOCOL OPPGAVE

BROADCAST

EXAMPLES

When to Use Which Communication Model?

When should unordered message communication be used, and when should links be used?

The aim is to model a system as faithfully and abstractly as possible[11]. Links should be used if the underlying communication medium is assumed to provide ordered communication, *and this is significant for the system to be modeled.* Communication should be modeled by "ordinary" message passing if it may not be assumed that communication is ordered. Note that "ordinary" message passing is more general, since it models *any possible* delivery sequence—including the ordered one! If a system works correctly when using *unordered* communication is assumed, it will also work correctly when messages are received in the order they are sent.

[10] Indeed, there are now *two* ways of unicasting a message: by inserting it directly in the link, or by using the spreader and setting a single destination node in its `sendTo` attribute.

[11] "faithfully" and "abstractly" may seem contradictory, but should not be

Chapter 9

Case Study: The Two-Phase Commit Protocol for Distributed Databases

This chapter illustrates how Maude can be used to model and analyze the *two-phase commit* (2PC) protocol [58] for *distributed database systems*.

The book [85] defines a distributed database as *a collection of multiple, logically interrelated databases distributed over a computer network*. More roughly speaking, a distributed database is one database where parts of the database are distributed at different locations. Data are usually *replicated*, so that some information are stored in multiple locations. For example, my banking details may be stored both in my local branch and at the bank's main branch. Furthermore, the same information seems also to be stored in government databases. Likewise, many different government agencies store my name and social security number. You can probably think of better examples.

One crucial requirement of a distributed databases is *consistency*. Replicated data should have the same value in all parts of the database where they appear. Imagine, for example, that I make a down payment of 10.000 NOK on my mortgage. This fact should be reflected both in my local bank's database, and in the other branches of the bank that stores information about my mortgage. Likewise, if I change my name, civil status, or social security number, this change must be made (or *not* made!) in all the appropriate government databases. The two-phase-commit protocol aims at achieving such consistency by ensuring that a multi-database update can be regarded as one atomic update.

Why is it potentially interesting to model and analyze a distributed database protocol in Maude?

- A Maude specification is a *formal*, unambiguous specification of the protocol, whereas a text-book prose description may often be unclear and ambiguous.
- There can be many *implicit assumptions* in informal descriptions. For example, under what assumptions about communication is the protocol supposed to work (ordered communication? lossy communication?)? Such assumptions are sometimes not made very explicit in a description of a protocol. The authors probably have a set of assumptions in mind, and these may well be stated in, or inferred from, a long text-book.

- A mathematical model of a protocol can be subjected to *mathematical* reasoning about its correctness.

- We can analyze the protocol by *executing* it using Maude's rewriting, search, and temporal logic model checking commands.

- There are many kinds of distributed systems and protocols (e.g., cryptographic protocols, transport/communication protocols, database protocols, etc.), each with its own notation(s) and implicit knowledge. This leads to a "fragmentation" where it is difficult to understand different kinds of systems. The Maude formalism is expressive, general, and fairly intuitive. Maude therefore has the potential to be some kind of common *lingua franca* in which many different kinds of systems can be easily specified and understood. This is one of the themes of these lecture notes. Furthermore, larger systems may well combine different aspects, such as, say, both a transport protocol and a security protocol; such advanced systems may then be easily expressible in Maude.

- Finally, we have had experiences with network engineers who found that an object-oriented Maude specification was more intuitive than the different ways (such as informal UML-like use cases and transition diagrams) they used to describe a sophisticated communication protocol [82]. Maybe the reader of these lecture notes will likewise find it easier to understand a Maude specification than a text-book description? Or at least that the two combined allows her to understand the system well.

9.1 Description of the Two-Phase Commit Protocol

Assume that we want to update a distributed database. For example, I want to change my name. This update should then happen at *all* the database components which contain my name. However, for various reasons, it may be the case that a component can *not*, or *does not* want to, update its local data. That could be either because some component doesn't allow the change (e.g., one branch of government does not allow me to change my name), a site failure, some kind of deadlock, or some other database-theoretic reason which is beyond the scope of this course. For our purposes, it is enough to be aware of the possibility that a database component may be unable/unwilling to update the database on disk.

Since the most important concern is that the database is consistent, it must be the case that either *all* the distributed component *commit* to physically update the database, or that *no* component does so. So, just like a permanent member of the UN Security Council, *any* component may "veto" an update of the distributed database. The two-phase commit protocol is intended to achieve exactly such consistency. The databases are not physically updated *during* the database transaction. Instead, the database is physically changed only at the end of the transaction if everything went well in each database component.

The 2PC protocol starts by selecting some component to be the *coordinator* of the commit effort. The two phases of 2PC are then given as follows in the text-book [33, Chapter 19]:

Phase 1. When all participating databases signal the coordinator that the part of the multidatabase transaction involving each has concluded, the coordinator sends a message *prepare for commit* to each participant to get ready for committing the transaction. Each participating database receiving that message will force-write all log records and needed information for local recovery and then send a *ready to commit* or *OK* signal to the coordinator. If the force-writing to disk fails or the local transaction cannot commit for some reason, the participating database sends a *cannot commit* or *not OK* signal to the coordinator. If the coordinator does not receive a reply from the database within a certain amount of time, it assumes a *not OK* response.

Phase 2. If *all* participating databases reply *OK*, and the coordinator's vote is also *OK*, the transaction is successful, and the coordinator sends a *commit* signal for the transaction to the participating databases. Because all the local effects of the transaction and information needed for local recovery have been recorded in the logs of the participating databases, recovery from failure is now possible. Each participating database completes transaction commit by writing a *commit* entry for the transaction in the log and permanently updating the database if needed. On the other hand, if one or more of the participating databases or the coordinator have a *not OK* response, the transaction has failed, and the coordinator sends a message to *roll back* or *UNDO* the local effect of the transaction to each participating database. This is done by undoing the transaction operations, using the log.

The net effect of the two-phase commit protocol is that either all participating databases commit the effect of the transaction or none of them do. In case any of the participants—or the coordinator—fails, it is always possible to recover to a state where either the transaction is committed or it is rolled back. A failure during or before Phase 1 usually requires the transaction to be rolled back, whereas a failure in Phase 2 means that a successful transaction can recover and commit.

Although we model the two-phase commit protocol as described above, it must be noted that the protocol is often described somewhat differently, so that each component must also send an *acknowledgment* message back to the coordinator as a response to a *commit* or *roll back/UNDO* message (see e.g., [96]).

9.2 Abstraction

In order to conveniently being able to model and analyze a system, it is important to disregard any detail that is not necessary for the kind of modeling and analysis to be performed. In other words, one should *abstract from as many details as possible, but not more.*

For example, when modeling the previous communication protocols, we had a very abstract model of communication, modeled by axioms such as associativity and commutativity of the configuration union operator. This is of course an abstraction for the way communication really is performed, with messages having detailed packets headers, packets being routed through a bunch of nodes, and so on. It should be obvious that modeling communication as it really happens in all its glorious details would be a quite futile exercise. It is not only for simplifying the modeling and analysis task that we abstract from some details. Such abstraction makes the system more general. For example, if we can show that a communication protocol is "correct" with our usual (asynchronous, out-of-order) communication model, then

the protocol is also correct for the settings where the communication is ordered. If we insisted on capturing much detail, then the system would only be analyzed for that very specific setting.

In analyzing 2PC, we are mainly interested in analyzing whether it indeed guarantees consistency of the distributed database. That is, will the different database components be updated or not? For instance, to analyze 2PC we are not interested even in the content of the database! Another issue is that the description of 2PC says that "if the coordinator does not receive a reply from the database within a certain amount of time, it assumes a *not OK* response." We could of course add timers to capture this aspect. That would potentially give us a more precise description of 2PC, but at the cost of generality (the protocol would only be analyzed for *one* given value of "a certain amount of time") and of having to deal with *time* (which can be done in an extension of Maude called Real-Time Maude [81, 80]). Instead, we assume that a *prepare for commit* message always gets an answer, where the timeout scenario sketched above corresponds to receiving a *not OK* message. We have then hidden the details of how the system deals with time, and just assume that when the system somehow detects a timeout, it is the same as reading a *not OK* message. Other aspects of a database system, such as reading and writing from/to the database, do not appear in the description of the 2PC protocol and should not be modeled.

9.3 Assumptions

The above informal description of 2PC leaves many assumptions implicit. For example: is communication assumed to be ordered or unordered? is communication reliable? does each node know the other nodes? I guess that reading the text-book carefully and/or having experience in database theory would answer these questions. According to our database expert, communication can be assumed to be unordered, each node that will ever be a coordinator knows all the other nodes, and the protocol should be analyzed for both reliable and unreliable communication.

9.4 Maude Specification of 2PC

The crucial requirement of 2PC that we want to analyze is of course whether the database is consistent at the end of a protocol run. That is, either *all nodes* or *no node* should have physically updated itself. Furthermore, we need to make sure that if even one participant votes to abort the transaction, then no updates are performed, and if all nodes want to commit, then all components should be updated.

We will first analyze 2PC for reliable communication, and then for unreliable communication.

Each component of the database is modeled by an object of the following class `2PCDB`:

```
class 2PCDB | updated : Bool, state : CommitState, otherNodes : OidSet,
              coordState : CoordState, anyNegative : Bool .

sort CoordState .
```

```
op notCoord : -> CoordState [ctor] .       --- not coordinator
op waitFor : OidSet -> CoordState [ctor] . --- coord waits for reply

sort CommitState .
ops initial ready abort : -> CommitState [ctor] .
```

Here, `updated` is the main attribute. It is `true` if and only if the database component has performed the update on disk. `state` is the internal state of the node (`initial` in the beginning; and then the node decides whether *it* is `ready` to commit or must `abort`). `otherNodes` denotes all the other nodes in the system, `coordState` is `notCoord` for nodes that are not currently coordinators, and it `waitFor(`*os*`)` when a coordinator is waiting for reply from the nodes *os*, and, finally, `anyNegative` is just a place for the coordinator to remember if any nodes wants to abort.

The messages in the system are declared as follows:

```
ops prepare OK notOK abort commit : -> MsgCont [ctor] .
msg startCommit : Oid -> Msg .
```

We use `startCommit` as a "message" which starts a run of the protocol. In addition, we use the usual "message wrappers" (or "envelopes") `msg_from_to` and `multimsg_from_to`, *which now are assumed to take as first parameter a term of sort* `MsgCont` *denoting the content of messages.*

2PC starts with the coordinator (that is, the node that gets the `startCommit` message) sending a `prepare` (to commit) message to all the other nodes, and going into waiting mode:

```
vars O O' : Oid .  var OS : OidSet .

rl [prepareReq] :
   startCommit(O)
   < O : 2PCDB | state : initial, otherNodes : OS >
  =>
   < O : 2PCDB | coordState : waitFor(OS) >
   multimsg prepare from O to OS .
```

When a node gets a prepare message, it must send either an `OK` or a `notOK` message back to the coordinator. (Remember that we also model a timeout by sending a `notOK` message.)

```
rl [ok] :
   (msg prepare from O to O')
   < O' : 2PCDB | state : initial >
  =>
   < O' : 2PCDB | state : ready >
   (msg OK from O' to O) .
```

```
rl [notOK] :
   (msg prepare from O to O')
   < O' : 2PCDB | state : initial >
  =>
   < O' : 2PCDB | state : abort >
   (msg notOK from O' to O) .
```

Of course, the coordinator itself should also vote:

```
rl [coordNotOk] :
   < O : 2PCDB | state : initial, coordState : waitFor(OS) >
  =>
   < O : 2PCDB | state : abort, anyNegative : true > .

rl [coordOk] :
   < O : 2PCDB | state : initial, coordState : waitFor(OS) >
  =>
   < O : 2PCDB | state : ready > .
```

In the second phase, the coordinator should read the responses and decide whether or not to order a global abort or a global commit. First, it reads the responses, and sets `anyNegative` to `true` if some node cannot commit:

```
rl [recOK] :
   (msg OK from O' to O)
   < O : 2PCDB | coordState : waitFor(O' ; OS) >
  =>
   < O : 2PCDB | coordState : waitFor(OS) > .

rl [recNotOk] :
   (msg notOK from O' to O)
   < O : 2PCDB | coordState : waitFor(O' ; OS) >
  =>
   < O : 2PCDB | coordState : waitFor(OS), anyNegative : true > .
```

Next, the coordinator sends its decision and stops being a coordinator (and updates its own database if needed):

```
rl [commitAll] :
   < O : 2PCDB | coordState : waitFor(none), otherNodes : OS,
                 anyNegative : false >
  =>
   < O : 2PCDB | coordState : notCoord, updated : true >
   (multimsg commit from O to OS) .
```

```
rl [abortAll] :
   < O : 2PCDB | coordState : waitFor(none), otherNodes : OS,
                 anyNegative : true >
  =>
   < O : 2PCDB | coordState : notCoord, updated : false >
   (multimsg abort from O to OS) .
```

Finally, the other nodes receive the coordinator's decision and decide whether to physically update the database:

```
rl [recAbort] :
   (msg abort from O to O')
   < O' : 2PCDB | >
  =>
   < O' : 2PCDB | updated : false > .

rl [recCommit] :
   (msg commit from O to O')
   < O' : 2PCDB | >
  =>
   < O' : 2PCDB | updated : true > .
```

This finishes our specification of the two-phase commit protocol.

9.5 Analyzing 2PC in Maude

In our specification we did not add rules for message loss, so we first analyze our protocol in a reliable setting. The following module defines a suitable initial state with five database components:

```
(omod TEST-2PC is
  including TWO-PHASE-COMMIT .
  protecting STRING .

  subsort String < Oid .

  op init : -> Configuration .
  eq init =
     startCommit("a")
     < "a" : 2PCDB | updated : false, state : initial,
                     otherNodes : "b" ; "c" ; "d" ; "e",
                     coordState : notCoord, anyNegative : false >
     < "b" : 2PCDB | updated : false, state : initial,
                     otherNodes : "a" ; "c" ; "d" ; "e",
                     coordState : notCoord, anyNegative : false >
```

```
     < "c" : 2PCDB | updated : false, state : initial,
                     otherNodes : "b" ; "a" ; "d" ; "e",
                     coordState : notCoord, anyNegative : false >
     < "d" : 2PCDB | updated : false, state : initial,
                     otherNodes : "b" ; "c" ; "a" ; "e",
                     coordState : notCoord, anyNegative : false >
     < "e" : 2PCDB | updated : false, state : initial,
                     otherNodes : "b" ; "c" ; "d" ; "a",
                     coordState : notCoord, anyNegative : false >  .
endom)
```

As always, to get some quick first feedback, we perform a quick rewrite of the system:

```
Maude> (frew init .)

result Configuration :
  < "a" : 2PCDB | state : ready, updated : false, ... >
  < "b" : 2PCDB | state : abort, updated : false, ... >
  < "c" : 2PCDB | state : ready, updated : false, ... >
  < "d" : 2PCDB | state : abort, updated : false, ... >
  < "e" : 2PCDB | state : ready, updated : false, ... >
```

This is promising: the databases `"b"` and `"d"` decided they needed to abort the transaction, and we see that none of the components updated its database. Of course, the rewrite command only analyzes one possible behavior. We therefore check for consistency of the distributed database *at the end of a run of 2PC* by searching for a *final* state in which one component has updated its database while another components has not done so:

```
Maude> (search [1] init
                  =>!
                   C:Configuration
                   < O:Oid : 2PCDB | updated : false >
                   < O':Oid : 2PCDB | updated : true > .)

No solution.
```

Good! It is not possible to reach an inconsistent database from our initial state. However, this was only a part of the correctness requirement, which also said that (i) if one component decides to abort, then no component should update its database, and (ii) if all components are ready to update, then they should indeed all update. Again, we analyze these two properties by searching for *final* states in which the properties do not hold:

```
Maude> (search [1] init
                 =>!
                  C:Configuration
```

```
                    < O:Oid : 2PCDB | state : abort >
                    < O':Oid : 2PCDB | updated : true > .)

No solution.

Maude> (search [1] init
                    =>!
                     < O1:Oid : 2PCDB | state : ready, updated : false >
                     < O2:Oid : 2PCDB | state : ready >
                     < O3:Oid : 2PCDB | state : ready >
                     < O4:Oid : 2PCDB | state : ready >
                     < O5:Oid : 2PCDB | state : ready > .)

No solution.
```

So, everything looks good. Of course, we have not *proved* 2PC correct in any way; we have only shown that from the initial state `init` all will work well. In principle, there *could* have been other initial states from which we could have reached an inconsistent state. Nevertheless, this analysis has certainly increased our confidence in the correctness of 2PC.

9.6 Analyzing 2PC with Unreliable Communication

We now analyze 2PC in a lossy communication setting.

First of all, we notice that we need *not* model loss of `prepare`, `OK`, and `notOK` messages, since we had from the informal specification that *if the coordinator does not receive a reply from the database within a certain amount of time, it assumes a* not OK *response.* So, if any of these messages would have been lost in transmission, the timeout mechanism would have discovered it and would have "assumed" (in our case: sent) a `notOK` answer. Therefore, we only need to consider the possibility that an `abort` or a `commit` message is lost. The following module extends our specification of 2PC, as well as the initial state `init`, with exactly those possible message losses:

```
(omod TWO-PHASE-COMMIT-WITH-MESSAGE-LOSS is
 including TEST-2PC .

 vars O O' : Oid .   var MC : MsgCont .

 crl [lose-abortCommit] :
     msg MC from O to O'
    =>
     (none).Configuration
    if MC == abort or MC == commit .
endom)
```

Let's test our new specification:

```
Maude> (frew init .)

result Configuration :
  < "a" : 2PCDB | state : ready, updated : false, ... >
  < "b" : 2PCDB | state : abort, updated : false, ... >
  < "c" : 2PCDB | state : ready, updated : false, ... >
  < "d" : 2PCDB | state : abort, updated : false, ... >
  < "e" : 2PCDB | state : ready, updated : false, ... >
```

Again, this looks very promising. Let's now check whether it is possible to reach an inconsistent final state:

```
Maude> (search [1] init
                  =>!
                   < O:Oid : 2PCDB | updated : false >
                   < O':Oid : 2PCDB | updated : true >
                   C:Configuration .)

Solution 1
... ; O':Oid --> "a" ; O:Oid --> "e"
```

OK, it was indeed possible to reach an inconsistent state. However, we are interested in actually *exhibiting* a behavior leading to an inconsistent state for the following reasons:

- To ensure that the faulty behavior really corresponds to a behavior in the 2PC, and is not just an error in our model of 2PC.
- To learn about the flaw in the protocol.

At present, Full Maude cannot output the path leading to state found during a search. However, it is easy to transform our module into a *core Maude* module by just writing

```
(show all .)
q
```

in the end of the file which contains our specification. The `(show all .)` command returns the core Maude module equivalent to our Full Maude module. We can then output this core Maude module to a file by executing the Maude command

```
Linux> maude 2pc.maude > core-2pc.maude
```

Remove the introductory text in the file `core-2pc.maude` as well as the final `Bye`, and you have a core Maude module of your specification. You can now give core Maude the above search command (without the parentheses). Core Maude will then find the inconsistent state (in my case it was state number 2676):

```
state 2676, Configuration:
< "a" : 2PCDB | state : initial, updated : true, anyNegative : false, ... >
< "b" : 2PCDB | state : ready, updated : true, ... >
< "c" : 2PCDB | state : ready, updated : true, ... >
< "d" : 2PCDB | state : ready, updated : true, ... >
< "e" : 2PCDB | state : ready, updated : false, ... >
```

Maude's `show path 2676 .` command will exhibit the trace to this state. For purposes of space, I just show the output of the `show path labels` command:

```
Maude> show path labels 2676 .
prepareReq
ok
recOK
ok
recOK
ok
recOK
ok
recOK
commitAll
recCommit
recCommit
recCommit
lose-abortCommit
```

Not a big surprise: everybody should have committed, but `"e"` probably never got the `commit` message (since rule `lose-abortCommit` is in the path) and hence never dared to update its database.

Finally, we can perform one of the above search commands to check if there is a possibility that one node is in state `abort` while some other node has updated its database. Such a state could not be reached.

Exercise 184 *1. My specification actually does not* force *the coordinator to choose whether it is ready to commit or wants to abort. How can we change specification so that the coordinator must make such a choice?*

2. Modify the specification of 2PC so that it models the version of the protocol where each `commit` *or* `abort` *message is acknowledged.*

Acknowledgment

I thank Ragnar Normann for suggesting to model and analyze 2PC in Maude, and for kindly clarifying aspects and assumptions about the protocol.

Chapter 10

Requirements and (In)validation of Invariants

So far in Part II, we have used Maude to specify the *possible behaviors* of a system. Such a specification is called a *system specification*. A system specification should be complemented by a *requirement specification* (or *property specification*) which describes "global" properties that the system should satisfy. For example, while our Maude system specification of the dining philosophers models the possible behaviors of the system, the global requirements on the system may be properties such as "two philosophers can never grab the same chopstick at the same time," or "each philosophers must eat at least n times" or "in each possible behavior of the system, philosopher 2 must have eaten at least m times," etc.

