Distributed and mobile applications in Maude

Adrián Riesco Rodríguez

Departamento de Sistemas Informáticos y Computación
Facultad de Informática

A Master's Thesis Directed by
Dr. Alberto Verdejo

June 2007

UCM
UNIVERSIDAD
COMPLUTENSE
MADRID

Contents

List of Figures

Chapter 1

Introduction

Most interesting computer systems today, as well as those of the future, are distributed in nature, including the Internet, cellular and PDA communications, biological and bio-tech computations, international trade, multi-national corporate databases, and multi-user games. Concretely, the popularity of the Internet has brought much attention to the world of distributed applications development. Now, more than ever, this network is being viewed as a platform for the development of cost-effective, mission-critical applications.

The main goal of a distributed computing system is to connect users and resources in a transparent, open, and scalable way. Ideally, this arrangement is drastically more fault tolerant and more powerful than stand-alone computer systems. Parallel algorithms divide the problem into more subproblems, pass them to many processors and put the results back together at the end. An *algorithmic skeleton* [25, 58] is an abstraction shared by a range of applications which can be executed in a distributed, parallel way. The aim is to obtain schemes that allow parallel programming where the user does not have to handle low level features like communication and synchronization [3].

On another front, mobile code and mobile agents are emerging technologies that promise to make much easier the design, implementation, and maintenance of distributed systems [41]. Mobile agents may reduce the network traffic, provide an effective means of overcoming network latency, and, perhaps more importantly, help us to construct more robust and fault-tolerant systems, thanks to their ability to operate asynchronously and autonomously [42].

The main issues when programming distributed applications are security and reliability. We must make sure that applications running in a remote host do not affect it, neither the host affects the applications, and that these applications perform the task they have been assigned. Moreover, since systems become larger and more complex, just testing them is not enough to assure the features shown above, and tools to formally analyze them are needed. Intrinsically concurrent formalisms with a precise semantics seem to be unavoidable for such a task, and declarative approaches seem quite promising.

Rewriting logic [48] was proposed in the early nineties as a unified model for concurrency in which several well-known models of concurrent and distributed systems can be represented in a common framework.

The flexibility of rewriting logic for representing very different styles of communication, both synchronous or asynchronous, its facility for supporting distributed, concurrent object-oriented systems, and its reflective capabilities for supporting metaprogramming and dynamic reconfiguration, make it a very suitable formalism for the specification of distributed systems based on mobile agents, on which the proof of properties about security, correctness, and performance, can be based.

Maude [19] is a high level, general purpose language and high performance system supporting both equational and rewriting logic computations. It can be used to specify in a natural way a wide range of software models and systems, and since (most of) the specifications are directly executable, Maude can also be used to prototype those systems. The recently incorporated support in Maude for communication with external objects makes many other application areas (such as mobile computing and distributed agents) ripe for system development in Maude. The Maude language main features are briefly explained in Chapter 2.

As said above, since its version 2.2 Maude supports rewriting with external objects, being TCP sockets the first of such objects, allowing for the first time to connect different Maude processes. Now it is possible to implement *really* distributed systems where different process are connected through sockets. Distributed applications in Maude must be executed on top of what we call an *architecture*, that is, a set of Maude processes running on the same or different machines, and connected through sockets. We describe in Chapter 3 how to implement several different generic architectures, on top of which concrete applications can be implemented. One of our aims has been to implement these architectures in a way as independent of the applications to be run on top of them as possible.

As substantial case studies of distributed applications, we have implemented different algorithmic skeletons and Mobile Maude, a mobile agent language extending Maude and supporting mobile computation.

Regarding the implementation of skeletons in Maude, we do it by means of parameterized object-oriented modules, that receive the basic operations needed to solve a concrete problem as a parameter. These operations usually are part of the sequential version of the concrete applications, thus encouraging code reusability. A skeleton can be executed on top of different architectures/topologies. However, there is often a most suitable architecture for each skeleton that takes advantage of the task distribution specified by it. We take advantage of the separation between the architectures and the concrete applications, allowing us to reuse the same architecture by different skeletons. Our work on skeletons is presented in Chapter 4, and has been published in [59].

Mobile Maude uses reflection to obtain a simple and general declarative mobile language design and makes possible strong assurances about mobile agent behavior. The formal semantics of Mobile Maude is given by a rewrite theory in rewriting logic. Since this specification is executable, it can be used as a prototype of the language, in which mobile agent systems can be simulated. The two key notions are *processes* and *mobile objects*. Processes are located computational environments where mobile objects can reside. Mobile objects have their own code, can move between different processes in different locations, and can communicate asynchronously with each other by means of messages. Mobile Maude's key characteristics include: (1) reflection as a way of endowing mobile objects with "higher-order" capabilities; (2) object-orientation and asynchronous message passing; and (3) a simple semantics without loss in the expressive power of application code.

Mobile Maude was first presented in [28] by F. Durán, S. Eker, P. Lincoln, and J. Meseguer. In that work, the authors presented an executable Maude 1.0.5 specification, in which locations and processes were encoded as Maude terms. The implementation effort was completed by F. Durán and A. Verdejo [30] with the release of Maude 2.0, utilizing the builtin object system, for object/message fairness, just by simplifying and extending the previous specification.

The really distributed implementation of Mobile Maude presented here is a joint work with Francisco Durán and Alberto Verdejo. It is fully described in Chapter 5 and it has been published in [29].

Model checking is a method for formally verifying finite-state concurrent systems [16]. Such systems can be seen as finite state machines, i.e., directed graphs consisting of nodes and edges. A set of atomic propositions is associated with each node. The nodes represent states of a system, the edges represent possible transitions which may alter the state, while the atomic propositions represent the basic properties that hold at a point of execution. Specifications about the system are expressed as modal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not.

Maude's model checker [31] allows us to prove properties on Maude specifications when the set of states reachable from an initial state in such a system is finite. Another Maude's analysis tool is the `search` command, that allows to explore (following a breadth first search strategy) the reachable states in different ways. Chapter 6 shows how to formally verify, by using these tools, properties about the applications shown in the previous chapters.

Finally, conclusions and ongoing work are presented in Chapter 7.

Thus, the use of Maude allows us to have the description of the architecture and the implementation of the application solving a problem in the same high-level language. This has the following advantages:

- Since Maude has a well-defined semantics, we obtain a good basis for formal reasoning and correctness proofs.

- In the skeletons case, it provides much flexibility, as skeleton implementations can easily be adapted to special cases, and if necessary, new skeletons can even be introduced by the programmer himself.

We describe in the following section some works about algorithmic skeletons and mobile agents, giving a succinct comparison with our work. The complete Maude code presented in this work is available in `http://maude.sip.ucm.es/~adrian/master-thesis/`.

1.1 Related work

The first formalization attempt of distributed computation was through process algebras such as CCS [51] and CSP [38]. These works were then extended by introducing several notions regarding mobility; examples of such calculi are the π-calculus [52], the join calculus [34], the ambient calculus [13], or the Seal calculus [14]. Nowadays, several distributed languages exist, being probably the most widely used Java. We show in this work two kinds of distributed applications: algorithmic skeletons and mobile agents.

The term *skeleton*, coined by Murray Cole in the late '80s [20], originates from the observation that many parallel applications share a common set of interaction patterns. Most of skeletons has been implemented in imperative languages [23] such as C [24], FORTRAN [26], and P^3L [2]. From the declarative point of view, a skeleton has typically be seen as a polymorphic higher-order function which can be applied with many different types and parameters [58], like the skeletons implemented in Eden [44] and HaskSkel [37].

Lately, a new generation of object-oriented skeletons such as JaSkel [33], muskel [21], and ASSIST [65] is being developed [22].

Mobile agents computing may be viewed as an extension of remote dispatch of script programs or remote submission of batch jobs. The most significant of these extensions has been the development of the security, because mobile agents carry their own code and run in a remote host, so this execution must be safe for both the agent and the host. Some mobile agents languages are Telescript [63], Obliq [11], Agent Tcl [36], Ajanta [64], and Mobile UNITY [62].

1.1.1 Eden Skeletons

Eden [6, 43] is a higher-order functional language that extends the lazy funcional language Haskell [56] by syntactic constructs for *explicitly* defining process. Eden's process model provides direct control over process granularity, data distribution, and communication topology.

Processes communicate via unidirectional channels which connect one writer to exactly one reader. When trying to access input which is not available yet threads are temporarily suspended. The type class `Trans` (short for transmissible) comprises all types which can be communicated. All primitive types belong to `Trans`. Using the high-level Eden constructs:

```
process :: (Trans a, Trans b) => (a -> b) -> Process a b
-- to transform a function into a process abstraction and
( # ) :: (Trans a, Trans b) => Process a b -> a->b
-- to instantiate a process
```

the programmer can partition the algorithm into parallel sub-tasks, thereby taking into account issues like task granularity, topology, and data distribution. Skeletons in Eden are polymorphic higher-order functions, that will receive as parameters the functions that solve the problem and the tasks to be computed [44].

As in the Maude case, Eden has a well-defined semantics that allows to verify the skeletons [55], and the functional specification and the parallel implementation of these skeletons are in the same language. While the generality of these skeletons relies in higher-order functions, we obtain the same abstraction level:

- by using parameterized modules, as shown with the parameterized skeletons in Chapter 4; and
- by using Maude's reflective features, that allow to use generic mobile objects to create skeletons in Mobile Maude (see Section 5.8).

An important difference between Maude and Eden skeletons is that Eden provides efficient transmission of messages through channels, while we can only use TCP sockets, so we must transform the data in order to transmit messages. This difference changes dramatically the performance of our skeletons.

1.1.2 JaSkel

JaSkel [33] is a skeleton-based framework to develop parallel and distributed applications. The framework provides a set of Java abstract classes as a skeleton catalog, which implements recurring parallel interaction paradigms. The current JaSkel prototype provides the programmer different versions of the farm and the pipeline skeletons (the implementation of these skeletons in Maude is shown in Sections 4.3 and 4.7, respectively).

A JaSkel skeleton is a simple Java class that implements the *Skeleton* interface and extends the *Compute* class. The *Skeleton* interface defines a method *eval* that must be defined by all the skeletons. This method starts the skeleton activity. To create objects that will perform domain-specific computations, the programmer must create a subclass of class *Compute*. The *Compute* abstract class defines an abstract method

public abstract Object compute(Object input)

that defines the domain-specific computations involved in a skeleton.

Thus, to write an application using JaSkel, a programmer must perform the following steps:

1. To structure the parallel program and to express it by using the available skeletons;
2. To refine the supplied abstract classes and write the domain-specific code used as skeleton parameters; and
3. To write the code that starts the skeleton, defining other relevant parameters (such as the number of processors, the load distribution policy, . . .).

Note that this implementation is really similar to ours. The *Compute* class and the *Skeleton* interface correspond with our Maude theories, while the class extending *Compute* and implementing *Skeleton* corresponds with the Maude view from the theory. We have requested the Java code to the authors in order to translate it to Maude and prove properties about these skeletons. Although we have obtained no answer yet, we hope to work in this translation in the future. Moreover, we can also do the inverse transformation, and translate our Maude skeletons into JaSkel ones, in order to increase their speed-up and obtain a more "commercial" implementation.

1.1.3 The ambient calculus

The ambient calculus [13, 12], devised by Luca Cardelli and Andrew D. Gordon in 1998, wants to capture in an abstract way notions of locality, mobility, and ability of crossing barriers. To this end, it focuses on *mobile computational ambients*; that is, places where computation happens and that are themselves mobile.

The main characteristics of ambient calculus are:

- An *ambient* is a bounded place where computation happens. The boundary determines what is inside and what is outside an ambient, and therefore determines what moves.
- Process mobility is represented as crossing of boundaries.
- Each ambient moves as a whole with all its subcomponents.
- Ambients can be nested within other ambients, forming a tree structure.
- Each ambient has a collection of local running processes.
- Each ambient has a name, that is used to control access. That name is unforgeable, being this fact the most basic security property.

Our Mobile Maude specification also has two levels that can be considered an "ambient". Each Maude process, encapsulating a whole configuration, is a place where computation happens and that mobile objects can cross, and each mobile object contains the (metarepresentation of) another configuration where computation can also occur, being the difference that the objects in the latter configuration cannot travel outside the mobile objects that contains them. But when a mobile object moves, it transports all the objects in the inner configuration.

Modal Logics for the ambient calculus

In order to describe properties of mobile computations, a modal logic that can talk about spaces as well as time has been developed [9, 10]. In the ambient calculus context, it makes sense to talk about properties that hold at particular locations, and it becomes natural to consider *spatial modalities* for properties that hold at certain location, any some location, or at every location. Mobility is regarded as the evolution of spatial configurations over time. A specification logic for mobility should be able to talk about the structure of spatial configurations and about their evolution through time; that is, it should be a modal logic of space and time.

In Maude we can use the model checker to describe properties defined with a modal logic of time. Furthermore, a spatial logic to describe properties in Maude specifications has also been studied [47, 57].

1.1.4 The Seal calculus

The Seal calculus [67, 14] is a distributed process calculus with localities and mobility of computational entities called seals. Seal is also a framework for writing secure distributed applications over large scale open networks such as the Internet.

Seal is an extension of Milner's π-calculus [51] with the goal of being able to express the essential properties of Internet programs. This language is based in five design principles that are particularly suited to Internet programming:

- *No reliance on global state*. The size of the systems represented is of millions of hosts. At this size, the language cannot afford to assume any shared state.
- *Explicit localities*. The location where computation occurs and the location of its resources are essential to the efficiency and the fault tolerance of a distributed program.
- *Restricted connectivity*. Failures of machines and firewalls can cause that a computation can only communicate with a subset of the other entities on the network.
- *Dynamic configuration*. New hosts can be added to the network.
- *Access control*. The security is one of the most important features of all the applications on the Internet.

Seal provides a model of mobility which subsumes message passing, remote evaluation as well as process migration and which models user mobility and hardware mobility. Moreover, it provides a hierarchical protection model, in which each level can implement security policies. Seal unifies several concepts from distributed programming into three abstractions, namely *locations*, *processes*, and *resources*:

- Locations stand for physical places such as routers and logical boundaries such as protection domains.
- The processes stand for any flow of control like, for example, threads.
- Resources unify physical resources with services.

1.1.5 Mobile UNITY

Mobile UNITY is a model for specifying and reasoning about concurrent systems that contain dynamically reconfigurable components [61]. In Mobile UNITY, each program is a unit of mobility. It captures movement by augmenting the program state with a location attribute whose change in value is used to represent motion. The Mobile UNITY approach to mobile agents has been studied in [62].

The objective of the Mobile UNITY model is to develop techniques that facilitate the verification and design of mobile systems. This reasoning relies on extensions to the UNITY proof logic [53]. Safety and liveness properties can be proved by quantifying over the statements of the program text. Properties that have already be proven correct can be used to derive other properties without reference to the original program text by employing established UNITY rules of inference.