In this chapter, we first explain different classes of "global" properties, and then show how we can use Maude's search facilities to analyze whether or not a system satisfies a particular class of such properties, namely *invariants*.

10.1 Global Properties of Systems

Examples of desired requirements of the different systems we have encountered so far are:

- two philosophers should never hold the same chopstick at the same time;
- in every final state the database should be consistent;
- sooner or later the desired sequence of strings will be received by the receiver in the sliding window protocol;
- throughout the run of the sliding window protocol, the receiver has received a *prefix* of the sender's initial list of strings;
- each philosopher should eat an unbounded number of times.

Of course, the fundamental question is: does a system, as given by its system specification, satisfy a given requirement?

Exercise 185 *Which of the above requirements are satisfied by the corresponding system specification?*

The ultimate goal is to let the computer check whether or not a system satisfies its requirement. Therefore, we must be able to formalize not only a system, but also its requirements. Then we can use *theorem provers* and *model checkers* to check whether or not a system satisfies its requirements. The other advantage of formalizing the requirements is that they are then made precise.

There are many different kinds of requirements one may want a system to satisfy, and therefore also many different possible formalisms to express such requirements. This chapter discusses different temporal properties, and focuses on *invariance*.

10.1.1 State-Based vs Action-Based Global Properties

First of all, we notice that the "global properties" may be stated in terms of the *states* in the system or in terms of the *actions* (or *events*) that are performed. Examples of state-based global properties may be:

- "in each state reachable from the initial state, the age of a person is greater than or equal to 0";
- "eventually, the value of the `msgsRcvd` attribute of the `Receiver` will equal the value of the `Sender`'s `msgsToSend` attribute in the initial state";
- "a `Person` in state `waitSep(`*spouse*`)` will eventually reach a state `separated(`*spouse*`)`"

Examples of *action-based* global requirements may be:

- "each `marriage`-event must be preceded by an `engagement`-event for the same couple";
- "no `Person` can perform more than one `baptism` action".

Properties could also be both state- and action-based, such as:

- "a `Person` in state `baptized` should never perform another `baptism` (or `hajj`) action";
- "a `shutDown` action must take place (soon!) after the water level value is dangerously low in a nuclear power plant".

We will focus on the *state-based* way of expressing global properties. This does not imply that event-based properties are not sometimes more suitable for expressing the desired properties of a system.

10.1.2 State Formulas

A *state formula* is a statement about *one* given state. It is *not* a statement about a state and its successor or predecessor states, or about the path leading to/from the state. In principle, we should be able to define a state formula P as a function

```
op P : State -> Bool [frozen (1)] .
```

where *State* denotes the sort of the states. For object-oriented specifications, *State* is the sort `Configuration`.

For example, "the `Steelers` have more point than the `Seahawks`", or "philosopher 2 is in state `eating`" are state formulas, since they talk about *one* state. On the other hand, the property "the points total in the state is 6 greater than in the previous state" is *not* a state formula, since it talks about *two* states, as well as about transitions.

Given that we know the initial state of the system, we also allow a state formula to mention this initial state. Therefore, we allow state formulas such as "in the state, the total numbers of points is at least as great as in the initial state" and "in the state, the receiver's `msgsRcvd` attribute value is a prefix of the sender's *initial* `msgsToSend` value."

10.2 Temporal Properties

In this section we describe some classes of temporal properties.

10.2.1 Invariance: "Nothing Bad Will Happen"

A state formula P is an *invariant* in a system *w.r.t. an initial state* t_0 if and only P holds for each state t that can be reached from t_0. That is, $t_0 \longrightarrow t$ implies $P(t)$. An invariance property is also called a *safety* property[1] as it can be seen to mean that "nothing bad will happen."

Example 76 A useful invariant in the alternating bit protocol (w.r.t. all "normal" initial states) is that "the value of the receiver's `msgsRcvd` is a prefix of the sender's `msgsToSend` attribute *in the initial state*." ♠

Exercise 186 *Can you state some interesting state formulas that are invariant(s) in the* `POPULATION` *specification (w.r.t. appropriate initial states).*

Example 77 The property "the elements in the current state are the same as those in the initial state" is an invariant in the specification `SORT` on page 166. ♠

Exercise 187 *We now consider the two-phase commit protocol* without message loss *and with the appropriate initial state.*

[1] The class of safety properties usually includes other kinds of properties.

1. *Explain why the property "the database is consistent" is* not *an invariant.*

2. *Can you give a useful invariant for the two-phase commit protocol which addresses the issue of whether the components are updated or not.* Hint: *Include (some kind of) messages in your invariant, so that if follows from your invariant that the database is consistent when no messages are present.*

10.2.2 Guarantee: "Something Good Will Eventually Happen"

Invariants are very useful, but they only say that something bad will not happen. We also want to able to state that something good *must* happen sooner or later. A state formula P is a *guarantee* (or *liveness*) property if a P-state is reached in all possible computations from the initial state(s). That is, in *each* infinite sequence

$$t_0 \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \cdots \longrightarrow t_i \longrightarrow \cdots$$

and each finite sequence

$$t_0 \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \cdots \longrightarrow t_i \longrightarrow \cdots \longrightarrow t_n$$

of *one-step sequential rewrites* $t_k \longrightarrow t_{k+1}$, where, in the latter case, t_n cannot be further rewritten, and where t_0 is an/the initial state, there must be a state t_i with $i \geq 0$ for which the state property P holds. The property P is called a *guarantee* property since it is guaranteed that a P-state will be reached sooner or later, no matter how the rules are applied.

Exercise 188 *For which initial states of the coffee bean game of Exercise 116 on page 156 is the property "there are no black beans in the state" guaranteed? How about the property "there are no white beans in the state"?*

Exercise 189 *What is the desired guarantee property of the specification* `SORT`*?*

Exercise 190 *Explain why invariance and guarantee properties are not* dual. *That is, why is it* not *the case that*

the state formula P is guaranteed *if and only if* *the property not-P is* not *invariant*

(where not-P *holds in a state if and only if P does not hold in the state).*

Exercise 191 *What is the state formula that we want to guarantee in the two-phase-commit protocol? Can it be guaranteed in the setting without message losses?*

Exercise 192 *Explain why the desired property "the value of the* `Receiver`*'s* `msgsRcvd` *attribute equals the initial value of the* `Sender`*'s* `msgsToSend` *attribute" is not guaranteed in the alternating bit protocol specification. Under what assumptions is the property guaranteed?*

Exercise 193 *Consider the three solutions to the dining philosophers problem presented in Section 7.3 and assume that the initial state below is always the "standard" initial state for this problem.*

1. *Explain why the property "philosopher* `3` *is in state* `eating`*" is not guaranteed in any of the solutions.*
2. *In which solution(s) is the property "some philosopher is in state* `eating`*" guaranteed?*
3. *Is the property "two philosophers are in state* `eating`*" guaranteed in any of the solutions?*

As you may have noticed, it is quite often difficult to guarantee a desired property (such as "philosopher 2 is eating" in the deadlock-free scenario, or "all messages have arrived at the receiver") because we must also take into account weird behaviors (e.g., all messages are lost, or only philosopher 1 does anything). Therefore, to be able to guarantee a desired property, it is often necessary to assume some *fairness* requirements in the way the rules are applied. One such fairness condition would be that "if a rule is enabled infinitely many times for an object, then the rule must be applied to that object." With such a fairness assumption about the application of the rules, one could guarantee that eventually philosopher 2 will eat. There are many different notions of fairness, and mentioning them is beyond the scope of these lecture notes.

10.2.3 Reachability: "Something Bad Could Happen"

A state formula P is a *reachability property* of a specification $\mathcal{R}$ w.r.t. to the initial state t_0 if there exists a ground term t such that $\mathcal{R} \vdash t_0 \longrightarrow t$ and P holds in the state t. That is, it is possible to reach an P-state. The difference between a reachability and a guarantee property is that the guaranteed property must be reached in *all* possible computations, while for a reachability property it is enough that there exists a computation where the property is reached. There is a big difference between me being *guaranteed* to become a multimillionaire no matter which steps I take in my life, and the microscopic possibility that I will reach a multimillionaire status playing the national lottery each week.

Reachability is the *dual* property of invariance in the sense that

$$P \text{ is invariant} \qquad \text{if and only if} \qquad \text{not-}P \text{ is not reachable.}$$

Reachability properties are mostly used to find/show errors (namely, the possibility of reaching a "bad" state) in a specification.

Example 78 We had lots of problems trying to define a good way for a married couple to separate through the sending of letters. The specification is unlucky if the property "the state contains a message `separate(`p`)` and a person p whose status is `single`" is reachable. ♠

Exercise 194 *Which is the property whose reachability shows the error in the two-phase-commit protocol for databases? (Remember that a property of the form "... and the state is a* final *state" is* not *a state formula.)*

Exercise 195 *The reachability of which property would imply that the alternating bit protocol is incorrect?*

10.2.4 Response

An important task of reactive systems is to respond appropriately to stimuli from its "environment." Such systems must therefore satisfy *response properties*, where a pair (P_1, P_2) of state formulas is a response property which holds if and only if, for all possible behaviors, each term satisfying state property P_1 will be followed in zero or more steps by a term satisfying state property P_2. Typical response properties could be expressed as

- every message should be followed by acknowledgment,
- every state in which a `Person` has `status waitSep` is followed by a state in which the same `Person` has `status separated`,
- whenever the automobile control system senses a crash it should activate the airbags (hopefully in not too many steps), and
- an `ls` command in an operating system should be followed in zero or more steps (some other process could get a share of processing time in-between) by a listing of the files in the current working directory.

Exercise 196 *Assume that the initial state t_0 satisfies state formula P_1. Explain why it is still not the case that "P_2 is guaranteed" and "(P_1, P_2) is a response property" are not the same for this initial state. Does one of these* imply *the other?*

10.2.5 Other Kinds of Properties

We mention briefly some other kinds of properties a system may satisfy.

Stability. A state formula S is *stable* if the property never stops holding once it first holds. The property "the value of the `Receiver`'s `msgsRcvd` attribute equals the *initial* value of the `Sender`'s `msgsToSend` attribute" is a crucial property which is stable in the alternating bit protocol, which continues "working" even after all the messages have been received by the `Receiver`. The stability property ensures that this positive result will not be destroyed by the continued actions of the protocol.

Until. A system satisfies the *until property* U_1 until U_2 if, in all computations, U_2 will hold sooner or later, and U_1 holds all the way until U_2 holds.

Other combinations. The properties can be combined in all kinds of ways. Until and stability properties may be combined so that U_2 is stable once it holds.

Exercise 197 *Explain why guarantee properties can be regarded as special cases of until properties.*

10.2.6 Deadlocks: Unwanted Irreducible States

An "unwanted" state from which the system cannot proceed is usually referred to as a *deadlock*. For instance, the stuck state in which the system cannot proceed because each philosopher has grabbed one chopstick each in the philosophers example is clearly an "unwanted" non-rewritable state, and is therefore referred to as a deadlock. However, the final states of e.g., the sorting algorithm and the two-phase-commit protocol are usually *not* referred to as deadlocks, because the system is not intended to proceed from these states. Furthermore, while our population system is not expected to terminate, the case when all the people have died is a "natural" terminated state and is usually not considered to be a deadlock.

10.2.7 Termination

We have mentioned before that *termination* is an important and sometimes desired property of systems such as the two-phase-commit protocol and the sorting algorithm. Termination means that there is no infinite sequence of one-step sequential rewrites of ground terms.

10.3 Temporal Logic

I promised in the beginning of this chapter that we would be able to express the different kinds of requirements formally. Typical requirements, such as invariance, guarantee, stability, until, and reachability properties, can often be expressed in different kinds of *temporal logics* (such as linear temporal logic, branching-time temporal logic ("computation tree logic"), CTL*, μ-calculus, etc.).

Maude comes with a *model checker* for *linear temporal logic*. It is therefore possible to both express temporal requirements of a specification, and to check in Maude whether the system satisfies the requirements, *when the state space reachable from the initial state is fine*. In this course we will not enter into the realms of temporal logic. Instead, we will study how Maude's search facilities can be used to analyze satisfaction of invariant properties.

10.4 Analyzing Invariants

It is fairly easy to analyze whether a state formula P is an invariant of a system $\mathcal{R}$ w.r.t. initial state t_0: Just search for one state that is reachable from the initial state and that does *not* satisfy the state formula P. For example, to check the invariant that the receiver's list of received messages is a prefix of the sender's initial list of messages to send, we just search for a state where the receiver's list is *not* such a prefix.

What are the results? Of course, if the search for a state satisfying "not P" finds such a state, then the property P is *not* an invariant in the system w.r.t. initial state t_0. Furthermore, if the state space reachable from T_0 is finite, and the search command terminates terminates without finding a counterexample, then the property is an invariant w.r.t. t_0. However, if the number of states that can be reached from t_0 is infinite, then:

- if P is not an invariant w.r.t. t_0, then the search will eventually return a state;
- if P *is* an invariant w.r.t. t_0, then the search command will never terminate.

Therefore, if the number of states reachable from the initial state is infinite, we can *never* conclude from a Maude search that a state formula P really is an invariant, since we cannot know whether a counterexample would have turned up if we only let the search run for a couple of minutes more. The only thing we can conclude from a search which has gone on for a long time without finding a "*non-P*"-state is that our confidence (or hope) that P is really invariant has increased.

Exercise 198 *In this exercise we analyze the coffee bean game in Section 5.6.3 on page 156. Given the following properties:*

i *"the state has 8 beans"*

ii *"the state has has* odd *number of beans"*

iii *"the state has an odd number of black beans if and only if the initial state has an odd number of black beans"*

iv *"if the state has 5 beans then no following state will have more than 5 beans"*

v *"the state has an* even *number of white beans"*

vi *"if the state has an odd number of black beans, then we will end up with one (black) bean"*

vii *"the state has 8 or fewer beans"*

viii *"the state has an even number of black beans"*

1. *Which of the above properties are state formulas according to our definition?*
2. *Which of the state formulas above are invariants w.r.t. the initial state given in Exercise 116?*
3. *Which of the state formulas above are invariants for all possible initial states?*
4. *For each of the state formulas above, give the (largest) set of initial states for which the formula is invariant.*
5. *Use Maude's search command to check, for each state formula above, whether the state formula is an invariant for the initial state given in Exercise 116 and for the initial state in Exercise 132 on page 178. (You may need to define some helpful auxiliary functions.)*

Exercise 199 *We can analyze invariance and (obviously) reachability requirements using Maude's search command. Explain why guarantee requirements cannot be analyzed using Maude's search command. How about response, stability, and until requirements?*

Using Maude's search command to analyze invariants is very useful, but:

1. we can only analyze invariance w.r.t. *single* initial states, and not with respect to *sets* of initial states, and

2. we cannot prove that a state formula is an invariant if the reachable state space is infinite.

We can prove "by hand" that a state formula P is an invariant w.r.t. to *set* of initial states I by showing that:

- each state initial state $t_0 \in I$ satisfies P; and
- for each rewrite rule r, if $t \longrightarrow t'$ is a one-step rewrite using the rule r for ground terms t, t' such that t satisfies P, then t' must also satisfy P.

For example, consider the (nonterminating) specification of a football game in Section 5.6.1. We want to prove that the state property "the total number of points in the state is greater than 10" is an invariant for *all* initial states where the points total is already at least 11. It is entirely obvious that the property is indeed an invariant for all such states, but both because we are dealing with an infinite number of initial states, and because an infinite number of states is reachable from each initial state, we cannot do much with Maude search. However, we can indeed see that the property holds for all the desired initial states, and that the application of any rewrite rule preserves the property.

Exercise 200 *Which of the state formulas in Exercise 198 are invariants for* all *the following initial states:*

1. *The initial state given in Exercise 116.*
2. *Any initial state with an odd number of black beans.*
3. *Any sequence of coffee beans.*
4. *Any sequence with less than 9 coffee beans.*
5. *Any initial state with an even number of black beans.*

Use "hand proofs" to prove that the state formulas really are invariants for the respective sets of initial states.

Chapter 11

Modeling and Analyzing the Needham-Schroeder Public-Key Authentication Protocol in Maude

In communicating distributed systems one needs sometimes to *authenticate* a *connection* between two agents. An Internet bank needs to be sure that the remote gentleman posing as "Peter" is not some intruder Walker pretending to be Peter. Likewise, a field commander in a high-tech war needs to be sure that he has a connection with Pentagon and not Mr. Hussein. It is the role of an *authentication protocol* to provide (a description of a way of getting) such assurance.

In this chapter we model and analyze the *Needham-Schroeder public-key authentication protocol* (NSPK) [75] from 1978 in Maude. The NSPK protocol has been extensively used and aims to provide *mutual authentication*, after which some session involving the exchange of security-critical data can take place. The ten million dollar question is whether it is possible to break the protocol. That is, is there a *possibility* that some *intruder* can fool the bank (or the field commander) into thinking that it has trusted connection with someone while it in reality is connected to the intruder?

We first model and test the NSPK protocol without intruders. To analyze whether some intruder can successfully impersonate another agent, we then model *all* possible behaviors of an intruder and analyze whether an undesired state can be reached. We will *not* manually analyze the protocol and try to fine-tune the intruder model to exploit potential weaknesses in the protocol. Such manual analysis and hard thinking was probably already done in 1978 and the following years. The publication and subsequent use of the protocol, as well as the many references to it, indicate that it may not be trivial to "design" a perfect intruder. It also implies that *if* the protocol can be broken, it may well be because of some very unexpected behavior by the intruder.

If we can prove that the protocol is safe with an intruder who can do *everything*, then the protocol really *is* safe. If we on the other hand only analyze the protocol with respect to a particular "smart" intruder, we can only show that the protocol is safe with respect to *that* intruder, but we cannot guarantee that some smarter intruder could not break the protocol.

The description of the protocol is taken from [61, 62, 68]. The first modeling and analysis of the NSPK protocol in Maude was done by Denker, Meseguer, and Talcott [19]. The specification and analysis presented here differ from theirs.

11.1 Why Model and Analyze NSPK in Maude?

The modeling and analysis of the NSPK protocol in Maude illustrates some of the potential benefits of modeling and analyzing distributed systems in Maude, such as:

- In contrast to an informal specification, which is usually described by a combination of natural language and pseudo-code, a Maude specification is a *formal* (mathematical) *model* of the protocol which can be subjected to mathematical analysis.
- The protocol can be tested and analyzed in different ways using Maude's rewrite, search, and temporal logic model checking commands, as well as by user-defined analysis strategies. These capabilities are invaluable to analyze a complex system.
- A Maude module is a *precise* mathematical object whose only "underlying assumptions" are the deduction rules of rewriting logic. Everything else must be modeled. An informal specification is usually *ambiguous* and contains *implicit assumptions*. The implicit assumptions are assumptions that the designer make without spelling them out, either because she thinks that they are obvious or because she does not consider all possible settings. Since a Maude model is a precise mathematical object[1], ambiguities have to be resolved. Likewise, since the only "assumptions" of a Maude module are the deduction rules of rewriting logic, all implicit assumptions must be made explicit in the Maude specification. The assumptions may be self-evident to people working in a certain field at a certain time, but may not automatically be assumed by others or years later. For example, the descriptions of the NSPK protocol I have read do not say anything about
  - whether messages can be lost (or corrupted);
  - whether message delivery is assumed to be ordered or unordered;
  - what an agent should do when it receives a message it did not expect (should it ignore the message? understand the message as a break-in attempt and not further trust the receiver of the message?);
  - whether two nodes can initiate contact with each other at the same time, or whether an agent can be either *only* an initiator or *only* a responder in the establishment of an authenticated connection between them?

  You can answer these questions by just inspecting the following Maude specification. There is no need to read security books to guess under what assumption the protocol is supposed to work.
- Larger software systems usually consist of different "aspects" or "components" from different fields of computer science. A distributed banking database may e.g. use the

[1] The mathematical object specified by a rewriting logic specification is the *initial model* of the specification [69].

two-phase commit protocol (or the three-phase commit protocol) to keep the database consistent, the *alternating bit protocol* to ensure reliable communication between components, and some *security* protocol to ensure that only the desired user(s) may access certain data in the database. Each of the fields of *database*, *data communication*, and *data security* theory have their own notations, logics, implicit assumptions, and so on. An expert in one field will have a hard time understanding other parts of the software system and will have a hard time communicating her ideas. Maude seems to be a fairly intuitive formalism in which a wide range of concurrent systems can be naturally represented. In this course we have looked at nontrivial database, communication, and security protocols in Maude. My hope is that you can understand these protocols and their implicit assumptions by studying their Maude specifications without having to read books on the different subjects, and without having to learn quite well three new specification formalisms. Maude can therefore be used as a general formalism in which many different kinds and aspects of distributed systems can be naturally modeled to overcome the Babel-like proliferation of languages, and to facilitate the interoperation between different components/aspects in a system.

11.2 Description of the Protocol

The NSPK protocol uses *public-key cryptography* [24, 88]. Each *agent* A has a *public key*, denoted PK_A. We assume that all agents know the public key of each agent.[2] Each agent A has a *private key*, denoted $PrvK_A$, which is *only known* by A. An agent which knows the key K can *encrypt* the data m with the key K. The data m encrypted with key K is written $\{m\}_K$. Data which has been encrypted with the public key PK_A can only be *decrypted* with the private key $PrvK_A$ of the same agent A, and vice versa[3].

It is assumed that the underlying cryptographic mechanisms are secure when examining the security of protocols [68], so

- no agent can successfully *guess* the value of a private key, and
- no agent can encrypt or decrypt with a (private) key whose value it does not know,

In short, if an agent A wants to send a message m which should only be understood by the `Bank`, A should send the message $\{m\}_{PK_{\texttt{Bank}}}$, because only `Bank` can decrypt this message.

Exercise 201 Digital signatures *(simplified). What should an agent* `Peter` *do if he wants to send a message* m *to the* `Bank` *and be sure that*

1. `Bank` *knows that the* content m *is the correct message, and*
2. `Bank` *knows that the message was really sent from* `Peter`*?*

[2] An agent can get the public key of another agent from a trusted *key server*, but we can abstract away such details.

[3] In some cryptosystems, the public key cannot be used to decrypt encrypted data. The Needham-Schroeder protocol does not use public keys for decryption and is therefore supposed to work also under these circumstances.

(It does not matter if somebody else overhears *the content m as it says "transfer 1000$ from* `Peter`*'s account to* `Lizzie`*'s account," but the* `Bank` *should trust the content and the sender.)*

Nonces are "freshly" generated random numbers to be used in a single run of the protocol. It is assumed that these numbers are generated in such a way that they cannot be guessed by other agents. A nonce generated by an agent A is denoted N_a below.

The NSPK protocol is described as follows in [62]:

$$\begin{array}{lll} \text{Message 1.} & A \rightarrow B: & A.B.\{N_a.A\}_{PK(B)} \\ \text{Message 2.} & B \rightarrow A: & B.A.\{N_a.N_b\}_{PK(A)} \\ \text{Message 3.} & A \rightarrow B: & A.B.\{N_b\}_{PK(B)} \end{array}$$

The agent A is the *initiator* who seeks out to establish a communication session with the *responder* agent B.

In the first step, A generates the (unguessable) nonce N_a, adds her[4] identity A, encrypts this concatenation $N_a.A$ with the public key of B, and sends this encrypted message, together with her own and B's name (unencrypted) to B. When B receives this first message, he decrypts the encrypted part using his private key $PrvK(B)$ to obtain the nonce N_a. Only A and B know the value of N_a at this stage, even if there are eavesdroppers "listening" to the messages being transmitted in the network. (Why?)

The responder B then generates his own nonce N_b, and returns the nonce N_a along with the new nonce N_b, encrypted with the public key of A. In addition, B sends the sender and receiver names B and A unencrypted. When A receives this Message 2 she can decrypt it with her private key to read both N_a and N_b. It seems that at this stage of the protocol run A should be assured that she is talking to B while B cannot be sure that he is talking to A.

Exercise 202 *Assume that a system may have malicious intruders who can send fake messages but cannot guess private keys and nonces. After A has received Message 2 in a protocol run,*

1. *why would it seem that A should be assured that she is talking to B?, and*
2. *why cannot B be sure he is talking to A?*

To convince B that he is talking to A, the initiator A encrypts the received nonce N_b with B's public key, and sends the message back to B (together with the receiver and sender names). Since only A could decrypt Message 2, only A and B know N_b, and when B receives $\{N_b\}_{PK(B)}$ he is convinced that only A could have sent this message. At the end of a protocol run A is convinced that she is talking to B and B is convinced that he is talking to A so they can now start to exchange all their secrets.

Exercise 203 *Does it seem necessary to encrypt N_b in Message 3? What do you think is the reason for this encryption?*

[4]For reasons I don't know, A is often called "Alice" and B is called "Bob" in cryptographic protocols.

□ Public-key cryptography is considered too slow for encrypting and decrypting large amounts of data. Two parties who wish to communicate huge secrets by sending and receiving encrypted data must agree on a *secret key* which only these two agents know. Once the agents know their common secret key, they can efficiently encrypt and decrypt data using symmetric-key encryption techniques. Public-key cryptography is often used to establish secret keys between pairs of agents.

Exercise 204 *Can you indicate how the NSPK protocol can be extended to establish a secret key between two (mutually authenticated) agents?*

□

11.2.1 Ambiguities and Other Uncertainties

The protocol as described above seems only to concern *one* run with a clearly defined initiator and responder. There should be more than two agents in a real system.[5] How should the protocol be extended to that setting? If there are more than two agents it could happen that an agent may play the role of initiator in one run of the protocol and play the role of responder in another run. If an agent can be both initiator and responder, then A could initiate a communication with B at the same time that B initiates a communication with A. What should happen in that case? The attentive reader recognizes that we have already encountered this problem as the *separation problem*.

As mentioned earlier in this chapter, the informal description does not say anything about whether it is assumed that message can be lost and/or reordered during transmission. Neither does the description say how an agent should handle an unexpected message. Furthermore, it is not obvious from the specification whether the responder should send Message 2 immediately when it receives Message 1, or if he can send it "later."[6] Which, if any, unstated assumptions are crucial for the correctness of the protocol? What assumptions do I make about all these issues? Check out the following Maude specifications and you will find the answers!

11.3 Modeling the Protocol in Maude

I allow more than two agents in the system in my Maude model of the NSPK protocol and therefore allow multiple runs, or *sessions*, of the protocol at the same time. I keep the distinction between *initiator* and *responder*, but also allow an agent to be both initiator and responder (in different runs of the protocol). For simplicity I only model the setting where an agent A can *initiate* at most one run of the protocol with the same responder. I allow two agents to simultaneously initiate contact with each other but to avoid further (separation-problem-like) complexity I don't treat this case in a special way but as two independent protocol sessions, so that in the end A and B will be "doubly assured" that they are talking to each other.

[5] Indeed, an agent *always* knows who she is talking to if there are only *two* agents in a system!

[6] The two cases are clearly equivalent when only a *single* run of the protocol is modeled. Is it possible in a multi-agent system that B reads Message 1 from A, then reads Message 1 from C, then sends the response Message 2 to C, and only thereafter sends Message 2 to A?

While the informal specification only describes a single run of the protocol, my slightly more complicated specification treats the interesting case with more than two agents and where many sessions can run simultaneously.

11.3.1 Modeling Nonces and Keys

A nonce is supposed to be a "random" number which, for the purposes of the analysis of this protocol, cannot be guessed and should be freshly generated for each run of the protocol. We can *abstract* from the actual numerical value of a nonce since we are only interested in whether an agent "knows" the value of the nonce or not. A nonce N_a is modeled by a term

$$\texttt{nonce}(A, i)$$

where i a number given to the nonce to separate it from the other nonces generated by A. The first nonce generated by A is represented by the term $\texttt{nonce}(A, \texttt{1})$, the second nonce by the term $\texttt{nonce}(A, \texttt{2})$, and so on. The beginning of my Full Maude specification of the NSPK protocol is therefore

```
(omod NSPK is
  protecting NAT .

  sort Nonce .
  op nonce : Oid Nat -> Nonce [ctor] .
```

> □ If someone insists on having numerical values as nonces, one could of course generate numbers instead of the nonces, or one could define `Nat` to be a subsort of `Nonce` and define the appropriate equalities `nonce(O, N) = ...`. However, such *over-specification* is unnecessary and makes the protocol more complicated (how do we make sure that no nonce is guessed?). A specification should be *as simple as possible, but not simpler*, and should therefore abstract from all details which are not necessary at the appropriate *level of abstraction*. It may well be that in some "lower level" key generating and exchange protocol one would need to model the actual key values, but that is neither necessary nor desirable in our case. Indeed, this kind of over-specification is directly undesirable because it *fixes* the values of the nonces so that the protocol is only defined for these values and a user cannot change them! In my protocol I don't fix any values of the nonces, so they may be "instantiated" with *any* values the user desires, making the above abstract specification more general than one with numeric nonce values. □

The public key of A is modeled by a term $\texttt{pubKey}(A)$:

```
  sort Key .
  op pubKey : Oid -> Key [ctor] .
```

It is not necessary to model the private keys since, for the purpose of analyzing the given protocol, we can assume that *only* the agent A can decrypt a ciphertext which was encrypted with the public key of A.

11.3.2 Modeling the Messages

The three messages in the protocol all have the form $O.O'.\{message\ content\}_K$ where *message content* is either a pair of two nonces, a pair of a nonce and an agent identifier, or just a nonce. This part of the message content is modeled by the following sort `MsgContent`:

```
sort MsgContent .
op _;_ : Nonce Oid -> MsgContent [ctor] .   *** Message kind "1"
op _;_ : Nonce Nonce -> MsgContent [ctor] . *** Message kind "2"
subsort Nonce < MsgContent .                *** Message kind "3"
```

(where we use '`;`' instead of '`.`' as concatenation operator).

The specification uses the following syntax for *encrypted* message contents for enhanced readability:

```
sort EncrMsgContent .
op encrypt_with_ : MsgContent Key -> EncrMsgContent [ctor] .
```

Finally, a message is equipped with the (presumed!) sender and receiver identities:

```
msg msg_from_to_ : EncrMsgContent Oid Oid -> Msg .
```

11.3.3 Modeling the Initiators

An agent which can *initiate* a run of the protocol is modeled as an object of the following class `Initiator`:

```
class Initiator | initSessions : InitSessions, nonceCtr : Nat .
```

The initiator needs to know the nonce it sent to the responder in Message 1, so that it can check whether this is the same nonce that it receives in Message 2. In our setting, where an initiator may be simultaneously involved in many runs of the protocol with different responders, the initiator must store information about the nonces of all these runs. In the attribute `initSessions` an initiator A stores such information in a multiset of elements of the following three kinds:

- `notInitiated`(B) indicates that A can/will initiate contact with B but has not yet done so;
- `initiated`(B, N) indicates that A has sent Message 1 to B with nonce N and is waiting for Message 2 from B; and
- `trustedConnection`(B) indicates that A has established (what she thinks is) an authenticated connection with B.

The data type representing this kind of information is defined as follows:

```
sorts Sessions InitSessions .
subsort Sessions < InitSessions .
op emptySession : -> Sessions [ctor] .
op __ : InitSessions InitSessions -> InitSessions
                                        [ctor assoc comm id: emptySession] .
op __ : Sessions Sessions -> Sessions [ctor assoc comm id: emptySession] .

op notInitiated : Oid -> InitSessions [ctor] .
op initiated : Oid Nonce -> InitSessions [ctor] .
op trustedConnection : Oid -> Sessions [ctor] .
```

The attribute `nonceCtr` is just a counter which gives the index to the next nonce generated by the object.

The following variables are used in the definition of the initiator:

```
vars A B : Oid .
vars M N : Nat .
vars NONCE NONCE' : Nonce .
var IS : InitSessions .
```

The first rule models the sending of Message 1. The agent `A` has a `notInitiated(B)` element in its `initSessions` attribute which indicates that it is interested in establishing an authenticated connection with `B`. The initiator generates a fresh nonce `nonce(A, N)`, encrypts this nonce together with its identifier with the public key of `B` (`encrypt ... with pubKey(B)`), and adds its own and `B`'s name (`msg ... from A to B`) to this message and sends it out into the configuration. Agent `A` must remember that it has `initiated` contact with `B` with nonce `nonce(A, N)` and must also increase its nonce counter. All this happens in the following rule:

```
rl [start-send-1] :
   < A : Initiator | initSessions : notInitiated(B) IS, nonceCtr : N >
    =>
   < A : Initiator | initSessions : initiated(B, nonce(A, N)) IS,
                     nonceCtr : N + 1 >
   msg (encrypt (nonce(A, N) ; A) with pubKey(B)) from A to B .
```

The next rule models the reception of Message 2 from, and the sending of Message 3 to, an agent `B` who sent a pair of nonces encrypted with `A`'s public key. If the first nonce (`NONCE`) in the message received (and decrypted) by `A` is the same as the nonce stored in `A`'s `initSessions` attribute for `B`, the agent `A` figures out that it has established an authenticated connection with `B`, and sends Message 3 (`B`'s nonce (`NONCE'`) encrypted with `B`'s public key) to `B`:

```
rl [read-2-send-3] :
   (msg (encrypt (NONCE ; NONCE') with pubKey(A)) from B to A)
   < A : Initiator | initSessions : initiated(B, NONCE) IS >
    =>
   < A : Initiator | initSessions : trustedConnection(B) IS >
   msg (encrypt NONCE' with pubKey(B)) from A to B .
```

11.3.4 Modeling the Responders

The agents which play the roles of responders in runs of the protocol can be modeled by objects of a class `Responder` in the same style as the initiators:

```
class Responder | respSessions : RespSessions, nonceCtr : Nat .
```

The attribute `respSessions` keeps track of the sessions in which the agent is responder, where a value `responded`(A, N) means that the agent has received Message 1 from A and has responded by creating its own nonce N and has sent Message 2 to A with this nonce:

```
sort RespSessions .
subsort Sessions < RespSessions .
op __ : RespSessions RespSessions -> RespSessions
                                      [ctor assoc comm id: emptySession] .

op responded : Oid Nonce -> RespSessions [ctor] .
```

The first responder rule models the reception of Message 1 from `A`. The test `not A inSession RS` ensures that the responder `B` is not already a responder in a protocol run with `A`. When the responder receives the message, it creates its own nonce (`nonce(B, N)`) and sends this nonce together with the received nonce (`NONCE`), appropriately encrypted, back to `A`:

```
var RS : RespSessions .

crl [read-1-send-2] :
    (msg (encrypt (NONCE ; A) with pubKey(B)) from A to B)
    < B : Responder | respSessions : RS, nonceCtr : N >
     =>
    < B : Responder | respSessions : responded(A, nonce(B, N)) RS,
                      nonceCtr : N + 1 >
    msg (encrypt (NONCE ; nonce(B, N)) with pubKey(A)) from B to A
    if not A inSession RS .

op _inSession_ : Oid RespSessions -> Bool .

eq A inSession emptySession = false .
eq A inSession (trustedConnection(B) RS) = (A == B) or (A inSession RS) .
eq A inSession (responded(B, NONCE) RS) = (A == B) or (A inSession RS) .
```

The second, and last, responder rule models the reception of Message 3 with the expected nonce from `A`:

```
rl [read-3] :
   (msg (encrypt NONCE with pubKey(B)) from A to B)
   < B : Responder | respSessions : responded(A, NONCE) RS >
    =>
   < B : Responder | respSessions : trustedConnection(A) RS > .
```

11.3.5 Modeling Agents which can be Both Initiators and Responders

My Maude specification of the NSPK protocol concludes with the definition of agents who may be both initiators and responders. Such agents are modeled by objects of the class `InitiatorAndResponder` which is just a subclass of the classes `Initiator` and `Responder` and therefore inherits the union of the attributes of these two superclasses, and well as the rewrite rules associated to these classes:

```
  class InitiatorAndResponder .
  subclass  InitiatorAndResponder < Initiator Responder .
endom)
```

Exercise 205 *Answer the following questions about ambiguities and implicit assumptions of the protocol by inspecting the Maude specification given above:*

1. *Is it assumed that messages can get lost or be corrupted during transmission?*
2. *Is it assumed that the underlying message transmission infrastructure provides "ordered" message delivery between pairs of objects?*
3. *Does a responder always* immediately *send a Message 2 when receiving Message 1, or can the responder "delay" his response to the initiator?*
4. *How does the protocol deal with "unexpected" messages? Also show that it possible to have a (messageless) initial state which will lead to the generation of undesired messages.*

Exercise 206 *Modify the specification so that unwanted messages are removed from the state.*

11.3.6 Executing the NSPK Specification

The following initial state `init` is used to test whether the specification works correctly in the absence of intruders (whose behavior is not yet modeled):

```
(omod TEST-NSPK is
  including NSPK .
  including STRING .

  subsort String < Oid .

  op init : -> Configuration .
  eq init =
        < "a" : Initiator | initSessions : notInitiated("b"), nonceCtr : 1 >
        < "b" : Responder | respSessions : emptySession, nonceCtr : 1 > .
```

The `initSessions` attribute of an initiator should contain an element `notInitiated(`B`)` for each of the responders B with which the initiator wants to communicate.

A first execution of this initial state checks whether `"a"` and `"b"` reach a state of mutual authentication in some arbitrary behavior:

```
Maude> (rew init .)

result Configuration :
  < "a" : Initiator | nonceCtr : 2, initSessions : trustedConnection("b") >
  < "b" : Responder | nonceCtr : 2, respSessions : trustedConnection("a") >
```

This first execution went well, so one may as well examine all possible final states reachable from `init`:

```
Maude> (search init =>! C:Configuration .)

Solution 1
C:Configuration <-
 < "a" : Initiator | nonceCtr : 2, initSessions : trustedConnection("b") >
 < "b" : Responder | nonceCtr : 2, respSessions : trustedConnection("a") >

No more solutions.
```

After these encouraging results one can define a state with three agents:

```
  op init2 : -> Configuration .
  eq init2 =
       < "a" : InitiatorAndResponder | initSessions : notInitiated("c"),
                                       respSessions : emptySession,
                                       nonceCtr : 1 >
       < "Bank" : Responder | respSessions : emptySession, nonceCtr : 1 >
       < "c" : InitiatorAndResponder | initSessions :
                                        notInitiated("Bank") notInitiated("a"),
                                       respSessions : emptySession,
                                       nonceCtr : 1 > .
endom)
```

The agent `"c"` can play both the initiator and responder roles in different runs of the protocol. This initial state is interesting because `"a"` and `"c"` may try to initiate a session with each other simultaneously. Furthermore, `"a"` does *not* want to establish communication with `"Bank"`, so the `"Bank"` should never have a trusted connection with `"a"`. We search for all final reachable states:

```
Maude> (search init2 =>! C:Configuration .)

Solution 1
C:Configuration <-
  < "Bank" : Responder | nonceCtr : 2, respSessions : trustedConnection("c") >
  < "a" : InitiatorAndResponder | nonceCtr : 3,
                                  initSessions : trustedConnection("c"),
                                  respSessions : trustedConnection("c") >
  < "c" : InitiatorAndResponder | nonceCtr : 4,
```

```
                                    initSessions : (trustedConnection("Bank")
                                                    trustedConnection("a")),
                                    respSessions : trustedConnection("a") >

No more solutions.
```

All behaviors lead to the same expected final state.

11.4 Modeling the Intruders

Our analysis indicated that the Needham-Schroeder public-key authentication protocol works well in the *absence* of an *intruder* (also called *adversary*, *enemy*, *eavesdropper*, *attacker*, or *impersonator*). While it is certainly necessary that the protocol works well in the absence of intruders, the aim of the protocol is to provide mutual assurance that parties have established communication with the pretended agent in the *presence* of intruders. This section presents a model of an intruder which can be used to analyze the protocol in the presence of intruders.

Our "Dolev-Yao" intruder is assumed to have the following capabilities [68, 62] since messages may be transmitted over an unprotected network:

- *Overhear* and/or *intercept* any messages being passed in the system;
- Decrypt messages that are encrypted with its own public key so as to learn new nonces;
- Introduce new messages into the system, using nonces that the intruder knows;
- Replay any message the intruder has seen even if the intruder could not understand the contents of the encrypted part of the message. The intruder may of course change the plaintext parts of such seen messages (for instance to change the sender field of a message).

The intruders are assumed to be part of the computer network and can also take part in "normal" runs of the protocol [68]. (After the malicious intruder `"Walker"` has fooled the `"Bank"` to transfer `"Peter"`'s money to `"Walker"`'s account, the intruder must initiate a "normal" connection with the `"Bank"` to cash in all the money in his account.)

As mentioned above, I model *all possible* behaviors of an intruder, most of which will make no sense whatsoever. The reasons are that

- I am not an analyst of security protocols so I leave it to Maude to search all possibilities instead of trying to come up with an optimal intruder strategy; and
- if the protocol can withstand all possible attacks by a "brainless" intruder who can do everything then it can also withstand any "smarter" intruder, since the behaviors of a smart intruder is a subset of all possible intruder behaviors.

The only restriction I will assume is that messages which contain ciphertext encrypted with the public key of an agent A are also addressed to the same agent A.

11.4.1 The Maude Model of the Intruders

An intruder is also a "normal" actor, so that an intruder object has all the attributes of a normal agent. In addition, an intruder stores all information it has gathered in three attributes:

- `agentsSeen` contains the set of all agent identifiers known by the intruder;
- `noncesSeen` contains the set of all nonces the intruder knows; and
- `encrMsgsSeen` contains the set of all encrypted message contents which the intruder has seen without being able to decrypt.