The Mobile UNITY approach is similar to ours, but we allow really distribution, so we do not need an attribute indicating the location, it is represented by each Maude process. Maude also allows to check properties, by using its built-in model checker, about distributed configuration, although in this case we need a "centralized" configuration (see Chapter 6) with explicitly defined bounds, and where mobility is defined by crossing them.

1.1.6 Mobile agents in Ajanta

Ajanta [64] is a Java-based system for programming mobile-agent applications on the Internet. The mechanisms supported by Ajanta include:

- Generic agent and server classes that can be easily extended for building agent-based applications.
- Mechanisms for protecting an agent's state while it travels over insecure networks, and to untrusted servers.
- A high-level programming abstraction based on the concept of composable patterns of migration for building agent itineraries. These patterns separate an agent's computation task from the specification of its migration path.
- Mechanisms for applications to monitor the status of their agents, and control them remotely. Applications can also provide mechanisms to handle exceptions.
- A location-independent global naming and name resolution mechanism that facilitates communication between mobile objects.

In Ajanta, the mobile agent implementation is based on the generic concept of a mobile object. Agents are active mobile objects, which encapsulate code and execution context along with data. Ajanta uses Java facilities such as object serialization, reflection, remote method invocation, and its security model. Agent mobility is implemented by using object serialization and dynamic class loading. The Ajanta programming primitives allow one to create and dispatch agents, control their mobility, monitor them, and recover from failures.

Ajanta introduces the concept of abstract migration patterns, which can be used to simplify the task of creating complex agent itineraries by composition of some basic patterns. These patters incorporate failure recovery for robustness.

Acknowledgements.

I thank Alberto Verdejo for his help in the development of this work, that he has read about googol times, giving with each review several good ideas, as well as for his friendship and patience throughout the last two years. I also thank Narciso Martí-Oliet for his continuous support, his help with my scholarship, and his office for almost a year!

Chapter 2

Maude

Maude [18, 19] is a language based on both equational and rewriting logic for the specification and implementation of a whole range of models and systems. It has already been used to specify and analyze distributed applications and protocols [27, 54]. The recently incorporated support in Maude for communication with external objects makes many other application areas (such as mobile computing and distributed agents) ripe for system development in Maude.

In this chapter, we first describe the Maude syntax for functional and system modules, and the Full Maude syntax for object-oriented modules used throughout the work. In Section 2.5 we explain the Maude reflective capabilities, that allow to use Maude modules as data, which will be used in Chapters 3 and 5. Parameterized modules, that will be used all over the work and specially in Chapter 4, are explained in Section 2.6. Sections 2.7 and 2.8 describe Maude sockets and buffered sockets, that will be used to implement all the distributed applications shown in this work. Finally, since Full Maude does not support sockets, Section 2.9 illustrates how to translate the specifications shown in the work to Core Maude.

2.1 Functional modules

In Maude the state of a system is formally specified as an algebraic data type by means of an equational specification. In this kind of specifications we can define new types (by means of keyword `sort(s)`); subtype relations between types (`subsort`); operators (`op`) for building values of these types, giving the types of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; and equations (`eq`) that identify terms built with these operators. Equations are assumed to be confluent and terminating, that is, we can use the equations to reduce a term t to a unique, canonical form t' that is equivalent to t (they represent the same value). These specifications are introduced in *functional* modules, with syntax `fmod...endfm`. For example, we can create a module with the definition of the natural numbers.

```
fmod NATURAL-NUMBERS is
```

This module defines two sorts: `Nat`, for all the natural numbers, and `NzNat`, for the natural numbers different from `0`. We declare the set of the non-zero natural numbers as a subsort of the natural numbers.

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

The constant `0` is a natural number.

```
  op 0 : -> Nat .
```

The constructor successor (declared with notation `s`) receives as argument a value of sort `Nat` and returns a value of sort `NzNat`.

```
  op s : Nat -> NzNat .
```

We define the addition operator with infix notation, where the underscores indicate where the arguments must be placed, being associative and commutative.

```
  op _+_ : Nat Nat -> Nat [assoc comm].
```

Finally, the behavior of the addition function is given by means of equations. Since it has been declared commutative, the first equation can match terms with `0` in both the right and the left side of the operator.

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

Maude uses a very expressive version of equational logic, namely *membership equational logic* [5, 50], that (in addition to all the above) allows the statement of *membership assertions* (`mb`) characterizing the elements of a sort. For example, we can extend the `NATURAL-NUMBERS` module with the following two memberships, that specify when a natural number is even.

```
  sort Even .
  subsort Even < Nat .

  mb 0 : Even .
  mb s(s(E:Even)) : Even .
```

Membership equational logic also has a notion of (implicit) error supersorts called *kinds*, which are represented in Maude as sort names between square brackets. Using kinds, we can declare partial operations (at the level of sorts), like for example the following integer division operation on natural numbers:

```
  op _div_ : Nat Nat -> [Nat] .
```

2.2 System modules

The *dynamic* behavior of a system is specified by rewrite rules, that can take the most general possible form in the variant of rewriting logic built on top of membership equational logic [7], that is, they can be of the form

$$t \longrightarrow t' \quad \textit{if} \quad (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \longrightarrow q_k)$$

with no restriction on which new variables may appear in the righthand side or in the condition. Conditions in rules are formed by an associative conjunction connective `/\`, allowing equations (both ordinary equations `t = t'`, and matching equations `t := t'` where new variables occurring in `t` become instantiated by matching [17, 19]), memberships (`t : s`), and rewrites (`t => t'`) as conditions.

These rules describe the local, concurrent transitions of the system. That is, when a part of a system matches the pattern t, it can be transformed into the corresponding instance of the pattern t' if the conditions are satisfied. Rewrite rules are included in *system* modules, with syntax `mod...endm`. For example, we can define a vending machine that sells apples (denoted with the constant `a`) and cakes (`c`), and accepts dollars (`$`) and quarters (`q`), all these constants with sort `Marking`.

```
mod VENDING-MACHINE is
 sort Marking .
 ops a c $ q : -> Marking .
```

These constants stand together in an associative and commutative "soup" by means of the juxtaposition operator `__`.

```
 op __ : Marking Marking -> Marking [assoc comm] .
```

This vending machine has a great limitation: it only accepts dollars. We can buy a cake for one dollar, and pay one dollar for an apple and receive one quarter as change. To partially solve this problem, the machine can change four quarters for a dollar. This behavior is modeled by the following rules:

```
 rl [buy-c] : $ => c .
 rl [buy-a] : $ => a q .
 rl [change] : q q q q => $ .
endm
```

2.3 Object-oriented modules

Regarding object-oriented specifications [49], *classes* are declared with the syntax `class` C `|` a_1`:`S_1`,`$\ldots$`,` a_n`:`S_n, where C is the class name, a_i is an attribute identifier, and S_i is the sort of the values this attribute can have. An *object* in a given state is represented as a term `<` O `:` C `|` a_1 `:` v_1`,` `...`, a_n `:` v_n `>` where O is the object's name, belonging to a set `Oid` of object identifiers, and the v_i's are the current values of its attributes. *Messages* are defined by the user for each application (introduced with syntax `msg`). Subclass relations can also be defined, with syntax `subclass`. We show as example a distance education school.

```
omod SCHOOL-SYNTAX is
 pr STRING .
```

We define the class `Person`, with attributes `name` and `age`.

```
 class Person | name : String, age : Nat .
```

In this very simple example, each student is modeled as an object of class `Student`, that is a subclass of `Person`, and adds a `score` attribute.

```
  class Student | score : Nat .
  subclass Student < Person .
```

Each teacher keeps a set of students (identified by their object identifier, of sort `Oid`) and has an state: `before` and `after` sending the exams.

```
  sort OidSet .
  subsort Oid < OidSet .
  op mtOidSet : -> OidSet .
  op __ : OidSet OidSet -> OidSet [comm assoc id: mtOidSet] .

  sort State .
  ops before after : -> State .

  class Teacher | students : OidSet, state : State .
  subclass Teacher < Person .
```

The messages interchanged between the teacher and the students are:

- the teacher sends the exam to a student;

  ```
  msg to_from_exam : Oid Oid -> Msg .
  ```

- the student solves the exam and sends it back to the teacher; and

  ```
  msg to_from_solution : Oid Oid -> Msg .
  ```

- the teacher sends the score to the student.

  ```
   msg to_score_ : Oid Nat -> Msg .
  endom
  ```

In a concurrent object-oriented system the concurrent state, which is called a *configuration*, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting (modulo the multiset structural axioms of associativity, commutativity, and identity) using rules that describe the effects of *communication events* between some objects and messages. The rewrite rules in the module specify in a declarative way the behavior associated with the messages. The general form of such rules is

$$
\begin{aligned}
& M_1 \ldots M_n \ \langle O_1 : F_1 \mid atts_1 \rangle \ldots \langle O_m : F_m \mid atts_m \rangle \\
& \longrightarrow \langle O_{i_1} : F'_{i_1} \mid atts'_{i_1} \rangle \ldots \langle O_{i_k} : F'_{i_k} \mid atts'_{i_k} \rangle \ \langle Q_1 : D_1 \mid atts''_1 \rangle \ldots \langle Q_p : D_p \mid atts''_p \rangle \\
& \quad M'_1 \ldots M'_q \quad \textit{if } C
\end{aligned}
$$

where $k, p, q \geq 0$, the M_s are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, and C is a rule condition. The result of applying this rule is that the messages $M_1, \ldots, M_n$ disappear; the state and possibly the class of the objects $O_{i_1}, \ldots, O_{i_k}$ may change; all the other objects O_j vanish; new objects $Q_1, \ldots, Q_p$ are created; and new messages $M'_1, \ldots, M'_q$ are sent. An important special case are rules with a single object and at most one message on the lefthand side. These are called *asynchronous* rules. They directly model asynchronous distributed interactions. Rules involving multiple objects are called *synchronous*; they are used to model higher-level communication abstractions.

By convention, the only object attributes made explicit in a rule are those relevant for that rule. In particular, the attributes mentioned only in the lefthand side of the rule are preserved unchanged, the original values of attributes mentioned only in the righthand side of the rule do not matter, and all attributes not explicitly mentioned are left unchanged. We use here the Full Maude object-oriented notation [19]. However, the implementation of the applications shown in this work is in Core Maude because Full Maude does not support external objects (see Section 2.7).

We show now how the behavior of the school described above can be modeled with rules.

```
omod SCHOOL is
 pr SCHOOL-SYNTAX .
```

First, the teacher sends the exams to its students. We use an auxiliary function `broadcastExam` that broadcasts the message to all of them. Notice that the `students` attribute only appears in the lefthand side of the rule because it is not changed, while `state` appears in both sides because we require its value to be `before` in the lefthand side and then updated to `after` in the righthand side.

```
 vars TCHR STDNT : Oid .
 var  OS : OidSet .
 var  N : Nat .

 rl [exam] :
    < TCHR : Teacher | students : OS, state : before >
 => < TCHR : Teacher | state : after >
    broadcastExam(OS, TCHR) .

 op broadcastExam : OidSet Oid -> Configuration .
 eq broadcastExam(mtOidSet, TCHR) = none .
 eq broadcastExam(STDNT OS, TCHR) = to STDNT from TCHR exam
                                   broadcastExam(OS, TCHR) .
```

When a student receives the exam, it solves it and sends the answer back to the teacher.

```
 rl [solve] :
    to STDNT from TCHR exam
    < STDNT : Student | >
 => < STDNT : Student | >
    to TCHR from STDNT solution .
```

The teacher corrects each exam when it is received, and sends the score to the student. The score is calculated non-deterministically by using one of the following rules:

```
 rl [correction] :
    to TCHR from STDNT solution
    < TCHR : Teacher | >
 => < TCHR : Teacher | >
    to STDNT score 0 .

 rl [correction] :
    to TCHR from STDNT solution
    < TCHR : Teacher | >
```

```
  => < TCHR : Teacher | >
     to STDNT score 5 .

  rl [correction] :
     to TCHR from STDNT solution
     < TCHR : Teacher | >
  => < TCHR : Teacher | >
     to STDNT score 10 .
```

Finally, when the student receives the score, it keeps it in the corresponding attribute. Notice that the `score` attribute only appears in the righthand side of the rule, because it previous value does not mind for the application of the rule.

```
  rl [score] :
     (to STDNT score N)
     < STDNT : Student | >
  => < STDNT : Student | score : N > .
 endom
```

2.4 Maude main commands

We show here briefly the main Maude commands used in this work, for more information about Maude commands see [18, 19]. First, we can evaluate expressions defined by means of equations by using the `reduce` command (abbreviated as `red`). Maude will print the result, prefaced by its least sort. For example, we can evaluate the expression `3 + 7` in the predefined module `NAT` of natural numbers (the module can be omitted if it is the last one loaded into Maude).

```
reduce in NAT : 3 + 7 .
```

Maude elicits the response

```
reduce in NAT : 3 + 7 .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 10
```

We can use the `rewrite` command (abbreviated as `rew`) to explore the behavior of a system module by using a leftmost, outermost rule fair strategy. The `frewrite` command (abbreviated as `frew`) rewrites a term using a depth-first position-fair strategy that makes it possible for some rules to be applied that could be starved using the `rewrite` strategy. For example, given the module `VENDING-MACHINE` from Section 2.2, we can rewrite a marking composed by two dollars with the following command:

```
  rew $ $ .
```

The response given by Maude is:

```
  rewrite in VENDING-MACHINE : $ $ .
  rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
  result Marking: a c q
```

While the `rewrite` and `frewrite` commands explore just one possible behavior (sequence of rewrites) of a system, the `search` command allows to explore (following a breadth-first strategy) the reachable state space. This command provides several options like search in 0 or more steps (=>*), in 1 or more steps (=>+), and search for final states (`=>!`). For example, if we want to know if we can obtain three apples with four dollars we use the command:

```
search [1] $ $ $ $ =>* a a a M:Marking .
```

where `[1]` stands for the first solution, and `M:Marking` matches the rest of the marking. The response obtained is:

```
Solution 1 (state 9)
states: 10  rewrites: 12 in 0ms cpu (2ms real) (~ rewrites/second)
M:Marking --> q q q $
```

That is, we can buy three apples and obtain one dollar and three quarters of change. The `search` command can also be used to check invariants, as we will see in Section 6.1.

2.5 Reflection and metalevel computations

Informally, a reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. In other words, a reflective logic is a logic which can be faithfully interpreted in itself. Maude implementation makes systematic use of the fact that rewriting logic is reflective [18]. We explain here the semantic principles and implementation techniques through which efficient ways of performing reflective computations are achieved in Maude through its predefined `META-LEVEL` module.

The `META-LEVEL` module uses the following modules:

- `META-TERM`, where sorts and kinds are metarepresented as data in specific subsorts (`Sort` and `Kind`, both subsorts of `Type`) of the sort `Qid` of quoted identifiers, and terms are metarepresented as elements of the data type `Term` of terms; and

- `META-MODULE`, where functional and system modules are metarepresented data of the sorts `FModule`, `SModule`, being the sort `Module` the supersort of both of them.

It has several built-in descent functions that provide useful and efficient ways of reducing metalevel computations to object-level ones, as well as several useful operations on sorts and kinds.

One of the most important functions we are going to use is `upModule`, that takes as arguments the metarepresentation of the name of a module and a Boolean value, and returns the metarepresentation of the module. The second argument indicates if the data imported by the module is also metarepresented. In a similar way, we can "move up" a term by using the function `upTerm`, that takes a term and returns the metarepresentation of its canonical form.

We can reduce the `Term` `t` in the `Module` `M` by means of the function `metaReduce(M, t)`, that returns the metarepresentation of the canonical form of `t`, using the equations in `M`, together with the metarepresentation of its corresponding sort or kind.

```
sort ResultPair .
op {_,_} : Term Type -> ResultPair [ctor] .
op metaReduce : Module Term ~> ResultPair [special (...)] .
```

The (partial) operation `metaRewrite` takes as arguments the metarepresentation of a module `M`, the metarepresentation of a term `t`, and a value `b` of the sort `Bound`, i.e., either a natural number or the constant `unbounded`.

```
sort Bound .
subsort Nat < Bound .
op unbounded :-> Bound [ctor] .
op metaRewrite : Module Term Bound ~> ResultPair [special (...)] .
```

The result of `metaRewrite(M, t, b)` is the metarepresentation of the term obtained from `t` after at most `b` applications of the rules in `M` using the `rewrite` strategy, together with the metarepresentation of its corresponding sort or kind.

Finally, the operation `metaSearch` takes as arguments the metarepresentation of a module, the metarepresentation of the starting term for search, the metarepresentation of the pattern to search for, the metarepresentation of a condition to be satisfied, the kind of search to carry on, a `Bound` value, and a natural number.

```
op metaSearch :
Module Term Term Condition Qid Bound Nat ~> ResultTriple? [special (...)] .
```

The searching strategy used by `metaSearch` coincides with the object-level `search` command in Maude. The `Qid` values that are allowed as arguments are: `'*` for a search involving zero or more rewrites (corresponding to =>* in the search command), `'+` for a search consisting in one or more rewrites (=>+), and `'!` for a search that only matches canonical forms (=>!). The `Bound` argument indicates the maximum depth of the search, and the `Nat` argument is the solution number. To indicate a search consisting in exactly one rewrite, we set the maximum depth of the search to the number 1.

2.6 Parameterized modules

Most of the data types that we consider are generic, that is, they are constructions on top of other data types that appear as parameters in the construction. Therefore, these specifications are *parameterized*.

For example, lists can be constructed on top of any data, but sorted lists only make sense for data that have a total order; for a binary operation to be a total order, several properties have to be satisfied, which are written in the corresponding parameter theory as equations. Parameterized modules use *theories* to specify the requirements that the parameter must satisfy. A (functional) theory is also a membership equational specification but since its equations are not used for equational simplication, they need not satisfy any requirement about variables in the righthand side, confluence, or termination.

The simplest theory is the one requiring just the existence of a sort, as follows:

```
fth TRIV is
 sort Elt .
endfth
```

This theory is used as requirement for the parameter of parameterized data types such as stacks, queues, lists, multisets, sets, and binary trees.

A more complex theory is the following, requiring a (strict) total order over elements of a given sort. Notice the *new* variable E2 in the righthand side of the first conditional equation. This makes this equation non-executable, as stated by the attribute nonexec next to the equation.

```
fth TOSET< is
  protecting BOOL .
  sort Elt .
  ops _<_ _>_ : Elt Elt -> Bool .
  vars E1 E2 E3 : Elt .
  eq E1 < E1 = false .
  ceq E1 < E3 = true if E1 < E2 and E2 < E3 [nonexec] .
  ceq E1 < E2 or E2 < E1 = true if E1 =/= E2 .
  eq E1 > E2 = E2 < E1 .
endfth
```

This theory imports in protecting mode the predefined module BOOL of Boolean values, meaning that the Boolean values are not disturbed in any way. In addition to the usual Boolean values and operations, the module BOOL adds equality _==_ and inequality _=/=_ operations on the sort Elt.

Theories are used in a parameterized module as in the following example:

```
fmod EXAMPLE{X :: TRIV} is  ...  endfm
```

where X :: TRIV denotes that X is the label of the formal parameter, and that it must be instantiated with modules satisfying the requirement expressed by the theory TRIV. The way to express this instantiation is by means of *views*. A view shows how a particular module satisfies a theory, by mapping sorts and operations in the theory to sorts and operations (or, more generally, terms) in the target module, in such a way that the induced translations on equations and membership axioms are provable in the module. In general, this requires theorem proving that is not done by the system, so the user must take care of it. However, in many simple cases the proof of obligations associated to views is completely obvious, as for example in the following view from the theory TRIV to the predefined module NAT of natural numbers, where, since TRIV has no equations, no proof obligations are generated.

```
view Nat from TRIV to NAT is
  sort Elt to Nat .
endv
```

Then, the module expression EXAMPLE{Nat} denotes the instantiation of the parameterized module EXAMPLE{X :: TRIV} by means of the above view Nat.

2.6.1 Predefined parameterized modules

We show here the predefined parameterized modules that will be used in the following chapters. The LIST{X :: TRIV} module constructs *lists* over a given sort of elements (provided by the theory TRIV). The constructors used are nil for the empty list and __ for list concatenation, which is associative and has nil as its identity. The sorts provided by this module are `List{X}` for general lists and `NeList{X}` for non-empty lists, where `X` is the sort of elements received as parameter. It also provides several functions over lists such as `size`, `occurs`, and `append`.

```
fmod LIST{X :: TRIV} is
 protecting NAT .
 sorts NeList{X} List{X} .
 subsort X$Elt < NeList{X} < List{X} .

 op nil : -> List{X} [ctor] .
 op __ : List{X} List{X} -> List{X} [ctor assoc id: nil prec 25] .
 op __ : NeList{X} List{X} -> NeList{X} [ctor ditto] .
 op __ : List{X} NeList{X} -> NeList{X} [ctor ditto] .

  ...
endfm
```

We can create for example lists of natural numbers by importing in a module the parameterized `LIST` module instantiated with the view `Nat`.

```
fmod NAT-LIST is
  pr LIST{Nat} .
endfm
```

A very similar parameterized module is `SET{X :: TRIV}`. This module constructs *sets* over the sort received in its parameter, with constructors `empty` for the empty set and `_,_` for the union, which is associative and commutative and has `empty` as identity, while the idempotence property is fulfilled by means of an equation.

```
fmod SET{X :: TRIV} is
 protecting EXT-BOOL .
 protecting NAT .
 sorts NeSet{X} Set{X} .
 subsort X$Elt < NeSet{X} < Set{X} .

 op empty : -> Set{X} [ctor] .
 op _,_ : Set{X} Set{X} -> Set{X} [ctor assoc comm id: empty prec 121] .
 op _,_ : NeSet{X} Set{X} -> NeSet{X} [ctor ditto] .

 eq N, N = N . *** Idempotence

 ...
endfm
```

Partial functions are defined in the module `MAP{X :: TRIV, Y :: TRIV}`, where `X` defines the sort of the keys while `Y` defines the sort of the values. Maps are constructed with the associative and commutative operator `_,_` and the constant `empty`. The constructor for each `Entry` (subsort of `Map`) is `_|->_`. When a map is consulted about a key that does not appear in the map, it returns the special value `undefined`, a constant defined at kind level `[Y$Elt]`.

```
fmod MAP{X :: TRIV, Y :: TRIV} is
  protecting BOOL .
  sorts Entry{X,Y} Map{X,Y} .
  subsort Entry{X,Y} < Map{X,Y} .

  op _|->_ : X$Elt Y$Elt -> Entry{X,Y} [ctor] .
  op empty : -> Map{X,Y} [ctor] .
  op _,_ : Map{X,Y} Map{X,Y} -> Map{X,Y} [ctor assoc comm id: empty prec 121] .
  op undefined : -> [Y$Elt] [ctor] .
```

Maps offer the functions `insert`, to update the map, and `_[_]`, to look up for the value associated to the given key.

```
  var D : X$Elt .
  vars R R' : Y$Elt .
  var M : Map{X,Y} .

  op insert : X$Elt Y$Elt Map{X,Y} -> Map{X,Y} .
  eq insert(D, R, (M, D |-> R')) =
     if $hasMapping(M, D) then insert(D, R, M)
     else (M, D |-> R)
     fi .
  eq insert(D, R, M) = (M, D |-> R) [owise] .

  op _[_] : Map{X,Y} X$Elt -> [Y$Elt] [prec 23] .
  eq (M, D |-> R)[D] =
     if $hasMapping(M, D) then undefined
     else R
     fi .
  eq M[D] = undefined [owise] .

  op $hasMapping : Map{X,Y} X$Elt -> Bool .
  eq $hasMapping((M, D |-> R), D) = true .
  eq $hasMapping(M, D) = false [owise] .
endfm
```

For example, a map from natural numbers to integers is defined as follows:

```
 fmod EXAMPLE is
  pr MAP{Nat, Int} .
 endfm
```

where `Int` is a view, from `TRIV` to the predefined module of integer numbers `INT`, mapping the sort `Elt` to the sort `Int`.

2.7 Sockets provided by Maude

Since version 2.2, Maude supports communication with external objects by means of TCP sockets, which allows the implementation of real distributed applications. Currently only IPv4 TCP sockets are supported; however, other protocol families and socket types may be added in the future. This section explains Maude's support for rewriting with external objects and an implementation of sockets as the first such external objects. Most of the material in this section has been extracted from [19].

Configurations that want to communicate with external objects must contain at least one *portal*, where

```
  sort Portal .
  subsort Portal < Configuration .
  op <> : -> Portal [ctor] .
```

is part of the predefined module CONFIGURATION in the file prelude.maude. Rewriting with external objects is started by the external rewrite command erewrite (abbreviated as erew), which behaves as frewrite (see Section 2.4) and allows messages to be exchanged with external objects that do not reside in the configuration. The expected behavior of these (external objects) sockets is formally defined by means of rewrite rules in Section 6.2.

Note that, even if there are no more possible rewrites, erewrite may not terminate; if there are requests made to external objects that have not yet been fulfilled because of waiting for external events from the operating system, the Maude interpreter will suspend until at least one of those events occurs, at which time rewriting will resume. Due to this fact, it is recommended to execute these applications with the trace on. In this way, we can see what is happening in each Maude process. When the execution of a concrete example seems to be finished, because we do not see evolution in any of the involved processes, we can finish it by typing ^C.

The first example of external objects is *sockets*, which are declared in the SOCKET module, included in the file socket.maude which is part of the Maude distribution. The external object named by the socketManager constant is a factory for socket objects. Almost everything in the socket implementation is done in a nonblocking way; so, for example, if you try to open a connection to some webserver and that webserver takes 5 minutes to respond, other rewriting and transactions may happen in the meanwhile as part of the same command erewrite. The one exception is DNS resolution, which is done as part of the createClientTcpSocket message handling and which cannot be nonblocking without special tricks.

2.7.1 Client sockets

To create a client socket, you send socketManager a message

```
  createClientTcpSocket(socketManager, ME, ADDRESS, PORT)
```

where ME is the name of the object the reply should be sent to, ADDRESS is the name of the server you want to connect to (say "www.google.com"), and PORT is the port you want to connect to (say 80 for HTTP connections). You may also specify the name of the server as an IPv4 dotted address or as "localhost" for the same machine where the Maude system is running on.

The reply will be either

```
  createdSocket(ME, socketManager, NEW-SOCKET-NAME)
```

or

```
  socketError(ME, socketManager, REASON)
```

where NEW-SOCKET-NAME is the name of the newly created socket (an object identifier of sort Oid) and REASON is the operating system's terse explanation of what went wrong.

You can then send data to the server with a message

```
send(SOCKET-NAME, ME, DATA)
```

which elicits either

```
sent(ME, SOCKET-NAME)
```

or

```
closedSocket(ME, SOCKET-NAME, REASON)
```

Notice that all errors on a client socket are handled by closing the socket.

Similarly, you can receive data from the server with a message

```
receive(SOCKET-NAME, ME)
```

which elicits either

```
received(ME, SOCKET-NAME, DATA)
```

or

```
closedSocket(ME, SOCKET-NAME, REASON)
```

When you are done with the socket, you can close it with a message

```
closeSocket(SOCKET-NAME, ME)
```

with reply

```
closedSocket(ME, SOCKET-NAME, "")
```

Once a socket has been closed, its name may be reused, so sending messages to a closed socket can cause confusion and should be avoided.

Notice that TCP does not preserve message boundaries, so sending "one" and "two" might be received as "on" and "etwo". Delimiting message boundaries is the responsibility of the next higher-level protocol, such as HTTP. We will present an implementation of buffered sockets in Section 2.8 which solves this problem.

In [19] an implementation using sockets of a HTTP/1.0 client that requests one web page to a HTTP server is shown.

2.7.2 Server sockets

To have communication between two Maude interpreter instances, one of them must take the server role and offer a service on a given port; generally ports below 1024 are protected. You cannot in general assume that a given port is available for use. To create a server socket, you send `socketManager` a message

```
createServerTcpSocket(socketManager, ME, PORT, BACKLOG)
```

where PORT is the port number and BACKLOG is the number of queue requests for connection that you will allow (5 seems to be a good choice). The response is either

```
  createdSocket(ME, socketManager, SERVER-SOCKET-NAME)
```

or

```
  socketError(ME, socketManager, REASON)
```

Here SERVER-SOCKET-NAME refers to a server socket. The only thing you can do with a server socket (other than close it) is to accept clients, by means of the following message:

```
  acceptClient(SERVER-SOCKET-NAME, ME)
```

which elicits either

```
  acceptedClient(ME, SERVER-SOCKET-NAME, ADDRESS, NEW-SOCKET-NAME)
```

or

```
  socketError(ME, socketManager, REASON)
```

Here ADDRESS is the originating address of the client and NEW-SOCKET-NAME is the name of the socket you use to communicate with that client. This new socket behaves just like a client socket for sending and receiving. Note that an error in accepting a client does not close the server socket. You can always reuse the server socket to accept new clients until you explicitly close it.

2.7.3 Factorial server example

The following modules illustrate a very naive two-way communication between two Maude interpreter instances.[1] The issues of port availability and message boundaries are deliberately ignored for the sake of illustration (and thus if you are unlucky this example could fail).

The first module describes the behavior of the server.

```
 omod FACTORIAL-SERVER is
  inc SOCKET .
  pr CONVERSION .

  op _! : Nat -> NzNat .
  eq 0 ! = 1 .
  eq (s N) ! = (s N) * (N !) .

  class Server | .
  op aServer : -> Oid .
```

[1]In this section and the following ones we use the (more convenient) *object-oriented* notation provided by Full Maude [19, Chapter 14]. However, since Full Maude does not support external objects yet, this notation has to be translated (in a straightforward way) to Core Maude *object-based* notation in system modules. The transformation process will be explained in Section 2.9.