The class `Intruder` with appropriate sorts `NonceSet` and `EncrMsgContentSet` and the variables used in the module are given as follows:

```
(omod NSPK-INTRUDER is
  including NSPK .
  including OID-SET .

  vars NONCE NONCE' : Nonce .
  vars A B I O O' O'' : Oid .
  var ENCRMSG : EncrMsgContent .
  var ENCRMSGS : EncrMsgContentSet .
  var N : Nat .
  var OS : OidSet .
  var NSET : NonceSet .
  var IS : InitSessions .
  var RS : RespSessions .
  var MSGC : MsgContent .

  *** Set of nonces seen:
  sort NonceSet .
  subsort Nonce < NonceSet .
  op emptyNonceSet : -> NonceSet [ctor] .
  op __ : NonceSet NonceSet -> NonceSet [ctor assoc comm id: emptyNonceSet] .
  eq NONCE NONCE = NONCE .

  *** Set of encrypted messages seen:
  sort EncrMsgContentSet .
  subsort EncrMsgContent < EncrMsgContentSet .
  op emptyEncrMsg : -> EncrMsgContentSet [ctor] .
  op __ : EncrMsgContentSet EncrMsgContentSet -> EncrMsgContentSet
                            [ctor assoc comm id: emptyEncrMsg] .
  eq ENCRMSG ENCRMSG = ENCRMSG .

  *** Remove duplicates from an OidSet:
  eq O ; O = O .
```

```
class Intruder | initSessions : InitSessions, respSessions : RespSessions,
                 nonceCtr : Nat, agentsSeen : OidSet,
                 noncesSeen : NonceSet,
                 encrMsgsSeen : EncrMsgContentSet .
```

(The equations above make a multiset into a set by removing multiple occurrences of the same elements.)

The following four rules model the "normal protocol behavior" in which the intruder behaves as a normal initiator or responder. The only difference compared to normal agents is that the intruder stores all information about nonces it has seen and used. Apart from that, the rules `I-send-1` and `I-rcv-2` are the same as the `Initiator` rules, and the other two rules are the same as those for responders:

```
rl [I-send-1] :
   < I : Intruder | initSessions : notInitiated(B) IS,
                    nonceCtr : N,
                    agentsSeen : OS, noncesSeen : NSET >
    =>
   < I : Intruder | initSessions : initiated(B, nonce(I, N)) IS,
                    nonceCtr : N + 1,
                    agentsSeen : OS ; B, noncesSeen : nonce(I, N) NSET >
   (msg (encrypt (nonce(I, N) ; I) with pubKey(B)) from I to B) .

crl [I-rcv-1] :
    (msg (encrypt (NONCE ; A) with pubKey(I)) from A to I)
    < I : Intruder | respSessions : RS, nonceCtr : N,
                     agentsSeen : OS, noncesSeen : NSET >
     =>
    < I : Intruder | respSessions : responded(A, nonce(I, N)) RS,
                     nonceCtr : N + 1,
                     agentsSeen : OS ; A,
                     noncesSeen : NSET NONCE nonce(I, N) >
    (msg (encrypt (NONCE ; nonce(I, N)) with pubKey(A)) from I to A)
    if not A inSession RS .

rl [I-rcv-2] :
   (msg (encrypt (NONCE ; NONCE') with pubKey(I)) from B to I)
   < I : Intruder | initSessions : initiated(B, NONCE) IS,
                    noncesSeen : NSET >
    =>
   < I : Intruder | initSessions : trustedConnection(B) IS,
                    noncesSeen : NSET NONCE' >
   (msg (encrypt NONCE' with pubKey(B)) from I to B) .

rl [I-rcv-3] :
```

```
    (msg (encrypt NONCE with pubKey(I)) from A to I)
    < I : Intruder | respSessions : responded(A, NONCE) IS >
     =>
    < I : Intruder | respSessions : trustedConnection(A) IS > .
```

The next rule models the case where the intruder *overhears* a message which is encrypted with another agent's public key. Therefore, the intruder cannot decrypt the message and the intruder can only store the *encrypted* message content and the sender and receiver names:

```
crl [overhear-but-not-understand] :
    (msg (encrypt MSGC with pubKey(O)) from O' to O)
    < I : Intruder | agentsSeen : OS, encrMsgsSeen : ENCRMSGS >
     =>
    < I : Intruder | agentsSeen : OS ; O ; O',
                     encrMsgsSeen : (encrypt MSGC with pubKey(O)) ENCRMSGS >
    (msg (encrypt MSGC with pubKey(O)) from O' to O)
    if O =/= I .
```

The next rule models the case where an intruder *intercepts* a message it cannot decrypt:

```
crl [intercept-but-not-understand] :
    (msg (encrypt MSGC with pubKey(O)) from O' to O)
    < I : Intruder | agentsSeen : OS, encrMsgsSeen : ENCRMSGS >
     =>
    < I : Intruder | agentsSeen : OS ; O ; O',
                     encrMsgsSeen : (encrypt MSGC with pubKey(O)) ENCRMSGS >
    if O =/= I .
```

Exercise 207 *What is the difference between the two rules above?*

The next three rules model the reception of a message sent to the intruder which the intruder will just discard after extracting the available information. This could be because some other intruder sent a fake message or because the intruder does not want to continue a normal run of the protocol with an agent.[7]

```
rl [intercept-msg1-and-understand] :
   (msg (encrypt (NONCE ; A) with pubKey(I)) from O to I)
   < I : Intruder | agentsSeen : OS, noncesSeen : NSET >
    =>
   < I : Intruder | agentsSeen : OS ; O ; A,
                    noncesSeen : NSET NONCE > .

rl [intercept-msg2-and-understand] :
   (msg (encrypt (NONCE ; NONCE') with pubKey(I)) from O to I)
```

[7] An intruder may want to initiate a run of the protocol with another agent to obtain its nonce.

```
    < I : Intruder | agentsSeen : OS, noncesSeen : NSET >
     =>
    < I : Intruder | agentsSeen : OS ; O,
                     noncesSeen : NSET NONCE NONCE' > .

rl [intercept-msg3-and-understand] :
    (msg (encrypt NONCE with pubKey(I)) from O to I)
    < I : Intruder | agentsSeen : OS, noncesSeen : NSET >
     =>
    < I : Intruder | agentsSeen : OS ; O,
                     noncesSeen : NSET NONCE > .
```

After having modeled the spying ("intelligence gathering") capabilities of an intruder, we now model its capabilities of sending *any kind of fake message* around the system, using the agent identities, the nonces, and the encrypted message contents it knows.

The following rule models the case where an intruder sends a fake message with a content that (s)he has previously stored but could not decrypt. Since the content is encrypted with `B`'s public key, the fake message will be sent to `B`. The "sender" may be *any* agent whose identity the intruder knows (including the intruder's own identity):

```
crl [send-encrypted] :
    < I : Intruder | encrMsgsSeen : (encrypt MSGC with pubKey(B)) ENCRMSGS,
                     agentsSeen : A ; OS >
     =>
    < I : Intruder | >
    (msg (encrypt MSGC with pubKey(B)) from A to B)
    if A =/= B .
```

(A sceptic reader may wonder whether the intruder knows that the encrypted message is encrypted with the public key of `B` since that knowledge may not be given from the ciphertext itself. In that case, the intruder could have stored this information when it overheard/intercepted the message since it could read the "receiver" part of the message which was given in plaintext. Nevertheless, in order not to have to worry about these aspects, one *could* define a more general rule

```
crl [send-encrypted] :
    < I : Intruder | encrMsgsSeen : ENCRMSG ENCRMSGS,
                     agentsSeen : A ; B ; OS >
     =>
    < I : Intruder | >
    (msg ENCRMSG from A to B)
    if A =/= B /\ B =/= I .
```

where nothing is assumed known about the ciphertext.)

Finally, an intruder may compose a message of any of the three kinds Message 1, Message 2, and Message 3 using the nonces and agent identifiers it knows. (The intruder is *impersonating*

another agent if the "sender" field in the generated message is different from the intruder's own identity.)

```
  crl [send-1-fake] :
      < I : Intruder | agentsSeen : A ; B ; OS,
                       noncesSeen : NONCE NSET >
       =>
      < I : Intruder | >
      (msg (encrypt (NONCE ; A) with pubKey(B)) from A to B)
      if A =/= B /\ B =/= I .

  crl [send-2-fake] :
      < I : Intruder | agentsSeen : A ; B ; OS,
                       noncesSeen : NONCE NONCE' NSET >
       =>
      < I : Intruder | >
      (msg (encrypt (NONCE ; NONCE') with pubKey(A)) from B to A)
      if A =/= B /\ A =/= I .

  crl [send-3-fake] :
      < I : Intruder | agentsSeen : A ; B ; OS,
                       noncesSeen : NONCE NSET >
       =>
      < I : Intruder | >
      (msg (encrypt NONCE with pubKey(B)) from A to B)
      if A =/= B /\ B =/= I .
endom)
```

I believe that these rules model all possible behaviors of an intruder.

11.5 Analyzing the Protocol in Maude

The following module defines an initial state with an honest initiator, `"Peter"`, an honest responder, `"Bank"`, and a "brainless" and malicious intruder, `"Walker"`. The agent `"Peter"` is (for obvious reasons) *not* interested in initiating contact with the `"Bank"`, while the malicious `"Walker"` is interested in such contact, possibly for cashing in the money he may have gained by impersonating some unsuspecting agent such as `"Peter"`.

```
(omod TEST-INTRUSION is
  including NSPK-INTRUDER .
  including STRING .

  subsort String < Oid .

  op intruderInit : -> Configuration .
```

```
  eq intruderInit =
       < "Peter" : Initiator | initSessions : notInitiated("Walker"),
                               nonceCtr : 1 >
       < "Bank" : Responder | respSessions : emptySession, nonceCtr : 1 >
       < "Walker" : Intruder | initSessions : notInitiated("Bank"),
                               respSessions : emptySession,
                               nonceCtr : 1,
                               agentsSeen : "Bank" ; "Walker",
                               noncesSeen : emptyNonceSet,
                               encrMsgsSeen : emptyEncrMsg > .
endom)
```

The protocol can be considered broken if the `"Bank"` thinks that it has established an authenticated connection with `"Peter"`, who did not desire to communicate with the `"Bank"`. The protocol is nonterminating since the intruder can repeatedly send fake messages. To check whether the protocol can be broken we must search for a state which is reachable from the initial state and where `"Bank"` thinks that it has established a connection with `"Peter"`:

```
Maude> (search [1]
           intruderInit =>*
              C:Configuration
              < "Bank" : Responder | respSessions : trustedConnection("Peter")
                                                    RS:RespSessions > .)
```

After waiting for some time I got the following result:

```
Abort
```

It is a promising sign that Maude could not find a bad state before aborting. Nevertheless, let's analyze this non-result a little bit more. The system I have specified is highly nondeterministic since an intruder can do so many different things at any time. Let us for a rough approximation assume that the intruder knows three nonces, three agents (including himself), and one message it could not decrypt. From such a state the intruder can generate 12 messages of kind 1, 24 messages of kind 2, and two messages of kind 3. That is, the intruder can send 38 messages in *any* such stage. Other actions, such as normal protocol runs, interception of messages, etc., may also happen at this time. This nondeterminism is the cost of our very general intruder model.

Maude performs searches in a *breadth-first* way, so that it first computes all states reachable in *one* step from the initial state, then it computes all states reachable in *two* steps, and so on. To avoid searching from states it has already seen, Maude caches *all* states it has seen in the search, together with the *rewrite path* leading to each state (which is used to answer `show path` commands). Maude employs sophisticated techniques to implement the search command efficiently.

The search can run out of memory despite all the fancy techniques for caching the states. Assume that it is possible to perform at least n different rewrite steps from any state. It is then possible to reach at least $n+1$ states in one rewrite step, n^2+n+1 states in two steps,

$n^3 + n^2 + n + 1$ states in three steps, ..., and $\Sigma_{i=0}^{d} n^i$ (which is greater than n^d) states in d steps. Assume furthermore than the NSPK protocol needs at least 10 steps to reach the "bad" state we are looking for, and that, say, on average 20 distinct rewrite steps can be taken from any reachable state (in our example we have fewer steps early on, and many more steps later on). The "computation tree" up to 10 rewrite steps would then contain more than 20^{10} states. Quite a lot of these more than *ten trillion* states would probably be the same state, but even if there were only about one million distinct states the search would run out of memory, and Maude would spend all its time swapping data between primary and secondary memory, choking both the search and the computer.

Mindful of these limitations, we try a more modest search. Is it possible to reach a state where the `"Bank"` thinks it has responded to a request from `"Peter"`?

```
Maude> (search [1] intruderInit =>*
             C:Configuration
             < "Bank" : Responder | respSessions :
                                    responded("Peter", N:Nonce)
                                    RS:RespSessions > .)

Solution 1
RS:RespSessions <- emptySession ;
N:Nonce <- nonce("Bank",1);  V#1:AttributeSet <- nonceCtr : 2 ;
C:Configuration <-
  < "Peter" : Initiator | nonceCtr : 2,
                          initSessions : initiated("Walker", nonce("Peter",1)) >
  < "Walker" : Intruder | nonceCtr : 1, initSessions : notInitiated("Bank"),
                          respSessions : emptySession,
                          encrMsgsSeen : emptyEncrMsg,
                          noncesSeen : nonce("Peter",1),
                          agentsSeen : ("Bank" ; "Peter" ; "Walker") >
  msg encrypt nonce("Peter",1); nonce("Bank",1)with pubKey("Peter")
      from "Bank" to "Peter"
```

While this result *could* be a bad sign, it is still a long way from impersonating `"Peter"` to *initiate* a contact with the `"Bank"` to actually have the `"Bank"` think that is has *established* an authenticated connection with `"Peter"`. The following command searches for a disastrous state from the result above:

```
Maude> (search [1]
  < "Walker" : Intruder | nonceCtr : 1, initSessions : notInitiated("Bank"),
                          respSessions : emptySession,
                          encrMsgsSeen : emptyEncrMsg,
                          noncesSeen : nonce("Peter", 1),
                          agentsSeen : ("Bank" ; "Peter" ; "Walker") >
  < "Peter" : Initiator | nonceCtr : 2,
                          initSessions : initiated("Walker", nonce("Peter",1)) >
  (msg encrypt nonce("Peter",1); nonce("Bank",1) with pubKey("Peter")
```

```
      from "Bank" to "Peter")
  < "Bank" : Responder | respSessions : responded("Peter", nonce("Bank", 1)),
                         nonceCtr : 2 >
 =>*
  C:Configuration
  < "Bank" : Responder | respSessions : trustedConnection("Peter")
                                        RS:RespSessions > .)
```

Maude actually returns an unwanted state (in 16 seconds) as a result of this search command:

```
rewrites: 1530601 in 16120ms cpu (16330ms real) (94950 rewrites/second)

Solution 1
RS:RespSessions <- emptySession ; V#1:AttributeSet <- nonceCtr : 2 ;
C:Configuration <-
  < "Peter" : Initiator | initSessions : trustedConnection("Walker"), ... >
  < "Walker" : Intruder | encrMsgsSeen :
                            encrypt nonce("Peter",1); nonce("Bank",1)
                               with pubKey("Peter"),
                          noncesSeen : (nonce("Bank",1) nonce("Peter",1)), ... >
  ...
```

This is a remarkable result, since it purports to show that there is a behavior from the initial state to a state in which the malicious intruder has fooled the `"Bank"` into thinking it is connected to `"Peter"`.

It is important to analyze the path leading to the state which broke the protocol for at least the following reasons:

- The search result invalidates the *Maude specification* of the NSPK protocol. The question is whether it also invalidates the protocol, or whether it is just the case that the Maude specification does not faithfully model the protocol? By analyzing the path leading to the undesired state one could check whether that path really corresponds to a behavior in the NSPK protocol described by the informal specification.
- To understand *how* the protocol could be broken. This may help us to design an improvement of the protocol which is not vulnerable to the attack demonstrated by the path.

Maude's search command can, as explained in Section 6.3, be followed by a `show path` command which gives us the rewrite path leading to the desired state. *Full Maude* does not yet have this capability. However, Full Maude has a `show all` command which returns the (core) Maude module corresponding to the "current" Full Maude module. We can then perform a (core) Maude search on this translated (core) Maude module followed by a `show path` command. Since I do not enjoy cutting-and-pasting large modules I used the following technique to quickly produce a (core) Maude module from the Full Maude specification presented in this chapter:

1. Add the lines

   ```
   (show all .)
   q
   ```

   in the file right after the end of the module to analyze. (The command 'q' ends the Maude session.)

2. Use Linux/Unix to redirect the output of the Maude session to a file `nspk-show-all.maude`:

   ```
   Linux> maude nspk.maude > nspk-show-all.maude
   ```

3. The file `nspk-show-all.maude` will contain the executable (core) Maude translation of the Full Maude specification of the protocol, as well as some welcome and farewell greetings. Remove these greetings and you have a (core) Maude specification of the protocol.

We then repeat the two searches in (core) Maude, first:

```
Maude> search [1]
         intruderInit =>*
            C:Configuration
            < "Bank" : Responder | respSessions :
                                     responded("Peter", N:Nonce)
                                     RS:RespSessions,
                                   ATTS:AttributeSet > .

Solution 1 (state 134)
...
```

The resulting state has number 134, so the path from the initial state to the state where the `"Bank"` has responded to the fake message can be obtained by the `show path` command:

```
Maude> show path 134 .
state 0, Configuration:
  < "Bank" : Responder | nonceCtr : 1, respSessions : emptySession >
  < "Peter" : Initiator | nonceCtr : 1, initSessions : notInitiated("Walker") >
  < "Walker" : Intruder | nonceCtr : 1, initSessions : notInitiated("Bank"),
                          respSessions : emptySession,
                          encrMsgsSeen : emptyEncrMsg, noncesSeen : emptyNonceSet,
                          agentsSeen : ("Bank" ; "Walker") >
===[rl ... [label start-send-1] .]===>
  < "Peter" : Initiator | nonceCtr : 2,
                          initSessions : initiated("Walker", nonce("Peter", 1)) >
  < "Walker" : Intruder | ... >
  < "Bank" : Responder | ... >
  msg encrypt nonce("Peter", 1) ; "Peter" with pubKey("Walker")
      from "Peter" to "Walker"
===[rl ... [label intercept-msg1-and-understand] .]===>
```

```
  < "Bank" : Responder | ... >
  < "Peter" : Initiator | ... >
  < "Walker" : Intruder | initSessions : notInitiated("Bank"),
                          noncesSeen : nonce("Peter", 1),
                          agentsSeen : ("Bank" ; "Peter" ; "Walker"), ... >
===[crl ... [label send-1-fake] .]===>
  < "Bank" : Responder | ... >
  < "Peter" : Initiator | >
  < "Walker" : Intruder | ... >
  msg encrypt nonce("Peter", 1) ; "Peter" with pubKey("Bank")
      from "Peter" to "Bank"
===[crl ... [label read-1-send-2] .]===>
  < "Bank" : Responder | nonceCtr : 2, respSessions :
                                        responded("Peter", nonce("Bank", 1)) >
  < "Peter" : Initiator | ... >
  < "Walker" : Intruder | ... >
  msg encrypt nonce("Peter", 1) ; nonce("Bank", 1) with pubKey("Peter")
      from "Bank" to "Peter"
```

It is easy to see what happened: `"Peter"` wants to communicate with `"Walker"`, who intercepts this message without responding. `"Walker"` has now seen `"Peter"`'s nonce and identity and fakes a message with these data to the `"Bank"`, pretending to be `"Peter"`.