Using the following rules, the server waits for clients. If one client is accepted, the server waits for messages from it. When the message arrives, the server converts the received data to a natural number, computes its factorial, converts it into a string, and finally sends this string to the client. Once the message is sent, the server closes the socket with the client.

```
  vars SERVER SOCKET-MANAGER SOCKET NEW-SOCKET LISTENER : Oid .
  var  N : Nat .
  var  REASON IP NUMBER : String .

  rl [createdSocket] :
     createdSocket(SERVER, SOCKET-MANAGER, LISTENER)
     < SERVER : Server | >
  => < SERVER : Server | >
     acceptClient(LISTENER, SERVER) .

  rl [acceptedClient] :
     acceptedClient(SERVER, LISTENER, IP, NEW-SOCKET)
     < SERVER : Server | >
  => < SERVER : Server | > receive(NEW-SOCKET, SERVER)
     acceptClient(LISTENER, SERVER) .

  rl [received] :
     received(SERVER, SOCKET, NUMBER)
     < SERVER : Server | >
  => < SERVER : Server | >
     send(SOCKET, SERVER, string(rat(NUMBER, 10)!, 10)) .

  rl [sent] :
     sent(SERVER, SOCKET)
     < SERVER : Server | >
  => < SERVER : Server | >
     closeSocket(SOCKET, SERVER) .

  rl [closedSocket] :
     closedSocket(SERVER, SOCKET, REASON)
     < SERVER : Server | >
  => < SERVER : Server | > .
 endom
```

The Maude command that initializes the server is as follows, where the configuration includes the portal <>.

```
  Maude> erew <> < aServer : Server | none >
              createServerTcpSocket(socketManager, aServer, 8811, 5) .
```

The second module describes the behavior of the clients.

```
 omod FACTORIAL-CLIENT is
  inc SOCKET .

  class Client | .
  op aClient : -> Oid .
```

Using the following rules, the client connects to the server (clients must be created after the server), sends a message representing a number,[2] and then waits for the response. When the response arrives, there are no blocking messages and rewriting ends.

```
  vars CLIENT SOCKET-MANAGER NEW-SOCKET SOCKET : Oid .
  var  N : Nat .

  rl [createdSocket] :
     createdSocket(CLIENT, SOCKET-MANAGER, NEW-SOCKET)
     < CLIENT : Client | >
  => < CLIENT : Client | >
     send(NEW-SOCKET, CLIENT, "6") .

  rl [sent] :
     sent(CLIENT, SOCKET)
     < CLIENT : Client | >
  => < CLIENT : Client | >
     receive(SOCKET, CLIENT) .
 endom
```

The initial configuration for the client will be as follows, again with portal <>.

```
  Maude> erew <> < aClient : Client | none >
              createClientTcpSocket(socketManager, aClient, "127.0.0.1", 8811) .
```

2.8 Buffered sockets

As we said before, TCP does not preserve message boundaries; to guarantee it we have implemented a filter class `BufferedSocket`, defined in the module `BUFFERED-SOCKET`.

When a buffered socket is created, in addition to the socket object through which the information will be sent, a `BufferedSocket` object is also created on each side of the socket (one in each one of the configurations between which the communication is established). All messages sent through a buffered socket are manipulated before they are sent through the socket underneath. When a message is sent through a buffered socket, a mark is placed at the end of it; the `BufferedSocket` object at the other side of the socket stores all messages received on a buffer, in such a way that when a message is requested the marks placed indicate which part of the information received must be given as the next message.

An object of class `BufferedSocket` has two attributes: `read`, of sort `String`, which stores the concatenation of the strings already received but not handled yet, and `complete`, that keeps information relative to the fact that a complete message (with the mark) has already arrived.

```
 omod BUFFERED-SOCKET is
  inc SOCKET .

  class BufferedSocket | read : String, complete : FindResult .
```

The identifiers of the `BufferedSocket` objects are marked with a `b` operator, i.e., the buffers associated with a socket `SOCKET` have identifier `b(SOCKET)`. Note that there is a

[2]In this quite simple example, it is always the number 6 already represented as the string "6".

`BufferedSocket` object on each side of the socket, that is, may be two objects with the same identifier, but in different configurations.

```
op b : Oid -> Oid .
```

We interact with buffered sockets in the same way we interact with sockets, with the only difference that all messages in the module `SOCKET` have been capitalized to avoid confusion. Thus, to create a client with a buffered socket, you send to the `socketManager` a message

```
CreateClientTcpSocket(socketManager, ME, ADDRESS, PORT)
```

instead of a message

```
createClientTcpSocket(socketManager, ME, ADDRESS, PORT).
```

All the messages have exactly the same declarations, the only difference being their initial capitalization:

```
msg CreateClientTcpSocket : Oid Oid String Nat -> Msg .
msg CreateServerTcpSocket : Oid Oid Nat Nat -> Msg .
msg CreatedSocket : Oid Oid Oid -> Msg .

msg AcceptClient : Oid Oid -> Msg .
msg AcceptedClient : Oid Oid String Oid -> Msg .

msg Send : Oid Oid String -> Msg .
msg Sent : Oid Oid -> Msg .

msg Receive : Oid Oid -> Msg .
msg Received : Oid Oid String -> Msg .

msg CloseSocket : Oid Oid -> Msg .
msg ClosedSocket : Oid Oid String -> Msg .

msg SocketError : Oid Oid String -> Msg .
```

For most of these messages, a buffered socket just converts them into the corresponding uncapitalized message.

```
vars SOCKET NEW-SOCKET SOCKET-MANAGER O : Oid .
vars ADDRESS IP IP' DATA S S' REASON : String .
vars PORT BACKLOG N : Nat .
var  FR : FindResult .

rl [CreateServerTcpSocket] :
   CreateServerTcpSocket(SOCKET-MANAGER, O, PORT, BACKLOG)
=> createServerTcpSocket(SOCKET-MANAGER, O, PORT, BACKLOG) .

rl [AcceptClient] :
   AcceptClient(SOCKET, O)
=> acceptClient(SOCKET, O) .
```

```
rl [CloseSocket] :
   CloseSocket(b(SOCKET), SOCKET-MANAGER)
=> closeSocket(SOCKET, SOCKET-MANAGER) .

rl [CreateClientTcpSocket] :
   CreateClientTcpSocket(SOCKET-MANAGER, O, ADDRESS, PORT)
=> createClientTcpSocket(SOCKET-MANAGER, O, ADDRESS, PORT) .
```

Note that in these cases the buffered socket versions of the messages are just translated into the corresponding socket messages.

A `BufferedSocket` object can also convert an uncapitalized message into the capitalized one. The rule `socketError` shows this:

```
rl [socketError] :
   socketError(O, SOCKET-MANAGER, REASON)
=> SocketError(O, SOCKET-MANAGER, REASON) .
```

`BufferedSocket` objects are created and destroyed when the corresponding sockets are, and they start listening as soon as they are created. Thus, we have rules

```
rl [createdSocket] :
   createdSocket(O, SOCKET-MANAGER, SOCKET)
=> < b(SOCKET) : BufferedSocket | read : "", complete : notFound >
   CreatedSocket(O, SOCKET-MANAGER, b(SOCKET))
   receive(SOCKET, b(SOCKET)) .

rl [acceptedclient] :
   acceptedClient(O, SOCKET, IP', NEW-SOCKET)
=> AcceptedClient(O, b(SOCKET), IP', b(NEW-SOCKET))
   < b(NEW-SOCKET) : BufferedSocket | read : "", complete : notFound >
   receive(NEW-SOCKET, b(NEW-SOCKET)) .

rl [closedSocket] :
   closedSocket(O, SOCKET, DATA)
   < b(SOCKET) : BufferedSocket | >
=> ClosedSocket(O, SOCKET, DATA) .
```

Once a connection has been established, and a `BufferedSocket` object has been created on each side, messages can be sent and received. When a `Send` message is received by a buffered socket, it converts it in a `send` message with the same data plus a mark[3] to indicate the end of the message.

```
rl [Send] :
   Send(b(SOCKET), O, DATA)
   < b(SOCKET) : BufferedSocket | >
=> < b(SOCKET) : BufferedSocket | >
   send(SOCKET, O, DATA + "#") .

rl [sent] :
   sent(O, SOCKET)
=> Sent(O, b(SOCKET)) .
```

[3]We use the character '#' as the mark; therefore, the user data sent through the sockets should not contain such a character.

The key is then in the reception of messages. A `BufferedSocket` object is always listening through the associated socket. A `Receive` message is then handled if there is a complete message in the buffer (the number `N` in the `complete` attribute is the position of the mark), and then the part of the string before the mark is put in a `Received` message, updating the corresponding attributes.

```
  op getComplete : FindResult String -> FindResult .
  eq getComplete(N, S) = N .
  eq getComplete(notFound, S) = find(S, "#", 0) .

  rl [received] :
     received(b(SOCKET), O, DATA)
     < b(SOCKET) : BufferedSocket | read : S, complete : FR >
  => < b(SOCKET) : BufferedSocket | read : (S + DATA),
                                    complete : getComplete(FR, S + DATA) >
     receive(SOCKET, b(SOCKET)) .

  crl [Receive] :
      Receive(b(SOCKET), O)
      < b(SOCKET) : BufferedSocket | read : S, complete : N >
   => < b(SOCKET) : BufferedSocket | read : S', complete : find(S', "#", 0) >
      Received(O, b(SOCKET), DATA)
   if DATA := substr(S, 0, N) /\
      S' := substr(S, N + 1, length(S)) .
endom
```

2.8.1 The factorial example revisited

We illustrate how to use buffered sockets with the factorial example shown above. Both modules `FACT-SERVER` and `FACT-CLIENT` must include the `BUFFERED-SOCKET` module instead of `SOCKET`, and translate its messages, capitalizing their first letter.

```
 omod FACT-SERVER is
  pr BUFFERED-SOCKET .

  ...

  rl [CreatedSocket] :
     CreatedSocket(SERVER, SOCKET-MANAGER, LISTENER)
     < SERVER : Server | >
  => < SERVER : Server | >
     AcceptClient(LISTENER, SERVER) .

  rl [AcceptedClient] :
     AcceptedClient(SERVER, LISTENER, IP, NEW-SOCKET)
     < SERVER : Server | >
  => < SERVER : Server | >
     Receive(NEW-SOCKET, SERVER)
     AcceptClient(LISTENER, SERVER) .

  rl [Received] :
     Received(SERVER, SOCKET, NUMBER)
     < SERVER : Server | >
  => < SERVER : Server | >
```

```
     Send(SOCKET, SERVER, string(rat(NUMBER, 10)!, 10)) .

  rl [Sent] :
     Sent(SERVER, SOCKET)
     < SERVER : Server | >
  => < SERVER : Server | >
     CloseSocket(SOCKET, SERVER) .

  rl [ClosedSocket] :
     ClosedSocket(SERVER, SOCKET, REASON)
     < SERVER : Server | >
  => < SERVER : Server | > .
 endom

 omod FACT-CLIENT is
  pr BUFFERED-SOCKET .

  ...

  rl [CreatedSocket] :
     CreatedSocket(CLIENT, SOCKET-MANAGER, NEW-SOCKET)
     < CLIENT : Client | >
  => < CLIENT : Client | >
     Send(NEW-SOCKET, CLIENT, "6") .

  rl [Sent] :
     Sent(CLIENT, SOCKET)
     < CLIENT : Client | >
  => < CLIENT : Client | >
     Receive(SOCKET, CLIENT) .
 endom
```

2.9 From Full Maude to Core Maude

In this section we illustrate how Full Maude object-oriented specifications can be translated to Core Maude ones. We use the distance school example shown in Section 2.3.

First, we show how to transform the SCHOOL-SYNTAX module. Since Full Maude object-oriented modules imports implicitly the module CONFIGURATION, we must import it now explicitly. When a class is found, we declare a new sort with its name, that should be a subsort of Cid. Each one of its attributes are declared as constructors with result sort Attribute.

```
 mod SCHOOL-SYNTAX is
  pr STRING .
  pr CONFIGURATION .

  sort Person .
  subsort Person < Cid .
  op Person : -> Person .
```

The subclasses are defined in a similar way to the classes, with the exception that they are not directly a subsort of Cid, but a subsort of their superclass.

```
  sort Student .
  subsort Student < Person .
  op Student : -> Student .

  op score :_ : Nat -> Attribute [ctor] .

  sort State .
  ops before after : -> State .

  sort Teacher .
  subsort Teacher < Person .
  op Teacher : -> Teacher .

  sort OidSet .
  subsort Oid < OidSet .
  op mtOidSet : -> OidSet .
  op __ : OidSet OidSet -> OidSet [comm assoc id: mtOidSet] .

  op students :_ : OidSet -> Attribute [ctor] .
  op state :_ : State -> Attribute [ctor] .
```

The messages are defined as operators with result sort `Msg` and attributes `ctor` and `msg`.

```
  op to_from_exam : Oid Oid -> Msg [ctor msg] .
  op to_from_solution : Oid Oid -> Msg [ctor msg] .
  op to_score_ : Oid Nat -> Msg [ctor msg] .
 endm
```

Now, we show how to change the rules from the `SCHOOL` module:

```
 mod SCHOOL is
  pr SCHOOL-SYNTAX .
```

- We use new variables of the form `V@Class` of sort `Class` for each class, that will be used in the rules as class identifier in order to allow inheritance.
- We add to each object in a rule a different variable of sort `AttributeSet` to match with the attributes do not shown. For example, applying these two changes the rules correcting exams are:

```
  vars TCHR STDNT : Oid .
  var  OS : OidSet .
  vars N N' : Nat .
  var  V@Student : Student .
  var  V@Teacher : Teacher .
  var  AtS : AttributeSet .

  rl [correction] :
     to TCHR from STDNT solution
     < TCHR : V@Teacher | AtS >
  => < TCHR : V@Teacher | AtS >
     to STDNT score 0 .
```

```
rl [correction] :
   to TCHR from STDNT solution
   < TCHR : V@Teacher | AtS >
=> < TCHR : V@Teacher | AtS >
   to STDNT score 5 .

rl [correction] :
   to TCHR from STDNT solution
   < TCHR : V@Teacher | AtS >
=> < TCHR : V@Teacher | AtS >
   to STDNT score 10 .
```

- When an attribute that appears in the righthand side of a rule does not appear in the lefthand side it means that its value does not matter, and we use a new variable to match with all its possible values. For example, the rule where a student receives its score is now:

```
rl [score] :
   (to STDNT score N)
   < STDNT : V@Student | score : N', AtS >
=> < STDNT : V@Student | score : N, AtS > .
```

- When an attribute that appears in the lefthand side of a rule does not appear in the righthand side it means that it has not changed, so we copy it from the lefthand side. For example, the rule where the exams are sent is:

```
rl [exam] :
   < TCHR : V@Teacher | students : OS, state : before, AtS >
=> < TCHR : V@Teacher | students : OS, state : after, AtS >
   broadcastExam(OS, TCHR) .
```

The rest of the module looks as follows:

```
 op broadcastExam : OidSet Oid -> Configuration .
 eq broadcastExam(mtOidSet, TCHR) = none .
 eq broadcastExam(STDNT OS, TCHR) = to STDNT from TCHR exam
                                    broadcastExam(OS, TCHR) .

 rl [solve] :
    to STDNT from TCHR exam
    < STDNT : V@Student | AtS >
 => < STDNT : V@Student | AtS >
    to TCHR from STDNT solution .
endm
```

Chapter 3

Architectures

In this chapter we show how *distributed configurations*, made up of located configurations, can be built in Maude, in such a way that the architecture is transparent to the concrete application. Each located configuration is executed in a Maude process, and they are connected through sockets. In the following sections we present how to define three different architectures, namely, a star network, a ring network, and a centralized ring network, but we start with the part that is common to all of them.

3.1 Common infrastructure

In this section we describe the elements that are common to all the architectures we define below. They basically correspond to the way messages are redirected to reach their addressees. The different parts among the architectures correspond to the way the nodes are connected.

We assume that each located configuration contains one and only one *router*,[1] plus messages and possibly objects of other classes. The *names* of routers range over the sort `Loc` (subsort of `Oid`, the sort for objects identifiers declared in the predefined Maude module `CONFIGURATION`), and have the form `l(IP, N)` with the string `IP` the IP address of the machine in which the process is being executed and `N` a number used to distinguish locations in the same machine. We assume global uniqueness of router names in a distributed configuration.

```
fmod LOC is
 pr STRING .
 pr CONFIGURATION .
 sort Loc .
 subsort Loc < Oid .
 op l : String Nat -> Loc .                *** Router Oid
endfm
```

All objects can communicate with each other by using the message `to_:_`, that has as arguments the identifier of the addressee and a term of sort `TravelingContents`. We can communicate the name of a location when a socket is created by using the message `new-socket`.

[1]We identify the router and the location where it is.

```
fmod TRAVELING-CONTENTS is
 sort TravelingContents .
endfm

omod ARCHITECTURE-MSGS is
 pr LOC .
 pr TRAVELING-CONTENTS .

 msg to_:_ : Oid TravelingContents -> Msg .
 msg new-socket : Loc -> Msg .
endom
```

To be able to redirect the message to the appropriate location, the architecture must be able to obtain the location where the addressee resides. Since each application can define its own `Oid` syntax, we define a theory, that defines a function that extracts a `Loc` from the object identifier.

```
fth ARCH-COMPLEMENT is
 inc META-MODULE .
 inc LOC .
 op getLoc : Oid -> Loc .
```

Maude sockets can only transmit strings, so we must translate all the messages into strings and convert them back once they are received. To do it in a general way (independently of the concrete application) we use the reflective features of Maude. Concretely, we use a (metarepresented) module `MOD` with the definition of all the operators used to construct messages that are going to be transmitted, that is also included in the theory.

```
 op MOD : -> Module .
endfth
```

We suggest `o(L, N)`, with `L` the location where the object was created and `N` a number not used to name other objects, as the syntax for object identifiers, and we provide it and its corresponding `getLoc` function in the `OID` module.

```
fmod OID is
 pr LOC .
 op o : Loc Nat -> Oid .

 var L : Loc .
 var N : Nat .
 op getLoc : Oid -> Loc .
 eq getLoc(o(L, N)) = L .
endfm
```

We define views for `Loc` and `Oid`, that will be used by the architectures.

```
view Loc from TRIV to LOC is
 sort Elt to Loc .
endv

view Oid from TRIV to CONFIGURATION is
 sort Elt to Oid .
endv
```

The module MAYBE{X :: TRIV} adds a default value maybe to the sort used in the instantiation of the module.

```
fmod MAYBE{X :: TRIV} is
 sort Maybe{X} .
 subsort X$Elt < Maybe{X} .
 op maybe : -> Maybe{X} .
endfm
```

The COMMON-INFRASTRUCTURE module is parameterized by the ARCH-COMPLEMENT theory shown above.

```
omod COMMON-INFRASTRUCTURE{A :: ARCH-COMPLEMENT} is
 pr BUFFERED-SOCKET .
 pr ARCHITECTURE-MSGS .
 pr MAP{Loc, Oid} .
 pr MAYBE{Oid} * (op maybe to null) .
 pr META-LEVEL .
```

The Router class is defined as follows:

```
class Router | state : RouterState, port : Nat, neighbors : Map{Loc, Oid},
               defNeighbor : Maybe{Oid} .
```

This class will be specialized in the different architectures.

A router may be in states idle, waiting-connection, or active, although other values can be added in concrete architectures. The attribute state will take one of these values.

```
 sort RouterState .
 ops idle waiting-connection active : -> RouterState .
```

The attribute port keeps information about the port through which a server can offer its services or a client can ask for them.

To solve the *routing* problem we assume a very simple, although quite general, approach consisting in having a routing table in each router. Such a table gives the socket through which a message must be sent if one wants to reach a particular location. If there is a socket between the source and the target of the message then it reaches its destination in a single step; otherwise forwarding has to be repeated several times. The neighbors attribute maintains such a routing table as a map associating socket object identifiers to location identifiers. That is, the attribute neighbors stores in the partial function Map{Loc, Oid} information about the sockets through which data must be sent to reach a particular location. As we will see, each concrete architecture will use the new-socket message to update this attribute. The following rule describes how a message is redirected through the appropriate socket. If a message is sent to an object O and the message is in a location L, with L ≠ getLoc(O), that is connected to L (that is, LSPF[getLoc(O)] ≠ undefined), then the message is sent through the socket after converting it to a string with the function msg2string explained below.

```
 vars O O' SOCKET : Oid .
 vars L L' : Loc .
 vars DATA S S' S'' : String .
```

```
var  N : Nat .
var  MSG : Msg .
var  C : Contents .
var  LSPF : Map{Loc, Oid} .
var  Q : Qid .
var  QIL : QidList .

crl [redirect] :
    to O : TC
    < L : Router | neighbors : LSPF >
 => < L : Router | >
    Send(LSPF[getLoc(O)], L, msg2string(to O : TC))
 if getLoc(O) =/= L /\ LSPF[getLoc(O)] =/= undefined .
```

In case there is no socket associated to a particular location in the map `neighbors`, there can be a *default socket* stored in the attribute `defNeighbor`. Nevertheless, the value of the `defNeighbor` attribute may also be unspecified, that is, since `defNeighbor` is declared of sort `Maybe{Oid}`, it can take as value either an object identifier (representing a socket) or `null`. The rule `redirectDef` illustrates this behavior when there exists a default socket.

```
crl [redirectDef] :
    to O : TC
    < L : Router | neighbors : LSPF, defNeighbor : O' >
 => < L : Router | >
    Send(O', L, msg2string(to O : TC))
 if getLoc(O) =/= L /\ LSPF[getLoc(O)] == undefined .
```

Notice that `defNeighbor` cannot be `null` when this rule is applied because we use the variable `O`, of sort `Oid` (subsort of `Maybe{Oid}`). If `defNeighbor` should be used but it is `null`, then the data is not delivered.

When a router sees a `Received` message, it extracts the message from the string (by means of the function `string2msg`) and puts a new message in the configuration, and keeps listening with a new `Receive` message:

```
crl [Received] :
    Received(L, SOCKET, DATA)
    < L : Router | >
 => < L : Router | >
    MSG
    Receive(SOCKET, L)
 if MSG := string2msg(SOCKET, DATA) .
```

The `Sent` messages are just removed from the configuration:

```
eq Sent(O, O') = none .
```

Finally, we show how the `MOD` module from the theory `ARCH-COMPLEMENT` is used. This module must contain the definition (the operator declarations) of all the possible values that the message can take. The function `msg2string` uses the functions `upTerm` and `metaPrettyPrint` from module `META-LEVEL` to generate a `QidList` from the message. Then, the function `qidList2String` is used to generate a string from the `QidList`.

```
  op msg2string : Msg -> String .
  eq msg2string(MSG) = qidList2String(metaPrettyPrint(MOD, upTerm(MSG), none)) .

  op qidList2String : QidList -> String .
  op qidList2String* : QidList String -> String .
  eq qidList2String(QIL) = qidList2String*(QIL, "") .
  eq qidList2String*(nil, S) = S .
  eq qidList2String*(Q QIL, S) = qidList2String*(QIL, S + string(Q) + " ") .
```

The function `string2msg` uses a similar strategy. It uses `string2QidList` to generate a `QidList` from a string. Then, the function `metaParse` is used, that needs the same module than `metaPrettyPrint` as first parameter, to generate the message. We handle errors by putting an `error` message in the configuration.

```
  op string2msg : Oid String -> Msg .
  ceq string2msg(O, S)
    = if new-socket?(MSG) then new-socket(getLoc(MSG), O) else MSG fi
  if MSG :=
    downTerm(getTerm(metaParse(MOD, string2QidList(S), 'Msg)), error(S)) .

  op error : String -> Msg [ctor] .

  op string2QidList : String -> QidList .
  op string2QidList* : String QidList -> QidList .

  eq string2QidList(S) = string2QidList*(S, nil) .
  eq string2QidList*("", QIL) = QIL .
  ceq string2QidList*(S, QIL)
    = string2QidList*(S'', QIL qid(S') )
    if N := find(S, " ", 0)
       /\ S' := substr(S, 0, N)
       /\ S'' := substr(S, N + 1, length(S)) .
  eq string2QidList*(S, QIL) = QIL qid(S) [owise] .

  op new-socket : Loc Oid -> Msg .

  op getLoc : Msg ~> Loc .
  eq getLoc(new-socket(L)) = L .

  op new-socket? : Msg -> Bool .
  eq new-socket?(new-socket(L)) = true .
  eq new-socket?(MSG) = false [owise] .
 endom
```

Notice that `string2msg` receives the socket through where the message have arrived. It is used to put in the configuration the `new-socket` messages received, because the location that sends the message only knows the socket name *in its side*, so the name of the socket when the message arrives to the addressee is obtained from the `Received` message, putting into the configuration a slightly different `new-socket` message with the socket identifier in addition to the location name.

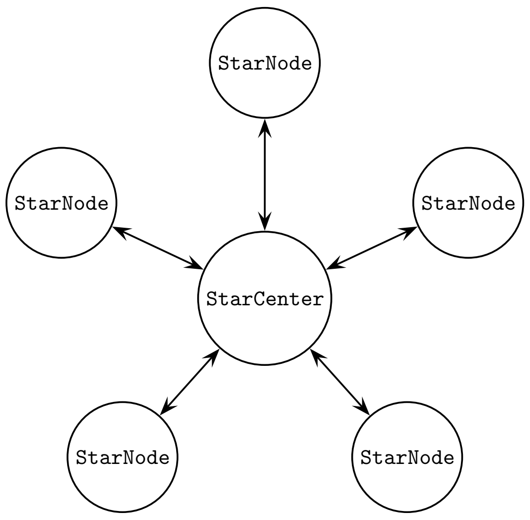

Figure 3.1: Star architecture

3.2 Star architecture

The architecture we present here consists of a location with a *server* router, and several locations with *client* routers. The server is connected to all clients, and each client is connected only to the server. That is, we have a *star network*, with the *center* redirecting the messages between the *nodes*, as illustrated in Figure 3.1.

We distinguish between the center and the nodes by declaring two subclasses of `Router`: `StarCenter` with no additional attributes; and `StarNode`, with an attribute `center`, that keeps the center IP address. These classes must define how the locations are connected by filling the `neighbors` and `defNeighbor` attributes.

```
omod STAR-CENTER{A :: ARCH-COMPLEMENT} is
 pr COMMON-INFRASTRUCTURE{A} .

 class StarCenter | .
 subclass StarCenter < Router .
```

The center plays the server role from the point of view of the sockets so it declares itself as a TCP server socket, and offers its services on its `port`.

```
 vars SOCKET NEW-SOCKET SOCKET-MANAGER : Oid .
 vars L L' : Loc .
 vars DATA IP : String .
 var  N : Nat .
 var  LSPF : Map{Loc, Oid} .

 rl [connect] :
    < L : StarCenter | state : idle, port : N >
 => < L : StarCenter | state : waiting-connection >
    CreateServerTcpSocket(socketManager, L, N, 5) .
```

Note that it goes from state `idle` to `waiting-connection`, so this rule is applied only once. The response is handled by the rule `connected` below. Once it receives the

CreatedSocket message, it becomes active and sends a message indicating that it is ready to accept clients through the server socket.

```
  rl [connected] :
     CreatedSocket(L, SOCKET-MANAGER, SOCKET)
     < L : StarCenter | state : waiting-connection >
  => < L : StarCenter | state : active >
     AcceptClient(SOCKET, L) .
```

In the rule acceptedClient below, in addition to sending messages AcceptClient and Receive indicating, respectively, that it is ready to accept new nodes as clients through the server socket, and messages through the new socket, the object that gets the AcceptedClient message sends to the node the message new-socket communicating its identifier. These new-socket messages are interchanged between the center and the nodes in both directions so they can know their Maude identifiers besides the socket that connects them.

```
  rl [acceptedClient] :
     AcceptedClient(L, SOCKET, IP, NEW-SOCKET)
     < L : StarCenter | state : active >
  => < L : StarCenter | >
     AcceptClient(SOCKET, L)
     Receive(NEW-SOCKET, L)
     Send(NEW-SOCKET, L, msg2string(new-socket(L))) .
```

When a new-socket message is received from a node with its name L', it is stored in the neighbors attribute.

```
  rl [new-socket] :
     new-socket(L', SOCKET)
     < L : StarCenter | state : active, neighbors : LSPF >
  => < L : StarCenter | neighbors : insert(L', SOCKET, LSPF) > .
 endom
```

When a StarNode is created, it first tries to establish a connection with the center by sending a CreateClientTcpSocket message that uses the IP address and the port of the center.

```
 omod STAR-NODE{A :: ARCH-COMPLEMENT} is
  pr COMMON-INFRASTRUCTURE{A} .

  class StarNode | center : String .
  subclass StarNode < Router .

  vars SOCKET SOCKET-MANAGER : Oid .
  vars L L' : Loc .
  vars DATA IP : String .
  var  N : Nat .

  rl [connect] :
     < L : StarNode | state : idle, center : IP, port : N >
  => < L : StarNode | state : waiting-connection >
     CreateClientTcpSocket(socketManager, L, IP, N) .
```