Likewise, we perform the second search in (core) Maude:

```
Maude> search [1]
   < "Bank" : Responder | nonceCtr : 2,
                          respSessions : responded("Peter", nonce("Bank", 1)) >
   < "Peter" : Initiator | nonceCtr : 2,
                           initSessions : initiated("Walker", nonce("Peter", 1)) >
   < "Walker" : Intruder | nonceCtr : 1, initSessions : notInitiated("Bank"),
                           respSessions : emptySession,
                           encrMsgsSeen : emptyEncrMsg,
                           noncesSeen : nonce("Peter", 1),
                           agentsSeen : ("Bank" ; "Peter" ; "Walker") >
   msg encrypt nonce("Peter", 1) ; nonce("Bank", 1) with pubKey("Peter")
       from "Bank" to "Peter"
 =>*
     C:Configuration
     < "Bank" : Responder | respSessions : trustedConnection("Peter")
                                           RS:RespSessions,
                            ATTS:AttributeSet > .

Solution 1 (state 127211)
states: 127212  rewrites: 1943486 in 0ms cpu (13109ms real) (~ rewrites/second)
...
```

It was indicated above that a search could involve many states, and the search above encountered 127 212 *distinct* states just from the "intermediate" state from which the second search started until it found the first state satisfying the search pattern. We again use the `show path` command:

```
Maude> show path 127211 .
```

```
state 0:
  < "Bank" : Responder | nonceCtr : 2,
                         respSessions : responded("Peter", nonce("Bank", 1)) >
  < "Peter" : Initiator | nonceCtr : 2,
                          initSessions : initiated("Walker", nonce("Peter", 1)) >
  < "Walker" : Intruder | nonceCtr : 1, initSessions : notInitiated("Bank"),
                          respSessions : emptySession,
                          encrMsgsSeen : emptyEncrMsg,
                          noncesSeen : nonce("Peter", 1),
                          agentsSeen : ("Bank" ; "Peter" ; "Walker") >
  msg encrypt nonce("Peter", 1) ; nonce("Bank", 1) with pubKey("Peter")
      from "Bank" to "Peter"
===[crl ... [label overhear-but-not-understand] .]===>
state 1:
  < "Bank" : Responder |... >
  < "Peter" : Initiator | ... >
  < "Walker" : Intruder | encrMsgsSeen :
                            encrypt nonce("Peter", 1) ; nonce("Bank", 1)
                            with pubKey("Peter"), ... >
  msg encrypt nonce("Peter", 1) ; nonce("Bank", 1) with pubKey("Peter")
      from "Bank" to "Peter"
===[crl ... [label send-encrypted] .]===>
state 14:
  < "Bank" : Responder | ... >
  < "Walker" : Intruder | ... >
  < "Peter" : Initiator | initSession : initiated("Walker", nonce("Peter", 1)), ... >
  msg encrypt nonce("Peter", 1) ; nonce("Bank", 1) with pubKey("Peter")
      from "Walker" to "Peter"
  ...
===[rl ... [label read-2-send-3] .]===>
state 150:
  < "Bank" : Responder |  ... >
  < "Walker" : Intruder | ... >
  < "Peter" : Initiator | initSessions : trustedConnection("Walker"), ... >
  msg encrypt nonce("Bank", 1) with pubKey("Walker") from "Peter" to "Walker"
  ...
===[rl ... [label intercept-msg3-and-understand] .]===>
state 1501:
  < "Bank" : Responder | ... >
  < "Peter" : Initiator | ... >
  < "Walker" : Intruder | noncesSeen : (nonce("Bank", 1) nonce("Peter", 1)), ... >
  ...
===[crl ... [label send-3-fake] .]===>
state 14315:
  < "Bank" : Responder | respSessions : responded("Peter", nonce("Bank", 1)), ... >
  < "Peter" : Initiator | ... >
  < "Walker" : Intruder | ... >
  msg encrypt nonce("Bank", 1) with pubKey("Bank") from "Peter" to "Bank"
  ...
===[rl ... [label read-3] .]===>
state 127211:
  < "Bank" : Responder | respSessions : trustedConnection("Peter"), ... >
  < "Peter" : Initiator | ... >
```

```
< "Walker" : Intruder | ... >
...
```

Let's analyze this path and see if it makes sense in the original protocol or if I just made a fatal mistake in my Maude specification. Before the break, as they say all the time on TV here, we saw that `"Peter"` wanted to communicate with `"Walker"` and sent messages to `"Walker"` which `"Walker"` then used to impersonate `"Peter"` in front of the `"Bank"`.

The `"Bank"` had sent Message 2 to `"Peter"` in response to the fake request by `"Walker"`. In the first step above, `"Walker"` overhears this message, but can't decrypt it, since it is intended for `"Peter"`. Nevertheless, `"Walker"` stores this message and in the next step replays it for `"Peter"`, with its own sender-address. `"Peter"`, who is expecting a connection with `"Walker"` is happy to see this message and answers with a Message 3 to `"Walker"`, *where he replays the `"Bank"`'s nonce* with `"Walker"`'s public key! In this way, `"Walker"` has learned the `"Bank"`'s nonce which the `"Bank"` only thought that `"Peter"` could read. Once it knows this nonce of the `"Bank"`-`"Peter"`-connection, it fakes a Message 3, pretending to be `"Peter"`, to the `"Bank"` who is waiting for exactly this confirmation of its connection with `"Peter"`. The `"Bank"` is therefore convinced it is talking to `"Peter"`, and `"Walker"` could have transferred `"Peter"`'s money to his own account if there would have been any money there.

The behavior above can be described in a style more reminiscent of the informal specification as follows, where $S1$ stands "session 1" of the protocol, $S1.M1$ stands the sending of Message 1 in the session $S1$, the agents are abbreviated B, P, and W, and $W(P)$ stands for "W pretending to be P" or "W reading a message meant for P":

$$\begin{array}{lrcr} S1.M1: & P \rightarrow W & : & P.W.\{N_p.P\}_{PK(W)} \\ S2.M1: & W(P) \rightarrow B & : & P.B.\{N_p.P\}_{PK(B)} \\ S2.M2: & B \rightarrow W(P) & : & B.P.\{N_p.N_b\}_{PK(P)} \\ S1.M2: & W \rightarrow P & : & W.P.\{N_p.N_b\}_{PK(P)} \\ S1.M3: & P \rightarrow W & : & P.W.\{N_b\}_{PK(W)} \\ S2.M3: & W(P) \rightarrow B & : & P.B.\{N_b\}_{PK(B)} \end{array}$$

If one checks the two runs of the protocol (the $S1$'s and the $S2$'s) one can figure out that they are indeed runs of the protocol as described on page 276. It should therefore be no doubt that the NSPK protocol can be broken. Furthermore, the above analysis reveals how an intruder should behave to break the protocol.

Another observation from the rewrite path above is that the number of distinct states reached in the search increases by a factor of 10 for each level one goes down in the computation tree (states numbered 1, 14, 150, 1501, 14315, and 127211 were encountered). If this trend would hold with the additional steps from the first search, it would mean that over 120 *million* distinct states would be reached[8] and stored during the search, so it is not a big surprise that the search runs out of memory.

[8]There may be slightly fewer rewrites possible in the initial stages of the protocol, but still ...

11.6 Discussion and Context

The Needham-Schroeder public-key authentication protocol, which was published in 1978, has been used extensively and was considered secure for a long time. It is described in the *Handbook of Applied Cryptography* [68] from 1996 without any comments that it has been broken. The protocol was also proved "correct" in the absence of intruders in 1989 [7].

We have shown how one could use Maude to find a crucial fault in the protocol *without any experience* with security protocols and without having to think hard about how to break it.

The fault in the protocol was originally reported by Lowe in 1995 [61], which means that the error went undetected for 17 years. It is remarkable that the fault was found during *formal analysis*[9] using a tool for the process algebra CSP [62]. The break-in scenario reported by Lowe is the same as the one found during the Maude analysis presented in this chapter.

Denker, Meseguer, and Talcott modeled the protocol in Maude in 1998 and defined their own search strategy[10] using Maude's reflective capabilities [19]. They found the fault without splitting the search, but at the cost of analyzing the intruder more closely and cutting off the search slightly earlier than I do, so they encounter only about 38000 states during their search, compared to my 127000 states. I chose to model an entirely "brainless" intruder and searched for an unsophisticated state which did not require much understanding of the protocol. The only thinking I had to do was to split the search into two parts. That seems to be a reasonable price to pay for analyzing a protocol whose fault went undetected by security experts for 17 years.

The state space explosion helps to high-light the differences between *sequential* programs you may have studied in earlier courses and *concurrent* systems. You could pretty much analyze a sequential deterministic program by simulating a "run" of the program on a piece of paper by just simulating one instruction after another. The events in a distributed system may occur in many different orders, depending on the relative speeds of the different computers, on their loads, on the speed of message transmission, and so on. One has to take into account all the different orders in which events may occur when analyzing a distributed system. The combination of the many different orders in which the events of multiple runs of the protocol could occur and the nondeterminism caused by the highly nondeterministic behavior of the intruder makes the trivial-looking protocol hard to analyze. In the example presented in this chapter, one could reach more than 20 states in *one* rewrite step from a certain state, more than 400 states in just *two* steps, and so on. Trying to analyze the runs of such a system with pen and paper will quickly feel like a futile exercise. Good analysis techniques and tools are therefore crucial for understanding distributed systems.

To conclude, I hope that this chapter has illustrated how Maude can serve as a formalism in which many different distributed systems, such as security protocols, can be naturally represented, so that the protocol can be well understood even if you are not an expert in computer security. Furthermore, the chapter illustrated that Maude's search command is very useful to analyze distributed systems, but that we must complement the search with a modest amount of thinking to analyze complex systems such as the NSPK protocol.

[9] Exhaustive search of a formal specification, just as our search in Maude!

[10] Their analysis was performed before Maude was equipped with a search command.

11.7 The Corrected Protocol

Gavin Lowe not only presented a fault in the NSPK protocol but also suggested a modification of the protocol to make it secure. The idea is that the responder adds its own identity to the *encrypted* part of Message 2, so that the Message 2 part of the protocol becomes

$$\text{Message 2.} \quad B \rightarrow A \ : \quad B.A.\{N_a.N_b.B\}_{PK(A)}$$

Exercise 208 *Explain why the attack outlined in this chapter no longer works (or can be easily modified to work) in the modified protocol.*

Exercise 209 *Modify the Maude specification of the protocol to model the new version of the protocol. Define an initial state with one honest initiator, one honest responder, and one intruder, and try to search for a break in the protocol, possibly by splitting up the search. Can you break the modified protocol?*

Searching is ultimately a technique for discovering errors and to get a feel for a specification. Nothing is *proved* about a system if a search goes well, since it applies only to the *single* initial state from which the search was started. If you cannot break the protocol with three agents it does not necessarily mean that you cannot break it if you start with four agents instead. However, in this particular case, Lowe claims to have proved that *if* the (modified) protocol can be broken, then it can also be broken by a smaller system with one initiator, one responder, and one intruder [62]. Furthermore, each honest agent only needs to create *one* nonce. In other words, if you cannot break the protocol with three agents you cannot break it with 58 agents either. This means that if you can use search to show that there is no undesired state reachable from a three-agent state described above, then you have *proved* the specification correct. (The additional information that each agent only needs to generate one nonce can be used to make the specification terminating so that the protocol can be proved secure by a search.)

11.8 More on Search Strategies

The unsuccessful first search in this chapter motivates some more thinking about suitable ways of searching through the (often) *infinite* "computation tree" defined by the computations starting from a single initial state.

The two well-known classes of strategies for searching/traversing a tree are *breadth-first traversal* and *depth-first traversal*. I assume that the reader is familiar with these techniques.

A breadth-first traversal has the crucial property that the desired states we are looking for will *always* be found, as long as they are reachable from the initial state. Furthermore, a breadth-first search will find those states that can be reached in the fewest number of steps among all the possible states satisfying the search criteria.

Exercise 210 *Explain briefly why it is that breadth-first search has the two properties described above.*

The main disadvantage with breadth-first search is that a breadth-first traversal must always store at least the *frontier* of the tree it has searched so far. This made our search run out of memory.

One can choose whether or not to cache *all* states encountered during the traversal and ignore states which have already been visited. Maude chooses to do this. This choice intuitively seems like a sound choice despite the memory problems because:

- the number of states "above" the frontier is most often significantly smaller than the number of states in the frontier;
- a distributed system will probably have a lot of "repeated" states in the computation tree, so this approach will truncate the computation tree significantly; and
- termination of a search command is guaranteed when only a finite set of states is reachable from the initial state.

A *depth-first* traversal of the computation tree has the advantage that only the states in the path from the initial state to the "current" state, and not the frontier of the tree, need to be cached, so that serious memory problems are avoided. The devastating disadvantage of a depth-first traversal of an infinite computation tree is that the search may very easily run into an infinite path in which the desired states do not occur, so that the search may never find the desired states that are reachable from the initial state. Furthermore, it may not make much sense to cache all states encountered during the depth-first traversal to avoided repeated searches, since a depth-first traversal has exactly the advantage of using less memory than a breadth-first search. But not caching the visited states leads to the same path being searched many times.

Is there an easy way of combining the good properties of the two traversal techniques: that solutions are always found without excessive memory usage? In [19] the authors suggest to analyze the NSPK protocol using the following "iterative depth-first search" strategy:

- Use depth-first traversal techniques to search the tree up to some depth d_0;
- If the desired state was not found in the previous search, use depth-first traversal again to search the tree up to some greater depth, say depth $d_0 + 1$ or $d_0 + 2$. Repeat this procedure until the states are found.

Exercise 211 *Explain why the "iterative depth-first" strategy will always find a solution, if there is one, without using excessive memory.*

The disadvantage of the described approach[11] is also obvious. It is bad enough that a depth-first traversal works on trees that are much larger than necessary since the search is not cut off when it encounters states it has seen before. In addition, the iterative depth-first search searches parts of these huge trees over and over again (since a traversal up to depth d is repeated in the search up to level $d + 1$ unless the desired state is found in the meantime).

[11] Remember that the strategy was written before Maude had a built-in efficient search command.

It could still be interesting to try this search strategy since Maude computes fairly efficiently and since our search was killed by excessive memory usage.

The next question is how to define the desired analysis strategy. The iterative depth-first search is a strategy which is not dependent on the particular application and could be used to analyze both the alternating bit protocol, the two-phase commit protocol, and the NSPK protocol.

A user may define her own analysis strategy in an entirely modular way *in Maude*[12] using the *reflective* capabilities in rewriting logic which are reified in Maude's `META-LEVEL` module. We will exemplify meta-programming by specifying the iterative depth-first search strategy in Maude in the course INF 5130. We will then use this strategy to analyze our NSPK protocol specification. My gut feeling is that the number of states encountered will be too large to finish the search in a reasonable amount of time (whatever that may be), but we'll see. The search in [19] probably succeeded because they searched through a smaller state space.

[12] I.e., one does not need to learn a new formalism to define analysis strategies.

Bibliography

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[3] L. Bachmair. *Canonical Equational Proofs.* Birkhäuser, 1991.

[4] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[5] C. de Oliveira Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics.* PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, 2001.

[6] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.

[7] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989. Preliminary version appeared as Digital Equipment Corporation Systems Research Center report no. 39, 1989.

[8] S. Chen, J. Meseguer, R. Sasse, H. J. Wang, and Y.-M. Wang. A systematic approach to uncover security flaws in GUI logic. In *IEEE Symposium on Security and Privacy*, pages 71–85. IEEE Computer Society, 2007.

[9] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.

[10] M. Clavel. *Reflection in general logics and in rewriting logic, with applications to the Maude language.* PhD thesis, University of Navarre, 1998.

[11] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel computation in Maude. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[12] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[13] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.3)*, January 2007. `http://maude.cs.uiuc.edu`.

[14] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[15] M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Reflection'96*, pages 263–288, 1996. `http://jerry.cs.uiuc.edu/reflection/`.

[16] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.

[17] O.-J. Dahl. *Verifiable Programming*. Prentice Hall, 1992.

[18] G. Denker, J. J. García-Luna-Aceves, J. Meseguer, P. C. Ölveczky, Y. Raju, B. Smith, and C. Talcott. Specification and analysis of a reliable broadcasting protocol in Maude. In B. Hajek and R. S. Sreenivas, editors, *37th Annual Allerton Conference on Communication, Control, and Computation*. University of Illinois, 1999.

[19] G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In N. Heintze and J. Wing, editors, *Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana*, 1998.

[20] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[21] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[22] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6. Elsevier, Amsterdam, 1990.

[23] A. J. J. Dick. An introduction to Knuth-Bendix completion. *The Computer Journal*, 34(1):2–15, 1991.

[24] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.

[25] E. W. Dijkstra. Two starvation free solutions to a general exclusion problem. EWD 625, Plataanstraat 5, 5671 Al Nuenen, The Netherlands, 1978.

[26] F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Mobile Maude. In D. Kotz and F. Mattern, editors, *Agent Systems, Mobile Agents, and Applications, Second International Symposium on Agent Systems and Applications and Fourth International Symposium in Mobile Agents, ASA/MA 2000*, volume 1882 of *Lecture Notes in Computer Science*. Springer, 2000.

[27] F. Durán and J. Meseguer. An extensible module algebra for Maude. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[28] F. Durán and J. Meseguer. The Maude specification of Full Maude. Manuscript, May 1999. `http://maude.cs.uiuc.edu/papers/`.

[29] S. Eker. Term rewriting with operator evaluation strategy. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[30] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, J. Meseguer, and K. Sonmez. Pathway logic: Symbolic analysis of biological signaling. In *Pacific Symposium on Biocomputing, Hawaii*, pages 400–412, Jan 2002.

[31] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, and C. Talcott. Pathway logic: Executable models of biological networks. In F. Gadducci and U. Montanari, editors, *Fourth International Workshop on Rewriting Logic and its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[32] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Fourth International Workshop on Rewriting Logic and its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[33] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, fifth edition, 2007.

[34] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 995–1072. Elsevier, 1990.

[35] A. Farzan, F. Chen, J. Meseguer, and G. Rosu. Formal analysis of Java programs in JavaFAN. In R. Alur and D. Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505. Springer, 2004.

[36] M. Fowler and K. Scott. *UML Distilled*. Object Technology Series. Addison-Wesley, second edition, 2000.

[37] M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. Freeman and Company, 1979.

[38] *Gilgamesj*. Cappelen, 1979. Norwegian edition by Per Seyersted.

[39] GMP home-page. `http://www.swox.com/gmp/`.

[40] I. Gnaedig. Termination of order-sorted rewriting. In *Third International Conference on Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 37–52. Springer, 1992.

[41] I. Gnaedig, C. Kirchner, and H. Kirchner. Equational completion in order-sorted algebras. *Theoretical Computer Science*, 72:169–202, 1990.

[42] J. A. Goguen and J. Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[43] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical report, SRI International, Computer Science Laboratory, 1988.

[44] A. Goodloe, C. A. Gunter, and M.-O. Stehr. Formal prototyping in early stages of protocol design. In *WITS'05*. ACM Press, 2005.

[45] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in JAVA*. J. Wiley & Sons, 1998.

[46] M. G. Hinchey and J. P. Bowen (eds). *Applications of Formal Methods*. Prentice-Hall, 1995.

[47] C. A. R. Hoare. An axiomatic approach to computer programming. *Comm ACM*, 12:576–580, 1969.

[48] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[49] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[50] Information systems processing – open systems interconnection – LOTOS. Tech. rep. International Standards Organization DIS 8807, 1987.

[51] Z.100 ITU specification and description language (SDL), June 1994.

[52] S. Kamin and J.-J. Lévy. Two generalizations of the recursive path ordering. Unpublished Note, Department of Computer Science, University of Illinois, Urbana, IL, 1980.

[53] S. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele. Scalable fair reliable multicast using active services. *IEEE Network Magazine (Special Issue on Multicast)*, 14(1):48–57, 2000.

[54] B. Kirkerud. Hvorfor er det så vanskelig å programmere? Talk given to Norwegian radio. In Norwegian.

[55] B. Kirkerud. Lecture notes on rewrite systems. Dept. of Informatics, University of Oslo, 1994. `http://www.ifi.uio.no/in307/notater/`.

[56] J. W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 1–116. Oxford University Press, 1992.

[57] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[58] B. Lampson and H. Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox Palo Alto Research Center, 1976.

[59] E. Lien. Formal modelling and analysis of the NORM multicast protocol using Real-Time Maude. Master's thesis, Department of Linguistics, University of Oslo, 2004.

[60] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of abstract data types*. J. Wiley & Sons and B.G. Teubner Publishers, 1996.

[61] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.

[62] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *TACAS'96*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[63] R. R. Lutz. Analyzing software requirements errors in safety-critical embedded systems. In *IEEE International Symposium on Requirements Engineering*, pages 126–133, San Diego, CA, January 1993.

[64] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[65] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1991.

[66] N. Martí-Oliet and J. Meseguer. Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science*, 285, 2002.

[67] T. McCombs. Maude 2.0 primer. `http://maude.cs.uiuc.edu/primer/`, 2003.

[68] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. `http://www.cacr.math.uwaterloo.ca/hac`.

[69] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[70] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[71] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In U. Montanari and V. Sassone, editors, *Concur'96*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer, 1996.

[72] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[73] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997*, NATO ASI Series F: Computer and Systems Sciences 165, pages 347–398. Springer, 1998.

[74] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[75] R. Needham and M. Schroeder. Using encryption for authenticationion in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[76] P. C. Ölveczky. Terminering av typeordnet omskrivning. Master's thesis, Department of Informatics, University of Oslo, Norway, 1994. In Norwegian.

[77] P. C. Ölveczky, P. Kosiuczenko, and M. Wirsing. An object-oriented algebraic steam-boiler control specification. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Application: Specifying and Programming the Steam-Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*, pages 379–402. Springer, 1996.

[78] P. C. Ölveczky and O. Lysne. Termination of order-sorted rewriting. In M. Hanus and M. Rodriguez-Artalejo, editors, *Fifth International Conference on Algebraic and Logic Programming*, volume 1139 of *Lecture Notes in Computer Science*. Springer, 1996.

[79] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.

[80] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[81] P. C. Ölveczky and J. Meseguer. The Real-Time Maude tool. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, Lecture Notes in Computer Science. Springer, 2008. To appear.

[82] P. C. Ölveczky, J. Meseguer, and C. L. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design*, 29(3):253–293, 2006.

[83] P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. In M. M. Bonsangue and E. B. Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *Lecture Notes in Computer Science*, pages 122–140. Springer, 2007.

[84] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Automated Deduction–CADE-11*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, 1992. `http://pvs.csl.sri.com`.

[85] M.T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, second edition, 1999.

[86] V. Pratt. Anatomy of the Pentium Bug. In P. D. Mosses, M. Nielsen, and H. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107. Springer, 1995.

[87] W. Reisig. *Petri Nets*, volume 4 of *EATCS monographs on Theoretical Computer Science*. Springer, 1985.

[88] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[89] H. Rueß, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 123–134. Springer, 1996.

[90] J. Rushby. Automated deduction and formal methods. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 1996.

[91] J. Rushby. Mechanized formal methods: Progress and prospects. In *16th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lecture Notes in Computer Science*, pages 43–51, Hyderabad, India, 1996. Springer.