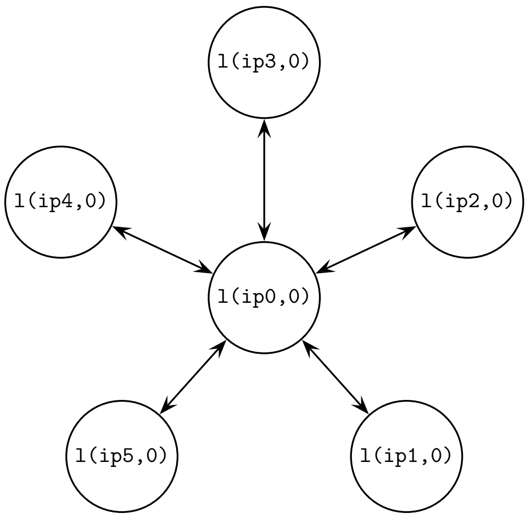

Figure 3.2: A concrete star architecture

Nodes go to the `waiting-connection` state as a result of the application of this rule. The response to a star node's socket connection request is handled by the following rule `connected`, where the node also sends the `new-socket` message right after the socket is created. In this very simple architecture the node does not need to wait for the `new-socket` message from the center, because it only updates the `defNeighbor` attribute, that only needs the socket through which messages have to be sent, so when the `new-socket` message is found, it is removed from the configuration. Nodes start listening with the `Receive` message.

```
  rl [connected] :
     CreatedSocket(L, SOCKET-MANAGER, SOCKET)
     < L : StarNode | state : waiting-connection, defNeighbor : null >
  => < L : StarNode | state : active, defNeighbor : SOCKET >
     Send(SOCKET, L, msg2string(new-socket(L)))
     Receive(SOCKET, L) .

  rl [new-socket] :
     new-socket(L', SOCKET)
     < L : StarNode | >
  => < L : StarNode | > .
 endom
```

We show how this architecture works by means of an example. The initial configuration for the node `l(ip0, 0)` in Figure 3.2 (that is, the star center) is:

```
  < l(ip0, 0) : StarCenter |
                state : idle,
                neighbors : empty,
                defNeighbor : null,
                port : 60039 >
```

The initial configuration for the node `l(ip1, 0)` in the same figure (the unique difference between the star nodes is their names) is:

```
< l(ip1, 0) : StarNode |
          state : idle,
          neighbors : empty,
          defNeighbor : null,
          center : ip0,
          port : 60039 >
```

Once all the connections has been established, the node `l(ip0, 0)` keeps in its `neighbors` attribute the buffered sockets (see Section 2.8) it must use to reach all the star nodes. The final state reached by the center is:

```
< l("127.0.0.1", 0) : StarCenter |
      state : active,
      neighbors : (l(ip1, 0) |-> b(socket(6)), l(ip2, 0) |-> b(socket(7)),
                   l(ip3, 0) |-> b(socket(10)), l(ip4, 0) |-> b(socket(8)),
                   l(ip5, 0) |-> b(socket(9))),
      defNeighbor : null,
      port : 60039 >
```

The star nodes, such as the `l(ip1, 0)` node, only have updated their `defNeighbor` attribute.

```
< l(ip1, 0) : StarNode |
      state : active,
      neighbors : empty,
      defNeighbor : b(socket(5)),
      port : 60039,
      center : ip0 >
```

Notice that the same socket have different names in each side. `l(ip0, 0)` sends messages to `l(ip1, 0)` through `b(socket(6))`, but `l(ip1, 0)` receives the messages through `b(socket(5))`.

3.3 Ring architecture

In a ring topology, each node is connected to two nodes, the previous and the next one. We show here how to implement a unidirectional ring where each node receives data from the previous one and sends data to the next one.

In this architecture, each node must be declared as a (Maude) server for the previous one and as a (Maude) client of the next one. However, to declare a node as a client it needs another one working as a server, what it is impossible for the first executed Maude instance. We have decided to distinguish between the *last* Maude instance executed (which knows that all other instances are already running) and the other ones by declaring two subclasses of `Router`:

- `RingNode` defines the behavior of all the nodes but the last one.[2] They first declare themselves as servers and then wait until someone ask to be their client. Once they have accepted a client, they try to be clients themselves of the next node in the ring.

[2]Although in a ring there is no "last" node, we refer to the order in which the nodes must be started to be executed.

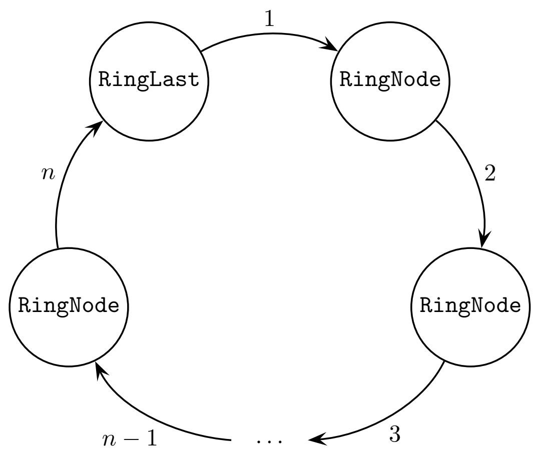

Figure 3.3: Ring architecture

- `RingLast` defines the behavior of the last node, that asks the next one (that must exist, because this node is the last one) to be its server, and then waits to accept clients.

The order in which the connections are established is illustrated in Figure 3.3.

In addition to the states inherited from `Router`, we define the `connecting2next` and `waiting4previous` states, that will be traversed respectively when a node tries to be client of the next node and when a node waits to accept the previous as a client. Both `RingNode` and `RingLast` will reach the same states although in different order (thus they need the same attributes), and will declare themselves as servers at start-up, so we define first a superclass `RingRouter` containing the common behavior. It is a subclass of `Router` with attributes `nextIP` and `nextPort` that keep, respectively, the IP address and the port of the next node in the ring.

```
omod COMMON-RING{A :: ARCH-COMPLEMENT} is
 pr COMMON-INFRASTRUCTURE{A} .

 ops connecting2next waiting4previous : -> RouterState .

 class RingRouter | nextIP : String, nextPort : Nat .
 subclass RingRouter < Router .
```

The `port` attribute inherited from class `Router` is the port used by the ring objects to declare themselves as servers and accept clients through it.

```
 var L : Loc .
 var N : Nat .

 rl [connect] :
    < L : RingRouter | state : idle, port : N >
 => < L : RingRouter | state : waiting-connection >
    CreateServerTcpSocket(socketManager, L, N, 5) .
endom
```

As we have said, all the nodes but the last one wait for clients after declaring themselves as servers (by using the rule `connect` above), reaching the state `waiting4previous`.

```
omod RING-NODE{A :: ARCH-COMPLEMENT} is
 pr COMMON-RING{A} .

 class RingNode | .
 subclass RingNode < RingRouter .

 vars SOCKET NEW-SOCKET SOCKET-MANAGER : Oid .
 var  L : Loc .
 vars IP IP' : String .
 var  N : Nat .

 rl [connected] :
    CreatedSocket(L, SOCKET-MANAGER, SOCKET)
    < L : RingNode | state : waiting-connection >
 => < L : RingNode | state : waiting4previous >
    AcceptClient(SOCKET, L) .
```

Once a client is accepted, the server tries to be client of the next node in the ring, reaching the state `connecting2next`. Note that there is not a new `AcceptClient` message on the righthand side of the rule, because each server has only *one* client.

```
 rl [acceptedClient] :
    AcceptedClient(L, SOCKET, IP', NEW-SOCKET)
    < L : RingNode | state : waiting4previous, nextIP : IP, nextPort : N >
 => < L : RingNode | state : connecting2next >
    Receive(NEW-SOCKET, L)
    CreateClientTcpSocket(socketManager, L, IP, N) .
```

When a node is accepted as client by the next node, the former keeps the socket in the attribute `defNeighbor`, in order to use it to redirect all the messages, and reaches the `active` state. Notice that the `neighbors` attribute remains `empty` in this architecture and that no `Receive` message has been put in the configuration, because a client does not receive data from the server in this architecture.

```
 rl [connected2next] :
    CreatedSocket(L, SOCKET-MANAGER, SOCKET)
    < L : RingNode | state : connecting2next, defNeighbor : null >
 => < L : RingNode | state : active, defNeighbor : SOCKET > .
endom
```

The last node traverses the states in different order. When it is accepted as a server, it tries to connect to the next node in the ring, reaching the `connecting2next` state.

```
omod RING-LAST{A :: ARCH-COMPLEMENT} is
 pr COMMON-RING{A} .

 class RingLast | .
 subclass RingLast < RingRouter .

 vars SOCKET NEW-SOCKET SOCKET-MANAGER : Oid .
 var  L : Loc .
```

```
  var  IP : String .
  var  N : Nat .

  rl [connected] :
     CreatedSocket(L, SOCKET-MANAGER, SOCKET)
     < L : RingLast | state : waiting-connection, nextIP : IP, nextPort : N >
  => < L : RingLast | state : connecting2next >
     AcceptClient(SOCKET, L)
     CreateClientTcpSocket(socketManager, L, IP, N) .
```

Once it is accepted as client, it saves the socket identifier in `defNeighbor` in order to redirect all the messages using it, and reaches the `waiting4previous` state. Again, no `Receive` message is needed, because the server does not send data to the clients.

```
  rl [connected2next] :
     CreatedSocket(L, SOCKET-MANAGER, SOCKET)
     < L : RingLast | state : connecting2next, defNeighbor : null >
  => < L : RingLast | state : waiting4previous, defNeighbor : SOCKET > .
```

Finally, when it accepts a client it starts to receive data through the socket and becomes `active`.

```
  rl [acceptedClient] :
     AcceptedClient(L, SOCKET, IP, NEW-SOCKET)
     < L : RingLast | state : waiting4previous >
  => < L : RingLast | state : active >
     Receive(NEW-SOCKET, L) .
 endom
```

Notice that in this architecture the `neighbors` attribute is not used; the ring nodes are just connected by `defNeighbor`, thus obtaining a unidirectional ring.

3.4 Centralized ring architecture

We show here a special ring architecture, where in addition to the ring we have a center connected to each node, as illustrated in Figure 3.4. We have a mixture of the two previous architectures, that we have tried to reuse as much as possible. We use the class `StarCenter` (from the star architecture presented in Section 3.2) for the ring center, and we reuse the classes `RingNode` and `RingLast` (from the ring architecture shown in Section 3.3) for the nodes in the ring, although some states must be renamed.

We define a new class `CRingRouter` in charge of connecting to the center. We will combine the behavior of this new class with the classes `RingNode` and `RingLast` to obtain the centralized ring. This new class has:

- New attributes `centerIP` and `centerPort`, with the IP address and port of the central server.

- New states `connecting2center` and `waiting4center`.

- Rules for connecting to the central node.

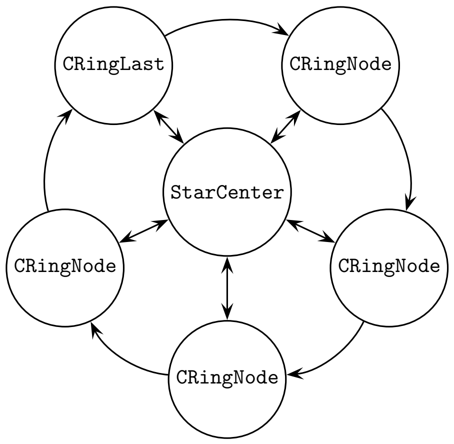

Figure 3.4: Centralized ring architecture

```
omod CENTRALIZED-RING{A :: ARCH-COMPLEMENT} is
 pr COMMON-RING{A} .

 class CRingRouter | centerIP : String, centerPort : Nat .
 subclass CRingRouter < RingRouter .

 ops connecting2center waiting4center : -> RouterState .
```

When it is in `connecting2center` state, it tries to connect to the center and reaches the `waiting4center` state.

```
 vars SOCKET SOCKET-MANAGER : Oid .
 vars L L' : Loc .
 vars DATA IP : String .
 var  N : Nat .
 var  LSPF : Map{Loc,Oid} .

 rl [connect2center] :
    < L : CRingRouter | state : connecting2center, centerIP : S,
                        centerPort : N >
 => < L : CRingRouter | state : waiting4center >
    CreateClientTcpSocket(socketManager, L, S, N) .
```

Once the connection has been created, the server and the client interchange `new-socket` messages, and the `neighbors` attribute is updated, reaching the `active` state.

```
 rl [connected] :
    CreatedSocket(O, SOCKET-MANAGER, SOCKET)
    < L : CRingRouter | state : waiting4center >
 => < L : CRingRouter | >
    Receive(SOCKET, L)
    Send(SOCKET, L, msg2string(new-socket(L))) .

 rl [connected2center] :
    new-socket(L', SOCKET)
    < L : CRingRouter | state : waiting4center, neighbors : LSPF >
```

```
  => < L : CRingRouter | state : active,
                         neighbors : insert(L', SOCKET, LSPF) > .
endom
```

Note that we update the `neighbors` attribute, so the messages to the center will use `SOCKET`, while all other messages will use `defNeighbor` from the ring architecture.

Now we look for a class that behaves as a `CRingRouter` and as a `RingNode` (or as a `RingLast`, if it is the last node). To obtain it, we define a new class `CRingNode`, which is a subclass of both `CRingRouter` and `RingNode` (and a new class `CRingLast`, which is a subclass of `CRingRouter` and `RingLast`). These new classes behave as the corresponding nodes in the ring, and once they are connected behave as clients of the center. However, we found the problem that all those classes finish in `active` state, so some of the rules could not be applied. We solve it by renaming the state `active` in the nodes of the ring to `connecting2center`, so the rules in `CRingRouter` can be applied *after* the ring connections has been established.

```
omod CENTRALIZED-RING-NODE{A :: ARCH-COMPLEMENT} is
 pr CENTRALIZED-RING{A} .
 pr RING-NODE{A} * (op active to connecting2center) .

 class CRingNode | .
 subclass CRingNode < CRingRouter RingNode .
endom

omod CENTRALIZED-RING-LAST{A :: ARCH-COMPLEMENT} is
 pr CENTRALIZED-RING{A} .
 pr RING-LAST{A} * (op active to connecting2center) .

 class CRingLast | .
 subclass CRingLast < CRingRouter RingLast .
endom
```

In the following section, as well as in Chapters 4 and 5, we will illustrate how these architectures can be used to execute concrete applications on top of them.

3.5 Ray tracing case study

Once we have several locations connected by means of an architecture like those shown above, we can implement distributed applications. We first illustrate in this section how a concrete distributed application can be implemented directly in Maude.

In order to implement a distributed application, the messages that objects in different locations will interchange should be declared in a separated module, that then will be combined with the messages of the architecture and used to instantiate the module `COMMON-INFRASTRUCTURE`. Then the objects that solve the application have to be implemented, in a way as independent of the concrete architecture as possible. Finally, in order to execute the application, a concrete architecture has to be chosen and the distribution of the application objects through the different locations has to be decided. We use the ray tracing problem as a case study [8].

3.5.1 Sequential implementation

Given a scene consisting of 3D objects, and given the position of the camera, a ray tracer calculates a 2D image of the scene. For every pixel of the output image, the ray tracer shoots a ray into the scene and tests whether it impacts with any object of the scene. When an impact is found, the ray is reflected and the color of the intersection point is computed based on the strength of the ray and on the texture of the object's material.

We show here the *sequential* implementation of the problem. We will reuse most of this code in the distributed versions in Sections 3.5.2 and 4.3. We will show in Section 6.6 how to check distributed applications by considering the sequential versions as the *specification* of the problem and the distributed one as the *implementation*.