[92] M. K. Srivas and S. P. Miller. Formal verification of the AAMP5 microprocessor. In [46], 1995.

[93] M.-O. Stehr and C. Talcott. Plan in Maude: Specifying an active network programming language. In F. Gadducci and U. Montanari, editors, *Fourth International Workshop on Rewriting Logic and its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[94] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[95] U. Waldman. Semantics of order-sorted specifications. *Theoretical Computer Science*, 94:1–35, 1992.

[96] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers, 2002.

[97] M. A. Weiss. *Data Structures and Problem Solving Using Java*. Addison-Wesley, 1998.

[98] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 13. Elsevier, Amsterdam, 1990.

Appendix A

Exam and "Review" Exercises

This appendix contains some exercises that have been used in my class at the University of Oslo as either exam exercises, specification and analysis "projects," or "review" exercises.

A.1 Exercises to Part I of the Course

Exercise 212 – Termination Orderings (Exam 2002, Exercise 1)

In this exercise we study an attempt at defining a termination ordering that (perhaps) can be used to prove termination of equational specifications.

The so-called *exam ordering* $\succ_{ex}$ extends a *total* ordering $\succ$ (also called a *precedence*) on the set of function symbols[1]. We define the exam ordering on ground terms such that

$$t \succ_{ex} u$$

holds if and only if the list (the number of occurrences in t of the largest function symbol in $\succ$, the number of occurrences in t of the second largest function symbol in $\succ$, ..., the number of occurrences in t of the least function symbol in $\succ$)

$$>^{lex}$$

the list (the number of occurrences in u of the largest function symbol in $\succ$, the number of occurrences in u of the second largest function symbol in $\succ$, ..., the number of occurrences in u of the least function symbol in $\succ$).

That is, we first check the number of occurrences of the $\succ$-largest function symbol in t and u. If t and u have the same number of occurrences of this function symbol, then we compare with respect to the second largest function symbol in $\succ$, and so on. For example, if $f \succ g \succ a \succ b$, then both $f(a, b, f(a, b, a)) \succ_{ex} f(g(g(a)), g(g(b)), b)$ and $f(g(a), g(b), f(a, b, a)) \succ_{ex} g(f(a, b, f(a, b, a)))$ hold, while $f(g(a), b, a) \succ_{ex} f(g(a), b, g(b))$ does not hold.

[1] A total ordering in this case is a strict partial ordering $\succ$ such that for each pair f, g of function symbols with $f \neq g$, either $f \succ g$ or $g \succ f$ holds.

1. Which of the following statements hold when the precedence $\succ$ is defined by $f \succ g \succ a \succ b$?

   (a) $a \succ_{ex} b$

   (b) $g(g(a)) \succ_{ex} g(g(b))$

   (c) $f(g(f(a,b,b)),b,b) \succ_{ex} f(g(g(b)),g(g(a)),b)$

2. Does there exist a precedence $\succ$ such that

   (a) $ack(s(a),0) \succ_{ex} ack(a,s(0))$ and/or

   (b) $f(a,s(b),c) \succ_{ex} f(a+b+c,c,a+a)$

   hold? Give the precedence in each case where such a precedence exists.

3. Is $\succ_{ex}$ well-founded (when the set of function symbols is finite)? Justify your answer.

4. Is $\succ_{ex}$ a simplification ordering? Justify your answer.

5. How can the exam ordering be used "in practice" to prove termination of systems? (*Hint:* Consider the treatment of variables, among other things.)

6. Can you use the exam ordering to prove that the specification $\{f(x) = g(x), g(b) = f(a)\}$ is terminating?

7. Can you give an example of a specification that can be proved to be termination using the exam ordering but that cannot be proved to be terminating using the lexicographic path ordering (lpo)?

Exercise 213 – Binary Search Trees (Exam 2002, Exercise 2)

In this exercise we use *binary search trees* to store pairs

$$key :: data$$

where *key* is an integer (a "key") and *data* is an element of a sort `Value` which is not further specified.

A binary *search* tree is a binary tree (of ($key :: data$)-pairs) that satisfies the following requirements:

- each key in the left subtree is smaller than the key in the root;
- each key in the right subtree is greater than or equal to the key in the root; and
- the above two requirements hold for all subtrees of the binary tree.

1. Figure A.1 shows four binary trees. Which of these are binary *search* trees according to the definition above?

2. We represent pairs by the following data type:

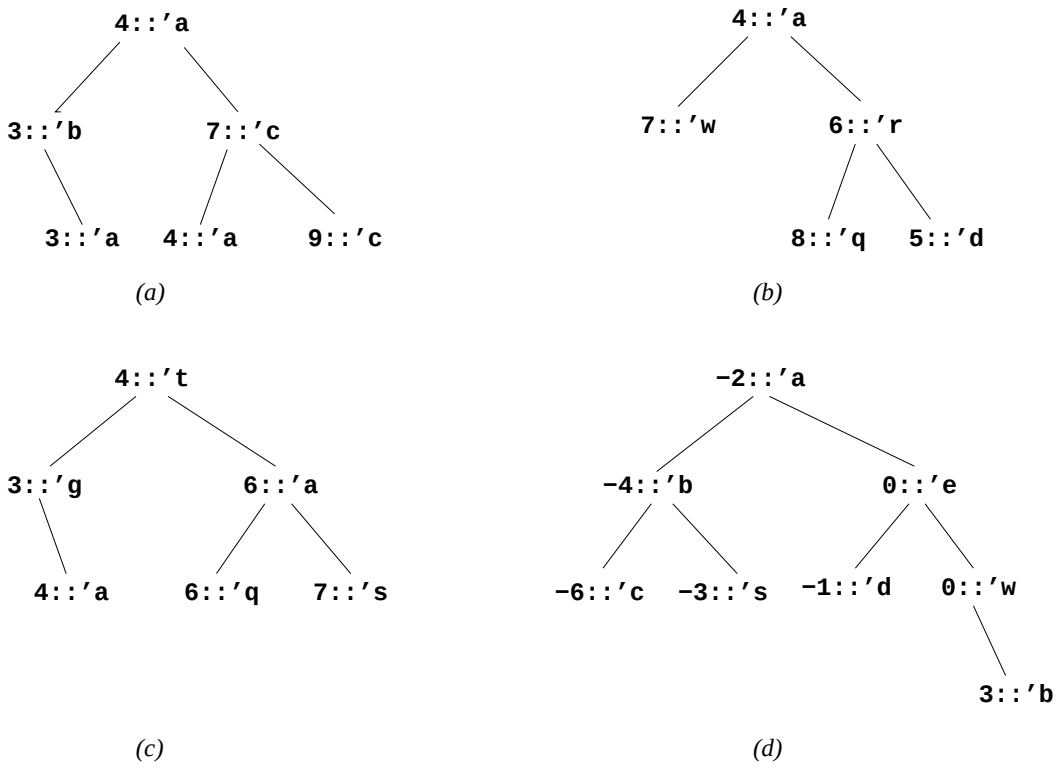

Figure A.1: Some binary trees.

```
sorts Value Pair .
op _::_ : Int Value -> Pair [ctor] .
```

We represent binary trees (that is, not only binary *search* trees) using the following data structure:

```
sort BinTree .
subsort Pair < BinTree .
op niltree : -> BinTree [ctor] .
op _^_^_ : BinTree Pair BinTree -> BinTree [ctor] .
```

A term LS `^` P `^` RS represents a binary tree with left subtree LS, root P, and right subtree RS.

(a) All trees (except the empty tree) can be represented in different ways in the specification above. Give a concrete example of multiple representations of the same binary tree.

(b) How would you represent the binary trees (a) and (b) in Fig. A.1 as terms of sort `BinTree`?

(c) Add one or more equations to the specification so that each binary tree has a unique normal form.

3. Use membership axioms to define a subsort `BSTree < BinTree` for binary *search* trees. (*Note: this exercise was not part of the exam.*)

4. Define a function

   ```
   op insertPair : BSTree Pair -> BSTree .
   ```

   which inserts a `Pair` into a binary search tree so that the resulting tree remains a binary search tree. **Requirement:** In this *and the following exercises* you should for efficiency reasons exploit the fact that the trees are *search* trees, so that functions should not traverse larger parts of the tree than necessary.

5. Define a sort `SetOfValues` of sets of `Value` elements such that `Value` is a subsort of `SetOfValues`, and define a function

   ```
   op dataValues : BSTree Int -> SetOfValues .
   ```

   that finds all data values with the given key in a tree.

6. Define an operation

   ```
   op newDataValue : BSTree Int Value -> BSTree .
   ```

   such that `newDataValue`(*tree*, *key*, *newValue*) returns the tree *tree* where each pair with key *key* has *newValue* as its data component.

Exercise 214 – Texts and Patterns (Exam 2003, Exercise 1)

We define a datatype `Text` as follows:

```
sorts Letter Text .
subsort Letter < Text .
ops a b c d e f g h i j k l m n o p q r s t u v w x y z : -> Letter [ctor] .
op __ : Text Text -> Text [ctor assoc] .
```

1. A *palindrome* is a text that is the same when read from left to right as when read from right to left. For example, "`a b b a`" and "`b o b`" and "`m a d a m i m a d a m`"[2] are all palindromes, while "`s a l l y`" is *not* a palindrome. Define a function

   ```
   op isPalindrome : Text -> Bool .
   ```

   that checks whether a given ground term of sort `Text` is a palindrome.

2. Define a function

   ```
   op _isPrefixOf_ : Text Text -> Bool .
   ```

[2] The first (?) sentence uttered, "Madam, I'm Adam," is therefore a palindrome.

such that t `isPrefixOf` t' is `true` if and only if t is a prefix of t'. For example, both "`b`", "`b o`", and "`b o b`" are prefixes of "`b o b`".

3. Define a function

```
op _isSubstringOf_ : Text Text -> Bool .
```

such that t `isSubstringOf` t' is `true` if and only if the `Text` t exists somewhere in t' (that is, t is a substring of t'). For example, both "`a b c isSubstringOf a b d b e a b c x`" and "`a b c isSubstringOf a b c`" are `true`, while "`a b c isSubstringOf a b`" and "`a b c isSubstringOf b a b d c e`" are `false`.

We now extend the data type `Text` with a sort `Pattern` whose elements are `Text`s that may also contain the character '`?`'. The character '`?`' is supposed to correspond to the UNIX "wild card," where the character "matches" *any* letter:

```
sort Pattern .
subsort Text < Pattern .
op ? : -> Pattern [ctor] .
op __ : Pattern Pattern -> Pattern [ctor assoc] .
```

4. Extend the function `_isPrefixOf_` to a function

```
op _isPrefixOf_ : Pattern Text -> Bool .
```

such that p `isPrefixOf` t is `true` if and only if p "matches" the beginning of t when each occurrence of '`?`' can be replaced by any *one* letter. For example, both "`? ? isPrefixOf a b c`" and "`? b ? isPrefixOf a b c`" are `true`, while both "`? a ? isPrefixOf a b c`" and "`a ? c ? isPrefixOf a b c`" are `false`.

5. Extend the function `_isSubstringOf_` to a function on patterns:

```
op _isSubstringOf_ : Pattern Text -> Bool .
```

Exercise 215 – Termination (Exam 2003, Exercise 2)

We consider unsorted equational specifications. By the *union* $E_1 \cup E_2$ of two equational specifications E_1 and E_2 we mean the union of their function symbols, variables, and equations.

1. Show that there exist *terminating* specifications E_1 and E_2 with *two* function symbols in common such that their union $E_1 \cup E_2$ is *nonterminating*.
2. Is it always the case that $E_1 \cup E_2$ is terminating when both E_1 and E_2 are terminating, and when they have exactly *one* function symbol in common? Justify your answer.
3. Assume that both E_1 and E_2 can be proved to be terminating by the lexicographic path ordering, and assume that E_1 and E_2 have no function symbol in common. Is their union $E_1 \cup E_2$ always terminating? Justify your answer.

4. Same question as above (3), but now E_1 and E_2 have exactly *one* function symbol in common.

5. Same question as above, but now E_1 and E_2 have exactly *two* function symbols in common.

6. (Slightly more difficult?) I have earlier claimed that the system $\{f(a, b, x) = f(x, x, x)\}$ is terminating. *Prove* that there is no *monotonic* weight function *weight* (that maps each ground term to a natural number, and where weight comparison is the usual "greater-than" relation on natural numbers) that can be used to prove termination of this specification. *Hint:* Check the possible relationships between $weight(a)$ and $weight(b)$. The case $weight(a) = weight(b)$ is somewhat tricky.

Exercise 216 – Termination (Exam Preparation, 2002, Exercise 3)

In this exercise we study a new attempt at trying to prove termination of equational specifications.

The so-called *exam preparation ordering* $\succ_{epo}$ is defined on *ground terms* by

$$t \succ_{epo} u$$

if and only if t contains all the function symbols that occur in u, and, in addition, contains at least one function symbol that does *not* occur in u.

1. Which of the following statements are true?
   (a) $f(g(a, b)) \succ_{epo} g(g(a, f(a)), f(f(a)))$
   (b) $a \succ_{epo} b$
   (c) $g(f(a)) \succ_{epo} f(h(a))$
   (d) $f(k(h(a))) \succ_{epo} f(h(a))$

2. Is $\succ_{epo}$ a simplification ordering? Justify your answer.

3. (a) Is $\succ_{epo}$ well-founded over ground terms in a *finite* signature?
   (b) Is $\succ_{epo}$ well-founded over ground terms in an *infinite* signature (that is, the signature may contain an infinite number of function symbols, including constants)?

   Justify your answers.

4. If $t \rightsquigarrow u$ implies $t \succ_{epo} u$, for all ground terms t and u in a specification E, is it then the case that E must be terminating? Justify your answer. (Notice that we do not require the signature to be finite or the specification to have only a finite number of equations.)

5. Can $\succ_{epo}$ in practice be used to prove termination of equational specification?

   - If your answer is "yes," explain how $\succ_{epo}$ can be used to prove termination of a specification.
   - If your answer is "no," explain why $\succ_{epo}$ is not practically useful for proving termination.

Exercise 217 – Even More Termination (Exam Preparation 2, 2002, Exercise 2)

1. Let Σ be a finite signature and let $\succ$ be a strict partial ordering on the set $\mathcal{T}_\Sigma$ of ground terms. Is it the case that $\succ$ is well-founded (on $\mathcal{T}_\Sigma$) if $\succ$ is a simplification ordering?

2. Let $\succ$ be a well-founded strict partial ordering on $\mathcal{T}_\Sigma$ for a finite signature Σ. Is it then the case that $\succ$ must be a simplification ordering? Explain!

We now define a new termination ordering as follows. Let $\succ$ be a strict partial ordering (a *precedence*) on the set F of function symbols in a given signature Σ. Let $w : F \to \mathbb{N}$ be a weight function that assigns to each *function symbol* f a non-negative weight $w(f)$. Note that $w(f)$ may be 0 for some f. Such a weight function $w : F \to \mathbb{N}$ can be extended to a weight function $w^* : \mathcal{T}_\Sigma \to \mathbb{N}$ on ground terms in the usual way, where the weight $w^*(t)$ is the sum of the weights of the function symbol occurrences in t. For example, if $w(a) = 3$ and $w(f) = 1$, then $w^*(f(f(a))) = 5$.

We define *our favorite ordering* $\succ_{ofo}$ inductively as follows:

- $t \succ_{ofo} u$ holds if $w^*(t) > w^*(u)$ (i.e., t weighs more than u);
- $f(t_1, \ldots, t_m) \succ_{ofo} g(u_1, \ldots, u_n)$ holds if $w^*(f(t_1, \ldots, t_m)) = w^*(g(u_1, \ldots, u_n))$ and $f \succ g$ (that is, if two terms have the same weight, then the term whose top symbol is greater in $\succ$ is the greater term);
- $f(t_1, \ldots, t_m) \succ_{ofo} f(u_1, \ldots, u_m)$ holds if $w^*(f(t_1, \ldots, t_m)) = w^*(f(u_1, \ldots, u_m))$ and $(t_1, \ldots, t_m) \succ_{ofo}^{lex} (u_1, \ldots, u_m)$ (that is, if two terms have the same weight and the same top symbol, then their subterms are compared lexicographically); and
- ($\succ_{ofo}$ is the *smallest* relation satisfying the above conditions).

3. Which of the following statements hold, given the precedence $f \succ g \succ a \succ b$ and the weight function w defined by $w(f) = 1$, $w(g) = 2$, $w(a) = 3$, and $w(b) = 1$?
   (a) $f(a, b) \succ_{ofo} f(b, a)$
   (b) $f(g(b), b) \succ_{ofo} f(a, b)$
   (c) $f(g(b), b) \succ_{ofo} g(g(b))$
   (d) $f(b) \succ_{ofo} g(a)$

4. Prove that $\succ_{ofo}$ in general is not a simplification ordering. (*Hint:* Remember that function symbols can have weight zero.)

5. Prove that $\succ_{ofo}$ is not (necessarily) well-founded.

From now on we assume that each constant must have weight strictly greater than 0.

6. Prove that $\succ_{ofo}$ still does not have to be well-founded.

7. What restrictions must be placed on a *unary* function symbol f with weight $w(f) = 0$ so that $\succ_{ofo}$ becomes a simplification ordering? (A *unary* function symbol is a function symbol that takes *one* argument.)

8. Prove that $\succ_{ofo}$, with the restriction you found above, is a simplification ordering.

9. Explain why there does not exist a precedence $\succ$ and weight function w such that $f(f(a)) \succ_{ofo} f(g(f(a)))$ when $\succ$ and w satisfy all the restrictions placed upon them throughout this exercise.

Exercise 218 – Even More Termination (Exam Preparation 2, 2002, Exercise 3)

1. Is it decidable whether or not a given equational specification E can be proved terminating using the lexicographic path ordering?

2. Explain how your answer above relates to the fact that it is in general undecidable to check whether or not a specification is terminating.

Exercise 219 – Termination and Weight Functions (Exam Preparation 2, 2002, Exercise 4)

1. Use weight function techniques to prove that the specification

$$\{s(x) + y = s(x + y),\ 0 + x = x\}$$

   is terminating.

2. Use weight function techniques to prove that

$$\{x * (y + z) = (x * y) + (y * z)\}$$

   is terminating.

Exercise 220 – Termination (Midterm Exam, 2005, Exercise 2)

1. Does there exist a *simplification ordering* that can be used to prove that the specification $\{f(h(x)) = k(x),\ k(k(x)) = f(g(f(x)))\}$ is terminating? Prove/justify your answer.

2. Does there exist a *simplification ordering* that can be used to prove that the specification $\{f(a) = f(g(b, a))\}$ is terminating? Prove/justify your answer.

3. Does there exist a *simplification ordering* that can be used to prove that the specification $\{f(f(x)) = g(x),\ g(g(x)) = f(x)\}$ is terminating? Prove/justify your answer.

Exercise 221 – Simulating *Blackjack* using Equational Specifications in Maude (Midterm Exam 2005, Exercise 1)

The writer of these lecture notes, living in a cold country and burdened by a huge mortgage, likes to visit that magical city in the desert, Las Vegas. However, despite "excellent" strategies, these visits have not produced significant financial benefits. Therefore, he wishes to use Maude to simulate different *blackjack*-strategies in order to become a millionaire.

A short introduction to blackjack. Blackjack is a game between a *player* and the *dealer* (also known as the *casino* or the *bank*). The objective of the game is for the player to be dealt cards so that his hand value is closer to 21 than that of the dealer, *without going over 21*. Other players at the table are of no concern.

The game starts starts with the player placing his desired *bet* and receiving his first card, upon which the dealer gets his first card[3], that is visible to all. The player then gets another card. The player can then evaluate the situation and can ask for more cards, one by one, in order to get a hand value (sum of the values of the cards) as close as possible to 21 without going over 21. The player must use his intelligence/intuition throughout the game to evaluate whether or not he should ask for another card. After the player is finished, the dealer draws his remaining cards. The dealer *must* follow an explicitly stated strategy. In short, the winner is the one whose hand value is closer to 21 without going over 21. By *blackjack* we mean a hand consisting of *two* cards and having value 21.

The player *loses* his bet if at least *one* of the following situations occur:

- the sum of the values of the player's card is greater than 21[4];
- the dealer has *blackjack* and the player has not (even if the player's hand value is also 21!); or
- both the player and the dealer have hand value ≤ 21, but the dealer is closer to 21 than the player.

The player is paid *twice* his bet if *all* the following conditions hold:

- the player's hand value is less than or equal to 21;
- the player does *not* have *blackjack*; and
- the dealer's hand value is greater than 21, or is smaller than the player's hand value.

The player gets his money back if *one* of the following conditions holds:

- both the player and the dealer have *blackjack*; or
- neither the player nor the dealer has *blackjack*, but both have the same hand value ≤ 21.

[3] Actually, both the player and the dealer gets *two* cards in the beginning, but we disregard this (meaningless) fact to have a somewhat simpler modeling task.

[4] The player loses his bet if he goes over 21, even if the dealer also goes over 21! This is the casino's built-in advantage and is the reason why casinos are happy to offer blackjack.