Let us see the modules in detail. The `FIGURE` module includes the modules `3D`, that defines operations over 3D elements like points and vectors, and `COLOR`, that defines the colors for the figures. In the module we define figures by their type, color and coordinates, and we declare the functions `filter`, that checks whether the figures on the list are too far or not; `getColor`, that extracts the figure color; and `distance`, that will be used later to calculate the intersection of the rays with the figures. Although only spheres are considered in this example, other types of figures can be easily added just defining their `Coordinates` and `FigureType` and the equations dealing with distance.

```
fmod FIGURE is
 pr 3D .
 pr COLOR .

 var  FT : FigureType .
 var  C : Color .
 var  CDT : Coordinates .
 vars F F' F'' F''' x x' x'' y y' y''
      z z' z'' RAD bSq r r2 alpha a2 : Float .
 vars u v v' : Vector .
 var  FIG : Figure .
 var  FL : FigureList .
 vars P P1 P2 P3 q : Point .

 sort Figure FigureList FigureType Coordinates .
 subsort Figure < FigureList .

 op figure : FigureType Color Coordinates -> Figure .

 op mtFigureList : -> FigureList .
 op __ : FigureList FigureList -> FigureList [assoc id: mtFigureList] .

 op sphere : -> FigureType .
 op coord : Point Float -> Coordinates .

 op getColor : Figure -> Color .
 eq getColor(figure(FT, C, CDT)) = C .

 op filter : FigureList Float -> FigureList .
 eq filter(mtFigureList, F) = mtFigureList .
 eq filter(FIG FL, F) = if farAway(FIG, F) then mtFigureList
                                            else FIG fi filter(FL, F) .
```

```
  op farAway : Figure Float -> Bool .
  eq farAway(figure(sphere, C, coord(< F, F', F'' >, RAD)), F''') = F''' < F'' .

  op distance : Point Point Figure -> Distance .
  ceq distance(P1, P2, FIG) = module(P1 - P3)
   if P3 := distanceAux(P1, P2, FIG) /\
      P3 =/= noIntersection .

  eq distance(P1, P2, FIG) = noDistance [owise] .

  op distanceAux : Point Point Figure -> Point .
  ceq distanceAux(< x, y, z >, < x', y', z' >,
                figure(sphere, C, coord(< x'', y'', z'' >, r))) =
     if (bSq > r2) then noIntersection
     else if (alpha >= sqrt(a2)) then (q - (sqrt(a2) * u))
          else if ((alpha + sqrt(a2)) > 0.0) then q + (sqrt(a2) * u)
               else noIntersection fi
          fi
     fi
   if u := unitVector(< x, y, z >, < x', y', z' >) /\
      v := < x'', y'', z'' > - < x, y, z > /\
      alpha := escProd(u, v) /\
      q := < x, y, z > + (alpha * u)  /\
      v' := q - < x'', y'', z'' >  /\
      F := module(v')  /\
      bSq := F * F  /\
      r2 := r * r  /\
      a2 := r2 - bSq .
endfm
```

The ROWTRACER module is in charge of coloring each row. The traceRow function traverses the row from left to right, and for every pixel in the current line it calculates all the collisions with the figures on the list and keeps the nearest one by using getColor (where d, a constant of sort Color, is the default color).

```
fmod ROWTRACER is
  pr FIGURE .
  pr CONVERSION .

  vars P P1 P2 : Point .
  var  FL : FigureList .
  var  F : Figure .
  var  C : Color .
  var  D : Distance .

  sort ColorList .
  subsort Color < ColorList .
  --- Color union
  op __ : ColorList ColorList -> ColorList [assoc] .

  op traceRow : Point Float Float Float FigureList -> ColorList .
  op getColor : Point FigureList -> Color .

  eq getColor(P, FL) = getColor(< 0.0, 0.0, 0.0 >, P, FL) .
  op getColor : Point Point FigureList -> Color .
  op getColorAux : Point Point FigureList Color Distance -> Color .

  eq getColor(P1, P2, F FL) = getColorAux(P1, P2, F FL, d, noDistance) .
  eq getColor(P1, P2, mtFigureList) = d .

  eq getColorAux(P1, P2, mtFigureList, C, D) = C .
  eq getColorAux(P1, P2, F FL, C, D) =
   if less(distance(P1, P2, F), D) then
    getColorAux(P1, P2, FL, getColor(FIG), distance(P1, P2, F))
   else
    getColorAux(P1, P2, FL, C, D)
   fi .

  op traceRow : Point Float Float Float Float FigureList -> ColorList .
  eq traceRow(P, Xr, Xr, Y, Near, FL) = getColor(P, < Xr, Y, Near >, FL) .
  ceq traceRow(P, Xl, Xr, Y, Near, FL) =
      getColor(P, < Xl, Y, Near >, FL) traceRow(P, Xl + 1.0, Xr, Y, Near, FL)
   if Xl < Xr .
 endfm
```

Finally, the `RAYTRACING` module below traverses all the rows with the function `rayTracing` and colors each one with `traceRow`.

```
 fmod RAYTRACING is
  pr ROWTRACER .

  sort Screen .
  op [_] : ColorList -> Screen .
  --- Screen union
  op __ : Screen Screen -> Screen [assoc] .

  op rayTracing : Point Float Float  Float Float   Float FigureList -> Screen .
              ---       Xleft Xright Ytop  Ybottom Near  Figures

  vars Xlef Xrig Ytop Ybot Near X Y : Float .
  var  FL : FigureList .
  var  P : Point .

  eq rayTracing(P, Xlef, Xrig, Y, Y, Near, FL) =
      [traceRow(P, Xlef, Xrig, Y, Near, FL)] .
  ceq rayTracing(P, Xlef, Xrig, Ytop, Ybot, Near, FL) =
      [traceRow(P, Xlef, Xrig, Ytop, Near, FL)]
      rayTracing(P, Xlef, Xrig, Ytop - 1.0, Ymin, Near, FL)
   if Ytop > Ybot .
 endfm
```

3.5.2 Distributed implementation

This application is highly parallelizable: each row (indeed, each pixel) can be colored by a different processor in an independent way, and then they can be combined to obtain the whole screen. So we will have a chief that delivers subproblems and combines subresults, and several painters that solve the subproblems, that is, color rows of the screen.

The communication between the chief and the painters is through the following messages, that must have sort `TravelingContents` to fit into the `to_:_` message. We use the module `OID` from Section 3.1 for the objects syntax.

```
fmod RT-TRANSMITTED-SYNTAX is
 pr OID .
 pr ROWTRACER .
 pr TRAVELING-CONTENTS .
```

- `world`, that sends the data describing the problem, that is, the list of figures in the scene, the position of the camera, and the size of the screen.

```
      op world : FigureList Point Float Float Float -> TravelingContents .
```

- `new-row`, that communicates a new task by identifying the height of the row to be colored.

```
      op new-row : Float -> TravelingContents .
```

- `colored-row`, that transmits a new result to the chief.

```
      op colored-row : Oid Float ColorList -> TravelingContents .
     endfm
```

The application only needs to import the `ARCHITECTURE-MSGS` module from the architecture, so it can be executed on different architectures; each utilization of the application must include the architecture it will use (we will see an example below). The module `ROWTRACER` from Section 3.5.1 is also used; it contains the ingredients of this problem, in particular the function `traceRow` used by the painters.

```
view ColorList from TRIV to ROWTRACER is
 sort Elt to ColorList .
endv

omod DISTRIBUTED-RAY-TRACING is
 pr ARCHITECTURE-MSGS .
 pr RT-TRANSMITTED-SYNTAX .
 pr ROWTRACER .
 pr MAP{Float, ColorList} .
 pr LIST{Oid} * (sort List{Oid} to OidList, op nil to mtOidList) .
 pr LIST{Float} * (sort List{Float} to FloatList, op nil to mtFloatList) .
```

We define now a class `RTChief` in charge of distributing subproblems (rows to be colored) and combining the results (colored rows). This new class has attributes that

- Describe the problem:
  - The list of `figures`.
  - The width of the screen (`xL` and `xR`).
  - The height of the screen (`yT` and `yB`).
  - The depth where figures can be traced (`zN` and `zF`).

- Keep the current row (y).

- Keep the `result`; the subresults may arrive unordered, so we use a partial function from rows, identified by floats, to colored rows to represent the (partial) result.

- Store the list with the identifiers of the `painters`.

```
class RTChief | figures : FigureList, xL, xR, yT, yB, zN, zF: Float,
                y: Float, result : Map{Float, ColorList},
                painters : OidList .
```

First, the chief must deliver the information of the world and the first subproblems to the painters. Initially they receive three tasks[3] so they can work in the following one while a new one arrives.

```
var  CM : Map{Float, ColorList} .
var  OL : OidList .
vars XL XR YT YB ZN ZF R Y : Float .
var  CL : ColorList .
var  FigL : FigureList .
var  FL : FloatList .
var  P : Point .
vars O O' : Oid .

rl [new-painter] :
   < O : RTChief | xL : XL, xR : XR, y : Y, yT : YT, yB : YB, zN : ZN,
                   zF : ZF, figures : FigL, painters : O' OL >
=> < O : RTChief | y : Y - 3.0, painters : OL >
   to O' : world(filter(FigL, ZF),
                 < (XR + XL) / 2.0, (YT + YB) / 2.0, 0.0 >, XL, XR, ZN)
   to O' : new-row(Y)
   to O' : new-row(Y - 1.0)
   to O' : new-row(Y - 2.0) .
```

When one result arrives, it is combined with the current partial result (by using the `insert` operation from partial functions) and it is checked if the problem has been fully distributed (in this case `Y < YB`); if this is not the case, the next row is sent to the painter.

```
crl [new-row] :
    to O : colored-row(O', R, CL)
    < O : RTChief | result : CM, y : Y, yB : YB >
 => < O : RTChief | result : insert(R, CL, CM), y : Y - 1.0 >
    to O' : new-row(Y)
 if Y >= YB .

crl [no-more-rows] :
    to O : colored-row(O', R, CL)
    < O : RTChief | result : CM, y : Y, yB : YB >
 => < O : RTChief | result : insert(R, CL, CM) >
 if Y < YB .
```

[3]We are assuming that the number of works to be dispatched is at least three times the number of painters. This will be generalized in the following sections.

Now we define the class `RTPainter`, that will define the painters' behavior. Its attributes will keep the information of the world (the screen – `xL`, `xR`, and `zN` –, position of the camera, and the list of `figures`), the row identifiers of the undone tasks (`nextRows`), and the identifier of the `chief`.

```
class RTPainter | xL, xR, zN : Float, pos : Point, figures : FigureList,
                  nextRows : FloatList, chief : Oid .
```

When the world or a new row arrives, the information is stored in the appropriate attributes:

```
rl [rec-world] :
   to O : world(FigL, P, XL, XR, ZN)
   < O : RTPainter | >
=> < O : RTPainter | figures : FigL, pos : P, xL : XL, xR : XR, zN : ZN > .

rl [new-row] :
   to O : new-row(R)
   < O : RTPainter | nextRows : FL >
=> < O : RTPainter | nextRows : FL R > .
```

While the list of scheduled tasks is not empty, we can do a new one (trace the row) and send it to the chief through the message `colored-row`. Notice that here we use the function `traceRow` from the sequential version (see Section 3.5.1):

```
crl [paint] :
    < O : RTPainter | figures : FigL, pos : P, xL : XL, xR : XR, zN : ZN,
                      chief : O', nextRows : R FL >
 => < O : RTPainter | nextRows : FL >
    to O' : colored-row(O, R, CL)
 if CL := traceRow(P, XL, XR, R, ZN, FigL) .
endom
```

To execute an example of this application, we must define a view from the theory `ARCH-COMPLEMENT` used by the architecture:

```
mod RT-COMPLEMENT is
 pr META-LEVEL .
 pr RT-TRANSMITTED-SYNTAX .
 pr ARCHITECTURE-MSGS .
endm

view RT-Complement from ARCH-COMPLEMENT to RT-COMPLEMENT is
 op MOD to term upModule('RT-COMPLEMENT, false) .
endv
```

Notice that the `getLoc` function has been defined in `OID`, that is included by the module `RT-TRANSMITTED-SYNTAX`.

We can now execute an example of distributed ray tracing. To do it we must first choose the architecture we are going to use and instantiate it with the view `RT-Complement`. In this case the most suitable one is the star topology, placing the chief in the center of the star and the painters in the nodes. We define modules `EXAMPLE-CHIEF` and `EXAMPLE-PAINTER` where the initial configurations will be executed. `EXAMPLE-CHIEF` also includes a generator of random spheres that uses the `RANDOM` module provided by Maude.

```
mod EXAMPLE-CHIEF is
 pr DISTRIBUTED-RAY-TRACING .
 pr RANDOM .
 pr STAR-CENTER{RT-Complement} .

 var L : FigureList .
 var N : Nat .

 op figListN : Nat -> FigureList .
 op figListN* : Nat FigureList -> FigureList .

 eq figListN(N) = figListN*(N, mtFigureList) .
 eq figListN*(0, L) = L .
 eq figListN*(s(N), L) = figListN*(N, L figure(sphere, r,
        coord(< float(random(4 * N)), float(random(4 * N + 1)),
                float(random(4 * N + 2)) >, float(random(4 * N + 3))))) .
endm
```

The initial configuration for the center of the star includes a `StarCenter` and a `RTChief` with the whole definition of the problem.

```
 erew <> < l(ip0, 0) : StarCenter |
                state : idle,
                neighbors : empty,
                defNeighbor : null,
                port : 60039 >
        < o(l(ip0, 0), 0) : RTChief |
                xL : -30.0,
                xR : 30.0,
                y : 30.0,
                yT : 30.0,
                yB : -30.0,
                zN : 10.0,
                zF : 1000000000.0,
                figures : figListN(10),
                result : empty,
                painters : o(l(ip1, 0), 0) o(l(ip2, 0), 0) > .
```

where the `ipi` are IP addresses.

All the other locations have in their initial configuration a `StarNode` and a `RTPainter`, being their unique difference their identifiers. The configuration for one of them is shown below.

```
mod EXAMPLE-PAINTER is
 pr DISTRIBUTED-RAY-TRACING .
 pr STAR-NODE{RT-Complement} .
endm

erew <> < l(ip1, 0) : StarNode |
           state : idle,
           neighbors : empty,
           defNeighbor : null,
           center : ip0,
           port : 60039 >
        < o(l(ip1, 0), 0) : RTPainter |
           chief : o(l(ip0, 0), 0),
           nextRows : mtFloatList,
           figures : mtFigureList,
           pos : < 0.0, 0.0, 0.0 >,
           xL : 0.0,
           xR : 0.0,
           zN : 0.0 > .
```

Chapter 4

Parameterized skeletons

We show in this chapter how distributed applications can be implemented in Maude by means of object-oriented parameterized skeletons, that receive the operations needed to solve a concrete problem as a parameter. These operations usually are part of the sequential version of the concrete applications, thus encouraging code reusability. The use of Maude allows us to have the description of the architecture, the definition of the skeleton, and the implementation of the application in the same high-level language. Moreover, since Maude has a well-defined semantics, we obtain a good basis for formal reasoning. Tools for doing some kinds of this reasoning in an automatic way and the possibility to define the properties the applications have to fulfill are also provided by Maude, as we will see in Section 6.6.