The player gets *two and a half* times his bet back if

- the player has *blackjack* and the dealer has not.

Some examples:

- Peter bets his last \$100, gets two cards with hand value 13, and chooses (after some "thinking") to ask for another card, which shows a '9'. Peter's hand value is 22 and dealer takes Peter's last \$100.
- Lizzie bets \$20 and gets sum 21 on her first two cards. Blackjack! Lizzie keeps her \$20 and, in addition, gets \$30 from the dealer, who did *not* get blackjack.
- Imtiaz bets \$200, gets sum 18 on his first two cards, and, amazingly enough, asks for another card ... which shows a '3'! Imtiaz has hand value 21, but does not have blackjack. The dealer stops on 20. Imtiaz keeps his \$200 and receives another \$200 from the dealer.
- George gets sum 16 on his first two cards and chooses (wisely) *not* to ask for another card. Unfortunately, the dealer gets sum 18, so George loses his bet.
- Denise places a \$80 bet, receives two cards with sum 12, and chooses *not* to ask for more cards. This turns out to be a smart move, as the dealer draws until he gets a hand value 24. Denise keeps her \$80 and gets another \$80 from the dealer.
- Petya bets \$50, gets sum 20 on his first two cards, and is happy with that. Unfortunately, the dealer also gets 20. Push. Petya keeps his \$50, but does not win anything.

In blackjack, the cards are valued as follows: An *ace* can count as either 1 or 11. The cards 2 through 9 have values as indicated. The '10', *jack*, *queen*, and *king* all have value 10. The suits of the cards do not have any meaning in the game. The value of a hand is simply the sum of the point counts of each card in the hand. For example, a hand containing $(5, 7, 9)$ has the value of 21. The *ace* can be counted as either 1 or 11. You need not specify which value the ace has. It is assumed to always have the value that makes the best hand. For example, a hand with a '9' and two aces can have value $9 + 1 + 1 = 11$, or $9 + 11 + 1 = 21$, or $9 + 11 + 11 = 31$. The best here is obviously 21.

The following sort `Card` models the set of possible playing cards:

```
sorts Suit Value Card .
ops 2 3 4 5 6 7 8 9 10 J Q K A : -> Value [ctor] .
ops club spade diamond heart : -> Suit [ctor] .
op <_,_> : Suit Value -> Card [ctor] .
```

An ace of diamonds is therefore modeled as a term `< diamond, A >`.

1. Define a sort `Hand` for *multisets* of playing cards. (Notice that we are interested in *multisets* instead of *sets* since we sometimes play with more than one deck of cards.)

2. Define a constant

```
op deckOfCards : -> Hand .
```

to be one deck of cards (that is, a multiset containing exactly one element of each constructor ground term of sort `Card`).

3. In this exercise you should define the value of a hand. Since an ace may have value 1 or 11, define two functions

```
ops smallestValue largestValue : Card -> Nat .
```

where `smallestValue` gives the smallest value of a card, and `largestValue` denotes its greatest value.

4. Define three functions

```
ops smallestValue largestValue bestValue : Hand -> Nat .
```

that denote, respectively, the smallest, largest, and best value of a hand. The best value is the value that, if possible, is the closest to 21 without going over 21.

5. Define a function

```
op blackjack : Hand -> Bool .
```

that checks whether or not a hand is a *blackjack*, that is, consist of two cards with (best) sum 21.

6. Define a function

```
op result : Hand Hand Nat -> Nat .
--- usage: result(playersHand, dealersHand, bet)
```

such that `result(`*playersHand*, *dealersHand*, *bet*`)` defines how much money the player gets back (that is, his own bet plus a possible win), given that the player made a $\$bet$ bet, ended up with hand *playersHand*, while the dealer ended up with hand *dealersHand*.

7. In the rest of this exercise, we assume that the following functions have been defined:

```
op delete : Card Hand -> Hand .
op random : Nat -> Nat .    --- random(i) generates the i'th random number
op getRandomCard : Hand Nat ~> Card .
```

The function `delete` removes (at most one occurrence of) a given card from a deck of cards, `random` is a function (built-in into Maude) so that `random(`i`)` generates the ith pseudo-random number. The function `getRandomCard` takes a deck of cards and a seed

(corresponding to the argument to `random`), and returns a "random" card from the deck given as the first argument[5].

Given these functions, you can now simulate a game of blackjack. We will simulate a seemingly coward strategy where the player asks for cards until the smallest value of his hands reaches 15 or more, or until the best value is 18 or more.

The dealer *has to* follow the strategy "dealer stands on all 17's," that is, the dealer *must* draw cards as long as his best value is less than 17, and must stand (stop) when his hand has reached a best value of 17 or more.

In this exercise you should define a function

```
op playSoft : Nat Nat -> Nat .
--- usage: playSoft(bet, randomNo)
```

that simulates one game of blackjack. The first parameter is the player's bet, and the second argument is the random number used by `getRandomCard` to draw "random" cards. The result of `playSoft` should be the amount of money the player gets back (own bet plus win) after a game. Each new game should start with *one* new deck of cards. You do not need to worry about a deck being exhausted before a game is over.

For example, in my simulations, `playSoft(100,1)` returned `200` (win!), `playSoft(100,11)` returned `250` (blackjack!), while `playSoft(100,21)` returned `0` (loss).

8. Finally, we will simulate a whole night at the blackjack table. That is, if I bring $\$n$ to the casino, bet exactly $\$b$ in each round, and play for r rounds (or until I have no money left), then how much money will I have when I leave the casino? I use the above mentioned "soft" strategy in each game. Define a function

```
op manySoftGames : Nat Nat Nat Nat -> Nat .
--- usage: manySoftGames(maxNoOfRounds, moneyLeft,
---                      betPerRound, randomNo)
```

so that `manySoftGames(`r, n, b, s`)` simulates such a night in the casino, with s the initial index of the random number generator. The result should be the amount of money you have after the night in the casino (disregarding money spent on drinks, etc.).

You should make sure that you start each round with a freshly shuffled deck of cards (that is, remember to update the seed for each new round by increasing s by, e.g., 5).

For example, in my Maude executions, `manySoftGames(30,1000,100,1)` returned `750` (i.e., a net loss of $250 after 30 rounds), `manySoftGames(10,1000,100,2)` returned `1150` (net *gain* of $150 after 10 rounds!), and, finally, `manySoftGames(40,1000,100,2)` returned `550` (that is, keep your money after 10 rounds or so!).

You can now experiment with other strategies. For instance, instead of stopping at 15, it may make more sense to stop at 14. With this strategy, the result of the above three gambling

[5]Notice that `getRandomCard` is a *partial* function, since it is mathematically impossible to remove a card from an empty deck of cards. However, in this exercise you do not need to worry about empty decks of cards, so you may very well assume that `getRandomCard` is a total function.

nights are, respectively, \$300, \$1350 (!), and \$850 (that is, slightly better than with the previous strategy). You may also experiment with the dealer strategy "dealer hits soft 17" (which is supposed to give you slightly worse odds[6]), so that you can choose the table/casino which gives you the best chance of walking away a millionaire.

Disclaimer. The strategies mentioned above, as well as the description of blackjack, are fairly simple. Any serious strategy should take the dealer's cards into account, and should use the possibilities of *doubling down* and *splitting*.

Exercise 222 – Confluence (Midterm Exam 2006, Exercise 1)

Given the specification $\{f(a) = b,\ f(f(x)) = x\}$.

1. Prove that the system is not confluent.
2. Add *one* equation to the system so that it becomes terminating *and* confluent, and so that no more terms are equivalent in the new specification than in the old one (that is, $\vdash t = u$ should hold in the new specification if and only if it holds in the old one). Prove that the new specification is confluent.

Exercise 223 – Yet More Termination (Midterm Exam 2006, Exercise 2)

Consider the specification $\{f(g(x)) = h(x),\ h(x) = g(f(x)),\ a = b\}$.

1. Can the system be proved to terminate using "weight functions"? If "yes": use weight function techniques to prove termination of the system; if "no": prove why there cannot exist any weight function that can prove that the specification above is terminating.
2. Can the system be proved to be terminating using the *lexicographic path ordering* or the *multiset path ordering*? Explain.
3. Does there exist a *simplification ordering* that can be used to prove that the system is terminating. Justify your answer well.

Exercise 224 – Inductive Theorems (Midterm Exam 2006, Exercise 3b)

Assume that it has somehow been proved that the `quicksort` function in the lecture notes is correct; that is, that is correctly sorts a list. However, we are less sure about the `mergeSort` function. State precisely and formally the property that states that `mergeSort` correctly sorts a list. *Note: you should just* state *formally the property that we would like to prove; you* do not *need to* prove *it!*

[6] Indeed, my Maude simulations with this dealer strategy left me with, respectively, \$300, \$1350, and \$650; that is, \$200 worse in three nights than with a "dealer stands on all 17s" dealer.

A.2 Exercises to Part II of the Course

Exercise 225 – The Coffee Bean Game (Exam INF 220A, 2002, Exercise 1)

In this exercise we consider some variants of the *coffee bean game* from Section 5.6.3 on page 156, where we are given a sequence of white and black coffee beans, and where a subsequence of coffee beans can be replaced by another subsequence according to given rules.

In the first updated version of the game we have only *one* rule: If a white bean has a black bean to its left and another black bean to its right, then these two black neighbors can be removed. In a more readable form:

$$\bullet \circ \bullet \longrightarrow \circ$$

1. Is the game confluent? Justify your answer.

We now add the following two rules to make the game more exciting:

- a black bean to the right of a white bean can be moved to the left of the white bean (that is, $\circ \bullet \longrightarrow \bullet \circ$)
- two black beans next to each other can be removed (that is, $\bullet \bullet \longrightarrow$ "nothing")

2. Specify this version of the coffee bean game in Maude.
3. Use mathematical techniques (such as "weight functions") to prove that the new version of the game is terminating.
4. Assume now that you start playing the game with the following initial state: black–black–white–white–black (that is, $\bullet \bullet \circ \circ \bullet$).
   (a) What are the possible final states?
   (b) Show how you can use Maude's `search` command to find all possible outcomes of the game when started in the above state.
5. Can you say something about the outcome of the game as a function of the initial state?

Exercise 226 – Equational Logic (Final Exam 2005, Exercise 1)

We consider equational specifications E with the following finiteness property:

> Each term t has only a finite number of terms that are E-equivalent to t. That is, the set $\{t' \mid E \vdash t = t'\}$ is *finite* for each term t in the signature of E.

1. Does the specification `NAT-ADD` in Chapter 2 satisfy the finiteness property described above?
2. Does the specification

```
fmod AC is
  sort S .
  ops a b c d e : -> S .
  op f : S S -> S .
  vars X Y Z : S .
  eq f(X,Y) = f(Y,X) .
  eq f(f(X,Y),Z) = f(X,f(Y,Z)) .
endfm
```

satisfy the finiteness property?

3. Explain why, for each equation in E, the set of variables occurring in the left-hand side of the equation must be the same as the set of variables occurring in the right-hand side of the same equation in all specifications that satisfy our finiteness criterion.

4. Let E be a specification that satisfies the above finiteness property (but is not necessarily terminating or confluent). Explain how one can use Maude to easily check whether $E \vdash t = t'$ holds for two ground terms t and t' in the signature of E.

Exercise 227 – More Blackjack (Final Exam 2005, Exercise 2)

Recall the casino game *blackjack* from Section A.1. In that exercise we simulated *one* particular tactic, and *one* possible dealing of cards (using a `random` function). In this exercise we will instead model *all reasonable plays*. The road to riches is almost too simple to be true: You bring a very small but extremely powerful computer that runs Maude to the casino. When you have to decide whether to *hit* or to *stand*, you just use Maude to (very quickly) check all possible final outcomes in case you hit, and all possible final outcomes in case you stand. Then you can trivially choose whether to hit or to stand.

1. In this exercise, we model all reasonable blackjack games in Full Maude. For simplicity, we first assume that you are sitting at an exclusive $100 table, so that you are the *only* player at the table. Nevertheless, we should develop an object-oriented model that can easily be modified to accommodate more players. The cards are dealt in the following way:[7]

   (a) The player gets his first card.

   (b) The dealer gets his first card.

   (c) The player gets his second card.

   (d) The player continues to draw cards as long as he desires.

   (e) Finally, the dealer draws his remaining cards according to fixed rules (*"dealer must stand on all 17s"*).

   The set of *reasonable* plays by the player are defined as follows:

   - A player *can* stand if the value of his hand is 12 or greater.

[7] This order is usually called the European version. It is a minor simplification of the American version, in which the dealer draws his *second* card after each player has received his second card.

- A player *can* hit if the value of his "best" hand is less than or equal to 18.

Since we wish to model all possible plays, the next card to be dealt can be *any* of the remaining cards. (We will therefore not use a `random` function.) The players and the dealer are modeled as objects of the following classes:

```
class Player | hand : State .
class Dealer | hand : State, remainingCards : Hand .
```

The sort `State` contains a hand and whether a player/dealer may want more cards:

```
sort State .
ops active finished : Hand -> State [ctor] .
```

As you can see, in my specification, the bank has the remaining cards in the deck in the attribute `remainingCards`. Since we model all possible scenarios in one round, it is unnecessary to simulate more than one round of the game.

Specify in Full Maude all reasonable blackjack games with one player and the dealer. Specify a suitable initial state, where we start a game with *one* deck of cards.

2. Assume that you are the middle of a round of blackjack with huge bets. You see your own cards and the dealer's lone card. Should you hit or stand? This is the moment to use your super-powerful Maude machine.[8] Assume that that you have hand H, and that the dealer's card is C, and that the game started with *one* deck of cards. From such a state, give the Full Maude commands to search for, respectively:

   (a) Search for all possible final states where the player *stands* and wins the game.

   (b) Search for all possible final states where the player *stands* and loses the game.

   (c) Search for all possible final states where the player *hits* and wins the game.

   (d) Search for all possible final states where the player *hits* and loses the game.

   (Remember that Full Maude will also output the *number* of solutions, allowing you to easily decide whether to hit or to stand.)

3. There is possibility of *doubling down* in blackjack. That is, the player can *double* his bet when he has received exactly *two* cards. The player then gets exactly *one* more card. This allows a player to exploit his good hand and the dealer's bad first card. For instance, this is a golden opportunity if you have sum 11 on your first two cards, while the dealer's visible card is rotten (say, 2 to 5).[9]

   Since we model the possibility of doubling the bet, we need to keep track of the bet and the player's wallet. A player that can double down is therefore modeled as an object of the following class `SemiAdvancedPlayer`[10]:

[8] Disclaimer: You are *not* allowed to use electronic equipment to help your game in casinos in Las Vegas.

[9] The writer of these lecture notes is less enthusiastic; he had once accumulated enough money and courage to play at the $100 table for the game of the year, when the dealer suggested a *double down*. Yours truly used the money allocated for food and gas for the rest of his vacation ... and lost.

[10] An *advanced* player can also perform a *split*.

```
class SemiAdvancedPlayer | bet : NzNat, moneyLeft : Nat .
subclass SemiAdvancedPlayer < Player .
```

In this exercise you should extend your model so that a `SemiAdvancedPlayer` has the possibility of performing a double down when his first two cards have sum 8, 9, 10, or 11 (without being blackjack).

4. There may be more than one player at a table. Even though each player plays only against the dealer, you might be interested in seeing the other players' cards to get an idea of the remaining cards in the shoe. In particular, such information must be added to the searches above. When there are more players at the table, the dealer deals one card to each player, from right to left (as seen from the player's side), then one to himself, then one card to each player, and then the dealer finishes with the first player, etc.

   (a) Explain/sketch how you can modify your solution to model all reasonable games with up to six players at the table.

   (b) Assume that we have four players and the dealer at the table. What is the largest number of rewrites that can be performed in *one concurrent* rewrite step?

Exercise 228 – More NSPK (Exam 2003, Exercise 4)

Chapter 11 of these notes presented a Full Maude model of the *Needham-Schroeder public-key authentication protocol* (NSPK), showed that NSPK did not necessarily provide authentication, and mentioned that Gavin Lowe has suggested an improvement where the "Message 2" part is replaced by

$$\text{Message } 2'. \quad B \rightarrow A : \quad B.A.\{N_a.N_b.B\}_{PK(A)}$$

so that the modified protocol is

$$\begin{array}{lll} \text{Message } 1. & A \rightarrow B : & A.B.\{N_a.A\}_{PK(B)} \\ \text{Message } 2'. & B \rightarrow A : & B.A.\{N_a.N_b.B\}_{PK(A)} \\ \text{Message } 3. & A \rightarrow B : & A.B.\{N_b\}_{PK(B)} \end{array}$$

1. Specify the modified protocol in Full Maude. That is, show exactly what has to change in our specification of NSPK. (You do not need to do the corresponding changes in the intruder.)

2. Given the following initial state:

```
op exInit : -> Configuration .
eq exInit =
     < "Imtiaz" : InitiatorAndResponder | initSessions : emptySession,
                                          respSessions : emptySession,
                                          nonceCtr : 1 >
     < "Maiken" : InitiatorAndResponder | initSessions : notInitiated("Perle"),
                                          respSessions : emptySession,
                                          nonceCtr : 1 >
     < "Perle" : Intruder | initSessions : emptySession,
```

```
                          respSessions : emptySession,
                          nonceCtr : 1,
                          agentsSeen : "Perle",
                          noncesSeen : emptyNonceSet,
                          encrMsgsSeen : emptyEncrMsg >
    < "Wolfie" : Intruder | initSessions : notInitiated("Imtiaz"),
                           respSessions : emptySession,
                           nonceCtr : 1,
                           agentsSeen : "Wolfie",
                           noncesSeen : emptyNonceSet,
                           encrMsgsSeen : emptyEncrMsg > .
```

Assume that the modified NSPK protocol is indeed secure, and assume that the intruder model is also changed. Which of the following state formulas are then invariant w.r.t. initial state `exInit`?

(a) Neither `"Imtiaz"`'s `initSessions` attribute nor `"Imtiaz"`'s `respSessions` attribute contains a value `trustedConnection("Wolfie")`.

(b) Neither `"Maiken"`'s `initSessions` attribute nor `"Maiken"`'s `respSessions` attribute contains a value `trustedConnection("Imtiaz")`.

(c) `"Maiken"`'s `initSessions` attribute contains a value `trustedConnection("Perle")`.

(d) `"Maiken"`'s `initSessions` attribute does *not* contain a value `trustedConnection("Perle")`.

(e) `"Perle"`'s `respSessions` attribute does *not* contain a value `trustedConnection("Wolfie")`.

(f) `"Wolfie"`'s `initSessions` attribute does *not* contain a value `trustedConnection("Perle")`.

3. We now want to use Full Maude to validate the *first* of the above state formulas that is indeed an invariant w.r.t. initial state `exInit`.

   (a) Which Full Maude command would you use to validate that invariant w.r.t. initial state `exInit`?

   (b) What "result" do you expect from this search command?

Exercise 229 – Sending an Important Message in an Unreliable Network (Exam 2003, Exercise 3)

In this exercise we have a sender that wants to send a *very important* message to a receiver. Unfortunately, the underlying communication medium is not very reliable, so that messages may be lost as described below.

1. Assume that messages can get lost, but that there is some "fairness" in the system, so that an infinite number of messages cannot be lost *without* some messages arriving. That is, if we send sufficiently many messages (without knowing exactly what "sufficiently many" means), then a message will sooner or later arrive. Describe a *terminating* protocol that ensures that the receiver has read at least one copy of the very important message, *or* explain why such a terminating protocol *cannot* exist.

2. Repeat the above exercise, but now assume that no more than 20 messages can get lost in a row.

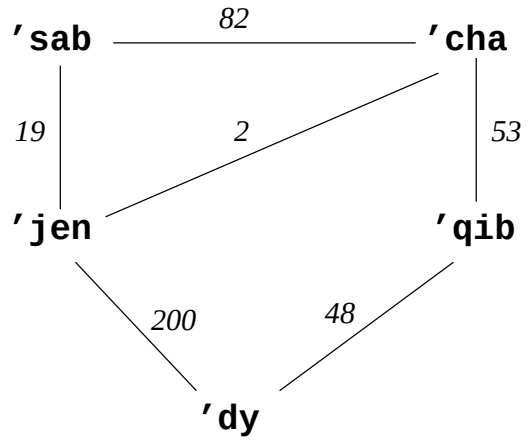

Figure A.2: A graph for the cheapest paths problem.

Exercise 230 – Invariants (Exam Preparation 1, 2002, Exercise 1)

Why is it in general undecidable whether a state formula is (an) invariant w.r.t. an initial state t_0?

Exercise 231 – Cheapest Paths in Undirected Graphs (Exam Preparation 1, 2002, Exercise 5)

We consider in this exercise the well known *cheapest path* problem between *each pair* of nodes in an undirected and connected weighted graph. We are given a a set of "nodes," where each node knows its "neighbors" and the "cost" (or "length" or "weight") of each of its outgoing edges. Our goal is to model a distributed algorithm that finds the cost of the cheapest path between each pair of nodes. Figure A.2 shows a graph where the cost of the edge between Sab(rina) and Cha(rlotte) is 82, while the cheapest path between Dy(ana) and Sab(rina) is 122. Notice that the cheapest path between two nodes is not necessarily the direct link. The cheapest way between from Cha(rlotte) to Sab(rina) includes a visit to Jen(nifer), while the cheapest path from Jen(nifer) to Dy(ana) is not the direct path's 200, but 103. (This phenomenon is fairly common for plane tickets.) Each edge has a cost of at least one dollar.

We model such a system as an object-oriented system, where each node is represented by an object, which initially only knows the cost of the edges to its neighbors. The objects cannot synchronize and can only communicate through message passing. At the end of a run of the algorithm, each node should know the cheapest cost of getting to each of the other nodes in the graph. (We could also have recorded the actual cheapest path, but do not worry about that.)

1. Give an overview of an algorithm which allows each node to find the cost of the cheapest way to each other node.

2. Is it important in your solution that messages from a node n to a node n' are read in the order in which they were sent?

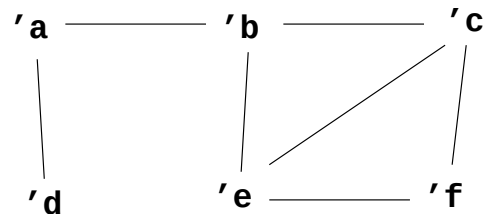

Figure A.3: A communication topology.

3. Specify your algorithm/protocol in Full Maude. Your specification should be simple and elegant (2–3 rules should suffice) and should avoid sending superfluous messages. (This part can be done after considering the following invariant part of the exercise.) How does your algorithm start?

4. Define a *useful* invariant (i.e., a state formula that should be invariant) about the "current" cheapest way values in each object. (Your state formula should be so useful that it can be used in part 7 of this exercise to show that the algorithm is correct.)

5. What is the initial state corresponding to the graph in Fig. A.2?

6. How can you analyze your invariant in Full Maude w.r.t. the above initial state?

7. Use the invariant and some informal reasoning to explain that if your specification terminates (from given initial states), then, in the final state(s?), each node has recorded the cost of the cheapest trip to each other nodes.

8. Explain convincingly why your specification is terminating.

9. What kind of search command would you use to validate your specification (w.r.t. to the initial state above)?

10. What is the *smallest* number of *concurrent* rewrite steps your specification would use ("from start to finish") from the initial state above? Explain briefly what these steps are. (You can disregard the nodes `'dy` and `'qib` to make your life simpler.)

11. Assume now that messages can get lost in the system. Indicate how you would change your specification to handle this situation. Would your state formula from part 4 of this exercise still be an invariant in the new system?

Exercise 232 – Links and Spreaders (Exam Preparation 2, 2002, Exercise 1)

In this exercise, we study communication through link and spreader objects. Assume that the nodes are object of the following class `Node`:

```
class Node | value : Nat, nbrSum : Nat, init : Bool .
```

Messages are supposed to be sent through links. Figure A.3 shows a communication topology with six nodes, `'a`, `'b`, `'c`, `'d`, `'e`, and `'f`.

1. Explain why you need "spreaders" (or something similar) if a node wants to send messages to all its neighbors in one step.

2. Assume that all `nbrSum` values are `0` and that all `init`-values are `false` in the initial states, and assume that the `value` attributes of `'a`, `'b`, `'c`, `'d`, `'e`, and `'f` are, respectively, `7`, `4`, `9`, `2`, `5`, and `1`. What is the initial state corresponding to the graph in Fig. A.3? (Include link and spreader objects in your initial state.)

3. It is boring and time-consuming to type the large initial states when we want to test the specification. In this part of the exercise we therefore define a function `linksAndSpreaders`, declared below, to generate the link and spreader objects in initial states.

```
op linksAndSpreaders : OidSet EdgeSet -> Configuration .
sort EdgeSet .
op _*_ : Oid Oid -> EdgeSet [ctor] .
op noEdge : -> EdgeSet [ctor] .
op __ : EdgeSet EdgeSet -> EdgeSet [ctor assoc comm id: noEdge] .
```

   Define the function `linksAndSpreaders` so that

```
linksAndSpreaders('a ; 'b ; 'c ; 'd ; 'e ; 'f,
                  ('a * 'b) ('b * 'c) ('b * 'e) ('c * 'e)
                  ('f * 'e) ('d * 'a))
```

   defines the links and spreaders for the above topology.

4. Specify in Full Maude an algorithm where the nodes using message passing to find (and record in the `nbrSum` attributes) the sum of the `value`s of all the neighbors. For example, the `nbrSum` attribute of node `'a` should be `6` when the algorithm has finished, since `'a` has neighbors `'b` and `'d` with respective `value`s `4` and `2`.

5. What is the smallest number of *concurrent* one-step rewrites needed to execute your specificaton on the above initial topology?

6. What search command would you use to check that the algorithm computes the correct `nbrSum` attribute for each node in the above initial state? What would you expect the result of the command to be?

7. Can you define a *useful* invariant for this system? (A high-level description is sufficient.)

8. How would use Maude's search command to validate your invariant w.r.t. the given initial state?

9. Can you *prove* the invariant for all possible reasonable initial states? (Again, a clear and informal description is sufficient here.)

Exercise 233 – Earliest Common Meeting Time (Exam Preparation 2, 2002, Exercise 5)

A set of nodes wants to schedule a date for a very important meeting. It is therefore crucial that *all* nodes can attend the meeting. Each node has a list of dates at which it can attend such important meetings. In this exercise we will look at a distributed algorithm for scheduling the meeting at the *earliest* date at which all nodes can attend.

At the end of a run of the algorithm, each node's `meetingTime` attribute should contain the earliest possible common meeting time, or the value `impossible` if no such common meeting time exists. For example, if we have four nodes n_1 to n_4, where n_1 can meet on the 5th, the 12th, the 19th, the 24th, and the 31st; n_2 can meet on the 5th, the 19th, the 24th, and the 31st; n_3 can meet on the 5th, the 24th, and the 31st; and n_4 can meet on the 4th, the 19th, the 24th, the 25th, and the 31st, then the earliest possible meeting time when all can attend is the 24th. If n_1 cannot meet on the 24th and n_3 cannot meet on the 31st, there is no common meeting time.

We use Full Maude to model a solution to the meeting time problem in a distributed system where nodes communicate by message passing. Each node sends out a message to all the other nodes saying the first possible time the node can meet. If all other nodes have also sent out that date as the desired date, then all is fine. If a node receives a later desired meeting time, it must check whether it can attend a meeting at that date; otherwise it must suggest the next possible meeting time it can attend, and so on.

Each node has a sorted list of the dates it can meet, a set of the other nodes in the system, the `meetingTime` attribute, and other attributes you need:

```
class Node | possibleMeetingTimes : NatList, otherNodes : OidSet,
             meetingTime : NatOrImpossible, ... .
```

1. Define a sort `NatOrImpossible` which contains both the natural numbers and an additional value `impossible`.
2. Give a high-level description of a solution to the meeting time problem, and explain why it is *not* necessary to assume that messages between the same pair of nodes are received in the order in which they were sent.
3. Give a Full Maude specification of your solution to the meeting time problem.
4. Define a state formula that is invariant in your specification (w.r.t. sensible initial states) and that relates the meeting time value in the messages with the earliest possible meeting time and the values in the `meetingTime` attribute. This invariant should imply that all the `meetingTime` values are the correct ones when the system has terminated.
5. How would you use Full Maude to prove that the state formula defined/described above is indeed an invariant w.r.t. a given initial state?
6. How would you prove that your specification always finds the correct meeting times, starting with the initial state described in the beginning of this exercise?
7. What is the smallest number of *concurrent* (one-step) rewrite steps that are required in your specification, starting with an initial state described above?

Exercise 234 – Swapping (Final Exam, 2006, Exercise 1)

Given the following Maude module:

```
mod SWAP is protecting NAT .
  sort NatList .     subsort Nat < NatList .
  op nil : -> NatList [ctor] .
  op __ : NatList NatList -> NatList [ctor assoc id: nil] .

  vars M N : Nat .   var L : NatList .

  crl [swap] : M N => N M if N < M .

  op init : -> NatList .
  eq init = 8 2 6 54 2 0 3 .
endm
```

1. What is the largest number of applications of the rule `swap` that can be performed in *one* concurrent step on the term `init`? (You do *not* need to prove your answer in detail, but only briefly explain which rule applications can be performed in the first concurrent rewrite step from `init`.)

2. A crucial invariant in the system is the following state formula: *"the elements in the (current) state are the same (and with the same multiplicity) as in the initial state."*

   (a) Use Maude to that check whether or not the above state formula is an invariant w.r.t. the initial state `init`.

   (b) What result will you get from executing your command in Maude?

3. Assume that we change the rule `swap` to

   ```
   crl [swap] : M N => N M if N <= M .
   ```

   To analyze the *modified* system, we give the command

   ```
   search init =>! L:NatList .
   ```

   Will the execution of this search command terminate? If so: what is the result?

Exercise 235 – Wireless Sensor Networks (Final Exam, 2006, Exercise 2)

A *wireless sensor network* consists of a set of small and cheap battery-powered "computers," called *sensor nodes*. A sensor node is equipped with some sensing capability that allows the node to observe/detect some phenomenon such as smoke, heat, seismic movements, etc. A sensor node is also equipped with a low-powered radio transmitter and receiver, allowing sensor nodes to communicate with each other via radio to form a network. Wireless sensor networks are being introduced in southern California to help detect and locate forest fires. In this exercise we will "help" California by modeling such a fire-detecting sensor network.

We assume for simplicity that the sensor nodes are located on a two-dimensional[11] surface in an area of size `Xsize` × `Ysize`. A sensor node is identified by its location.

Communication is, as mentioned, by radio, where the nodes do *not* have *directed* antennas. A node therefore broadcasts a radio signal in all directions. Due to the weak transmitter, the radio signal can only be received with sufficient strength by the nodes that are within a distance of `transmissionRange` from the sender. Note that a sensor node does not know its "neighbors."

We declare locations and some constants as follows:

```
sort Location .
op _._ : Nat Nat -> Location [ctor] .

--- Some parameters of the system:
ops transmissionRange Xsize Ysize : -> Nat .
eq transmissionRange = 10 .
eq Xsize = 100 .
eq Ysize = 100 .
```

A location is represented by a term x `.` y, with $0 \leq x \leq$ `Xsize` and $0 \leq y \leq$ `Ysize`. We remember from happy childhood days that the distance between two locations x `.` y and x' `.` y' is given by $\sqrt{(x-x')^2 + (y-y')^2}$.

1. Define a function

   ```
   op _withinTransmissionRangeOf_ : Location Location -> Bool .
   ```

   such that l `withinTransmissionRangeOf` l' is `true` if and only if the distance between l and l' is less than or equal to `transmissionRange`. (You do *not* need to use fancy mathematical functions like square root.)

2. Define a sort `LocationSet` of *sets* of `Location` elements. That is, you should not define a *multiset*. Furthermore, define a function

   ```
   op _in_ : Location LocationSet -> Bool .
   ```

   that checks whether or not a `Location` is in a set.

In the large Californian forests there may be fires at different locations, and our excellent system should discover them all. We have two kinds of objects: *sensor nodes* of the class `WSNode`, and a *base station* (or *fire station*) of the class `FireStation`. The fire station receives messages from nearby sensor nodes about the location of fires, and stores the location of *each* fire. To simulate our specification, we also need some fires! We have therefore added to our specification an additional object (a "pyromaniac") of a class `FireGenerator` that "creates" fires at different locations. These three classes are declared as follows:

```
class WSNode | firesSeen : LocationSet .
class FireStation | fireLocations : LocationSet .
class FireGenerator | fireLocations : LocationSet .
```

[11] our model can be easily extended to the three-dimensional setting

These classes should not have further attributes, if you can avoid it. However, you may declare empty sub- or superclasses if you feel like doing that. Notice that a `FireStation` is equipped with a radio receiver that can receive messages in the same way as a `WSNode`. Our `FireGenerator` object should not receive messages.

The fire-reporting algorithm we will model in this exam is a trivial flooding algorithm described as follows:

- The `FireGenerator` object stores in its `fireLocations` attribute all the fires that will take place in the system. For each fire (that is, for each location l in `fireLocations`), the `FireGenerator` object sends a message with content `fireAt(`l`)` that is received by each (`WSNode` or `FireStation`) node that is located within a distance of `transmissionRange` of l. (This models a sensor discovering a fire.) The messages that are communicated in our specification have content of the form `fireAt(`l`)`:

  ```
  sort MsgContent .
  op fireAt : Location -> MsgContent [ctor] .
  ```

- When a `WSNode` receives a signal with content `fireAt(`l`)`, it checks whether it has already seen a fire report from this location, in which case it does nothing. If the node has not seen a fire reported at l, it broadcasts the `fireAt(`l`)` message (to all the other nodes that are within its `transmissionRange`).

- When the `FireStation` object receives a `fireAt(`l`)` message, it records the location l in its attribute `fireLocations` (and hopefully sends a firefighting helicopter to that location).

Sensor nodes and the fire station do not have names. Their object identifiers should instead be their locations. Only the `FireGenerator` should have a name, `FireGen`:

```
op FireGen : -> Oid [ctor] .
subsort Location < Oid .
```

In the following exercise you should model communication for this kind of wireless network: radio transmission where each object (with a radio receiver) in the system within a distance of `transmissionRange` from the sender should receive the message. Remember: a `FireGenerator` object should not receive messages, but can "send" `fireAt(`l`)` messages that should be "received" by each node within distance `transmissionRange` from l; a fire *could* happen anywhere in the area, also at the exact location of a node; a node does not know its neighbors; and the sender of a broadcast should *not* receive the message it is broadcasting.

3. Model this form of broadcast communication, and define explicitly all necessary "message wrappers" and other declarations.

We will now define initial states, so that we can place n sensor nodes in pseudo-random locations in the sensing area. In addition, an initial state should have one `FireGenerator`

object with m fires in random locations. Finally, the initial state should have one `FireStation` object in a pseudo-random location.

Assume given a function `random`, so that, given a "seed" s, `random(`s`)` gives the first pseudo-random number, `random(`$s+1$`)` gives the second pseudo-random number, `random(`$s+2$`)` gives the third pseudo-random number, and so on.

4. Define a function

   ```
   op init : Nat Nat Nat -> ... .
   ```

   such that `init(`n`,`m`,`s`)` generates such a desired initial state with n `WSNode`s placed in pseudo-random locations, m (pseudo-randomly located) fires, and s the initial value of the "seed." For example, in my specification, the command `(red init(5, 3, 1) .)` returned the initial state

   ```
   {< FireGen : FireGenerator | fireLocations : (28 . 19) ; (70 . 82) ; (87 . 38) >
    < 18 . 60 : WSNode | firesSeen : none >
    < 46 . 67 : WSNode | firesSeen : none >
    < 72 . 53 : WSNode | firesSeen : none >
    < 78 . 44 : WSNode | firesSeen : none >
    < 86 . 92 : WSNode | firesSeen : none >
    < 95 . 21 : FireStation | fireLocations : none >}
   ```

5. Model in (Full) Maude the fire-reporting wireless sensor algorithm described above.

6. Give an informal—but precise—"prose" description of a state formula that we would like to be an invariant of the system, and which implies that the `FireStation` has recorded all the fires when there are no messages in the state, and when the `FireGenerator` does not have any more fires "to send." (You do *not* need to test in Maude whether your state formula is invariant.) (Whether or not the formula actually *is* an invariant of the system depends on the topology: if there are too few nodes then there might be fires that are too far away from a sensor node to be discovered, and/or the nodes cannot forward the fire information to the fire station.)

7. Is your (rewriting logic) specification *confluent*? A short and "high level" explanation without much detail is sufficient here.

Appendix B

Some Maude Commands

Below are listed some Maude commands. Maude has lots of other features and commands, including a sophisticated debugger, which are all described in [13].

`red` *term* .

Causes *term* to be reduced using the equations in the "current" module. The current module is the last module entered into Maude unless the `select` command has been used.

`red in` *mod* : *term* .

The term *term* is now reduced in the module *mod*. Example:

```
Maude> red in LIST-INT : length(2 4 66) .
```

`rew` *term* .

Rewrites the specified term *term* as long as possible. That is, the equations, membership axioms, and rewrite rules are used to rewrite and reduce the term until no rule or equation can be applied. The rewrite rules are only applied when no equation (or membership axiom) can be applied. That is, first *term* is reduced to its normal form, then a rule is applied once somewhere, and the resulting term is reduced to its normal form using the equations, and then a(nother) rule is applied, etc.

`rew [`*n*`]` *term* .

As above with the difference that at most *n* *rewrite* steps are performed. (The number of equational reductions is not influenced by this number *n*.) Useful when the rewrite specification is nonterminating.

`rew in` *mod* : *term* .

Performs the rewrite command using the module *mod* instead of the current module.

`rew [`*n*`] in` *mod* : *term* .

Performs at most *n* rewrite steps in the module *mod* starting with term *term* .

`frew` *term* `.`

Fair rewriting. Same as `rew` except that it applies the rewrite rules in a "position-fair" way. Useful in object-oriented specifications since it ensures that each object gets a chance to rewrite. This command can also be given a bound on the number of rewrite steps to perform, and/or a module in which the rewriting is to take place.

`search` *term arrow pattern* `.`

Searches (in a breadth-first way) for terms reachable from the initial state *term* which are matched by the *pattern*, which is a constructor term which may contain variables. The *arrow* is either of

`=>1` (search for terms reachable in *one* rewrite step from *term*),

`=>*` (search for terms reachable in *zero or more* rewrite steps from *term*),

`=>+` (search for terms reachable in *one or more* rewrite steps from *term*), and

`=>!` (search for reachable terms which cannot be further rewritten).

`search` *term arrow pattern* `such that` *condition* `.`

As above but there is now a *condition* on the matches. (Does not work in Full Maude.)

`search [`n`]` *term arrow pattern* `.`

Search for at most n solutions. Also works with `such that` conditions.

`show path` n `.`

Show the rewrite path from the initial term to term number n in the previous search.

`show path labels` n `.`

Show the labels of the rewrite rules applied in the rewrite path from the initial term to term number n in the previous search.

`select` *mod* `.`

Selects *mod* to be the "current" module.

`set protect` *mod* `off .`

Removes module *mod* from the list of modules which are automatically imported in `protecting` mode in every module. We used

```
Maude> set protect BOOL off .
```

before entering our own module `BOOLEAN` so that Maude would not include its `BOOL` module.

`set protect` *mod* `on .`

Adds the module *mod* to the list of modules which are imported automatically (in `protecting` mode) in every module. For example, if we want to automatically import `INT` in every module, then we could give the Maude command

```
Maude> set include INT on .
```

`set include` *mod* `on .`

Same as `set protect` *mod* `on`, except that *mod* is automatically imported *in* `including` *mode* in every module.

`set trace on .`

Turn on tracing. Maude will print how it applied the equations, etc.

`set trace off .`

Remove tracing. (This is the default.)

`trace exclude` *modules* `.`

Don't trace in the given modules. Specially useful when tracing Full Maude executions, in which case the Maude commands

```
Maude> set trace on .
Maude> trace exclude FULL-MAUDE .
Maude> set trace substitution off .
```

will trace the rewrites in your module and will give reasonably nice output.

`show module .`

Maude will output the current module. Good for example to check if Maude read what you wanted it to read.

`show all .`

Maude will output a *flattened* representation of the current module, that is, the current module where the "body" of each imported module is included.

`show sorts .`

Maude will output the sorts and subsorts of the current module. We can also write `ops` (for function symbols), `vars` (for variables), or `eqs` (for equations) instead of `sorts`.

`pwd`

This and the following commands are not ended by a period but by the end of the line. `pwd` is the UNIX command `pwd` which prints the current working directory.

`ls`

The `ls` command works as in UNIX, with flags and arguments.

`cd` *directory*

directory is the new working directory.

`in` *file-name*

Reads the file *file-name*. This line (and the three commands listed above and the two below) can also occur inside Maude modules. This is practical if you e.g., have some code part which you want to include in many files without using the more elegant `protecting` or `including`, for example when you have a bunch of variable declarations which are same in most of your specifications.

`load` *file-name*

Same as `in` *file-name*, but with less output for modules entered. Convenient for reading files with many modules, such as `full-maude.maude`.

`eof`

Causes Maude to respond as if it read the end of the file. Could be practical to insert into a file in which you have a lot of stuff you don't want Maude to read, e.g., while debugging your specifications.

`q`

(Or `quit`.) Exit Maude. Since Maude will also read this comment from a file, you are in for a hasty exit if a module of yours contains a line with just the letter `q`!

B.1 Full Maude Commands

Full Maude commands are always enclosed by parentheses. Maude's `rew`, `frew`, `red`, `search` (without conditions), `select`, and `show module/all/sorts/...` work in Full Maude. The debugger and the `show path` commands don't work in Full Maude.

Maude system commands such as `set trace on .`, `trace exclude`, `ls`, `pwd`, `in`, `load`, `eof`, and `q` also work in Full Maude as Maude system commands and should be written *without* parentheses. Don't forget to give the command `trace exclude FULL-MAUDE .` before tracing Full Maude executions.

Index