The work described in this chapter has been published in the technical report [60], and it is going to be presented in the *9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS 07*, which originated the publication [59].

4.1 Distributable applications

In this section we present some of the applications we have used to test the implemented skeletons by giving a sequential implementation of them. They are typical well-known case studies used to present parallel distributed computing. Most of this code will be reused in the corresponding distributed application and we will use it in Section 6.6 to formally analyze the distributed one.

4.1.1 Euler numbers

The Euler number of a given value x, denoted by $\varphi(x)$, is the number of natural numbers smaller than x that are relatively prime to x. We are interested in computing the sum of the Euler numbers of the first n numbers, that is $\sum_{i=1}^{n} \varphi(i)$. The function `euler` below computes the Euler number of one number.

```
fmod EULER is
 pr NAT .

 op relPrimes : Nat Nat -> Bool .
 op euler : Nat -> Nat .
 op euler* : Nat Nat Nat -> Nat .
 vars N N' Ac : Nat .

 eq relPrimes(N, N') = gcd(N, N') == 1 .
 eq euler(N) = euler*(N, 1, 0) .
 ceq euler*(N, N', Ac) =
               if relPrimes(N, N') then euler*(N, N' + 1, Ac + 1)
                                   else euler*(N, N' + 1, Ac) fi
  if N' < N .
 eq euler*(N, N', Ac) = Ac [owise] .
endfm
```

The function `sumEuler` below computes the total sum by using successive calls to the `euler` function.

```
fmod SUM-EULER is
 pr EULER .

 vars N Ac : Nat .

 op sumEuler : Nat -> Nat .
 op sumEuler* : Nat Nat -> Nat .

 eq sumEuler(N) = sumEuler*(N, 0) .
 eq sumEuler*(0, Ac) = Ac .
 eq sumEuler*(s(N), Ac) = sumEuler*(N, Ac + euler(s(N))) .
endfm
```

Notice that each $\varphi(i)$ (computed by the function `euler`) can be calculated separately of the other Euler numbers. This case is slightly different from the ray tracing example shown in Section 3.5.1, because in the latter there is some "fixed data" (shared by all the subproblems) that is needed every time a row is colored (the list of figures, the width of the screen, and the distance to the viewport) while each Euler number just needs the number to be calculated.

4.1.2 Force interactions

We want to determine the force undergone by each particle in a set of n atoms. The total force f_i acting on each atom x_i is $f_i = \sum_{j=1}^{n} F(x_i, x_j)$, where $F(x_i, x_j)$ denotes the attraction or repulsion between atoms x_i and x_j. We are interested in the value $\mathcal{F} = \sum_{i=1}^{n} f_i$.

We define a function `attraction` that calculates the value $\mathcal{F}$ by using the binary function `attraction` that computes the force interaction between two particle sets that are initially the whole set. It uses auxiliary functions `attraction*`, that calculates the summation of the forces between one particle and all the particles in a set, and `attraction**`, that calculates the force between two particles. We use the `3D` module again for particle positions and to calculate distances.

```
fmod PARTICLE is
 pr 3D .
 sort Particle .
 op particle : Point Float -> Particle .
endfm
```

```
view Particle from TRIV to PARTICLE is
 sort Elt to Particle .
endv

fmod PARTICLES is
 pr 3D .
 pr LIST{Particle} * (sort List{Particle} to ParticleList,
                     op nil to mtParticleList) .

 op K : -> Float .  eq K = 9.0e+9 .

 op attraction : ParticleList -> Float .
 op attraction : ParticleList ParticleList -> Float .
 op attraction* : Particle ParticleList -> Float .
 op attraction** : Particle Particle -> Float .

 vars P P' : Particle .
 vars PL PL' : ParticleList .
 vars Pt Pt' : Point .
 vars F F' R : Float .
 var  V : Vector .

 eq attraction(PL) = attraction(PL, PL) .
 eq attraction(mtParticleList, PL) = 0.0 .
 eq attraction(P PL, PL') = attraction*(P, PL') + attraction(PL, PL') .

 eq attraction*(P, mtParticleList) = 0.0 .
 eq attraction*(P, P' PL) = attraction**(P, P') + attraction*(P, PL) .

 eq attraction**(P, P) = 0.0 .
 ceq attraction**(particle(Pt, F), particle(Pt', F')) = (K * F * F') / (R * R)
  if V := Pt - Pt'  /\  R := module(V) .
endfm
```

We can parallelize this problem dividing the atoms set into smaller subsets $S_1, \ldots, S_k$, generating all the pairs (S_i, S_j) with $i \leq j$, independently calculating the force interaction $F(x, y)$ for every $x \in S_i$ and $y \in S_j$, and adding all the subresults.

4.1.3 Mergesort

The well-known sorting algorithm mergesort for lists of natural numbers can be easily implemented in Maude as follows, where we have used the predefined module `NAT-LIST` for list of natural numbers, renaming the empty list from `nil` to `mtNatList`.

```
fmod SORT is
 pr NAT-LIST * (op nil to mtNatList) .

 op mergesort : NatList -> NatList .
 op merge : NatList NatList -> NatList .

 vars N N' N'' : Nat .
 vars NL NL' NL'' : NatList .
 var  P : Pair .
```

```
  eq mergesort(mtNatList) = mtNatList .
  eq mergesort(N) = N .
  ceq mergesort(N NL N') = merge(mergesort(N NL'), mergesort(NL'' N'))
   if pair(NL', NL'') := halfDivide(NL) .

  eq merge(mtNatList, NL) = NL .
  eq merge(NL, mtNatList) = NL .
  ceq merge(N NL, N' NL') = N merge(NL, N' NL')
   if N <= N' .
  ceq merge(N NL, N' NL') = N' merge(N NL, NL')
   if N' < N .

  sort Pair .
  op pair : NatList NatList -> Pair .

  op halfDivide : NatList -> Pair .
  op halfDivide* : NatList Pair -> Pair .

  eq halfDivide(NL) = halfDivide*(NL, pair(mtNatList, mtNatList)) .
  eq halfDivide*(mtNatList, P) = P .
  eq halfDivide*(N, pair(NL, NL')) = pair(NL N, NL') .
  eq halfDivide*(N NL N', pair(NL', NL'')) =
               halfDivide*(NL, pair(NL' N, N' NL'')) .
 endfm
```

The mergesort algorithm uses the well-known *divide and conquer* approach [39]. In this technique, problems are divided in smaller problems until they are "simple" enough. We can parallelize these algorithms by solving each of these simple problems in parallel.

4.1.4 Traveling salesman problem

The traveling salesman problem is a problem in discrete optimization. It is a prominent illustration of a class of problems in computational complexity theory which are hard to solve. The problem we want to solve is: given a number of cities and the costs of traveling from any city to any other one, what is the cheapest route that visits each city exactly once and then returns to the starting city?

An equivalent formulation in terms of graph theory is: Given a complete weighted graph (where the vertices would represent the cities, the edges would represent the roads, and the weights would be the cost or distance of that road), find a Hamiltonian cycle with the least weight.

We use the *branch and bound* [39] approach to solve it. Branch and bound is an algorithmic technique to find the optimal solution by keeping the best solution found so far. If a partial solution cannot improve the best one, it is abandoned. In these algorithms we need to orient the traversal of the search tree in order to expand first the most promising nodes.

To calculate the initial upper bound we apply a greedy algorithm [39] that selects each time the cheapest edge. The `AUXILIARY-FUNCTIONS` module defines the sorts `City`, `Path` (a sequence of cities), `CityPair` (for pairs of cities), and `Graph` (a partial function from pairs of cities to natural numbers).

```
 fmod GREEDY-TRAVELER is
```

```
pr AUXILIARY-FUNCTIONS .

---- Initial city, Cities (from 0 to N), Roads
op greedyTravel : City Nat Graph -> TravelResult .

---- Initial city, Cities (from 0 to N), Roads, Path, Cost
op greedyTravel : City Nat Graph Path Nat -> TravelResult .

vars C C' C'' : City .
var  G : Graph .
vars N N' N'' : Nat .
var  NI NI' NI'' : NatInf .
vars P P' : Path .
var  PCN : Pair .

--- The initial Path is the initial city, with cost 0
eq greedyTravel(C, N, G) = greedyTravel(C, N, G, C, 0) .

--- The Path contains all the cities (i.e., its size is N), we
--- add the edge from the last city in the path to the initial one.
ceq greedyTravel(C, N, G, P C', N') =
                   result(P C' C, N' + (G [ pair(C', C) ]))
 if size(P) = N .

--- The Path does not contain all the cities, so we expand it.
ceq greedyTravel(C, N, G, P C', N') =
                   greedyTravel(C, N, G, P C' C'', N' + N'')
 if size(P) < N /\
    PCN := cheapest(C', N, P C', G) /\
    C'' := getCity(PCN) /\
    N'' := getCost(PCN) .
```

where `cheapest` looks for the cheapest edge from the last city in the path (`C'`) to other city not used yet, returning a pair with the city and the cost, that are extracted with `getCity` and `getCost`. To calculate the cheapest edge we use a special minimum function `minInf`, that deals with natural numbers and with `infinite`, a special value obtained when there is no road between two cities or when the city has been already used.

```
sort Pair NatInf .
subsort Nat < NatInf .

op pair : City NatInf -> Pair .
op inf : -> NatInf .

op getCity : Pair -> City .
op getCost : Pair -> NatInf .

eq getCity(pair(C, NI)) = C .
eq getCost(pair(C, NI)) = NI .

op minInf : Pair Pair -> Pair [comm] .

eq minInf(pair(C, inf), pair(C', NI)) = pair(C', NI) .
ceq minInf(pair(C, N), pair(C', N')) = pair(C, N)
 if N <= N' .
```

```
  op cheapest : City Nat Path Graph -> Pair .
  op cheapest : City Nat Path Graph Nat -> Pair .

  eq cheapest(C, N, P, G) = cheapest(C, N, P, G, 0) .

  ceq cheapest(C, N, P, G, N') = minInf(PCN, cheapest(C, N, P, G, s(N')))
   if N' < N /\
      PCN := pair(city(N'), getCost(C, city(N'), P, MC)) .
  eq cheapest(C, N, P, G, N) = pair(city(N), getCost(C, city(N), P, G)) .

  --- Cost between two cities, inf if there is no road between them
  --- or the second city has been used in the path. A natural number
  --- with the weight of the pair in the Graph in other case.
  op getCost : City City Path Graph -> NatInf .
  ceq getCost(C, C', P, G) = inf
   if in(C', P) .
  ceq getCost(C, C', P, G) = inf
   if G [ pair(C, C') ] == undefined .
  eq getCost(C, C', P, G) = G [pair(C, C')] [owise] .
 endfm
```

In the branch and bound algorithm we use as lower bound the current cost of the path, and a priority queue to keep the nodes, sorted by cost.

```
 fmod PQUEUE is
  pr TRAVELER-SORTS .

  vars N N' : Nat .
  vars P P' : Path .
  var  ND : Node .
  var  PQ : PNodeQueue .

  sorts Node PNodeQueue .
  subsort Node < PNodeQueue .

  op node : Path Nat -> Node .

  op mtPQueue : -> PNodeQueue .
  op __ : PNodeQueue PNodeQueue -> PNodeQueue [assoc id: mtPQueue] .

  op insert : Node PNodeQueue -> PNodeQueue .
  op getPath : Node -> Path .
  op getCost : Node -> Nat .

  eq insert(ND, mtPQueue) = ND .
  ceq insert(node(P, N), node(P', N') PQ) = node(P, N) node(P', N') PQ
   if N <= N' .
  ceq insert(node(P, N), node(P', N') PQ) =
                        node(P', N') insert(node(P, N), PQ)
   if N > N' .

 eq getPath(node(P, N)) = P .
 eq getCost(node(P, N)) = N .
endfm
```

```
fmod TRAVELER is
 inc PQUEUE .
 inc GREEDY-TRAVELER .

 vars C C' : City .
 var  G : Graph .
 vars N N' N'' UB : Nat .
 var  P : Path .
 var  PCN : Pair .
 var  ND : Node .
 vars PQ PQ' : PNodeQueue .
 var  R : TravelResult .

 op isResult : Node Nat -> Bool .
 eq isResult(node(P, N), N') = size(P) == s(N') .

 ---- Initial city, Cities, Roads
 op travel : City Nat Graph -> TravelResult .

 ---- Initial city, Cities, Roads, Best result so far, Queue
 op travel : City Nat Graph TravelResult PNodeQueue -> TravelResult .
 eq travel(C, N, G) = travel(C, N, G, greedyTravel(C, N, G), node(C, 0)) .
```

While the priority queue is not empty, the best node is not a result, and it is admissible, this node is expanded by adding to each new node a city that is not in the path yet.

```
 ceq travel(C, N, MC, R, ND PQ) = travel(C, N, MC, R, PQ')
  if not isResult(ND, N) /\
     getCost(ND) < getCost(R) /\
     PQ' := expand(ND, N, MC, PQ) .

 ---- Node, Number of cities, Roads, Queue
 op expand : Node Nat Graph PNodeQueue -> PNodeQueue .
 op expand : Node Nat Graph PNodeQueue Nat -> PNodeQueue .
 eq expand(node(P, N), N', G, PQ) = expand(node(P, N), N', G, PQ, 0) .
 ceq expand(node(P C, N), N', G, PQ, N'') =
                 expand(node(P C, N), N', G, insert(ND, PQ), s(N''))
  if N'' <= N' /\
     possible(P C, city(N'')) /\
     ND := node(P C city(N''), N + (G [pair(C, city(N''))])) .
 ceq expand(node(P, N), N', G, PQ, N'') = PQ
  if N'' > N .
 eq expand(ND, N, G, PQ, N') = expand(ND, N, G, PQ, s(N')) [owise] .

 op possible : Path City -> Bool .
 eq possible(P, C) = not in(C, P) .
```

When a result is found, we check if it is better than the current one, updating the current best result.

```
 ceq travel(C, N, MC, R, node(P C', N') PQ) =
             travel(C, N, MC, result(P C' C, UB), PQ)
  if isResult(node(P C', N'), N) /\
     N'' := getCost(R) /\
     UB := N' + (MC [pair(C, C')]) /\
     UB < N'' .

 ceq travel(C, N, MC, R, node(P C', N') PQ) = travel(C, N, MC, R, PQ)
  if isResult(node(P C', N'), N) /\
     N'' := getCost(R) /\
     UB := N' + (MC [pair(C, C')]) /\
     UB >= N'' .
```

When the cost of the best solution found so far is lower than the bound of the first node in the queue (i.e. with the lowest bound) or when the queue becomes empty, the algorithm ends.

```
  ceq travel(C, N, MC, R, ND PQ) = R
   if getCost(ND) >= getCost(R) .
  eq travel(C, N, MC, R, mtPQueue) = R .
 endfm
```

This kind of applications can be parallelized by expanding the nodes at the same time in different processes.

4.2 Parameterized skeletons

An important characteristic of skeletons is their *generality*, that is, the possibility of using them in different applications. For this, most skeletons are parameterized by functions and have a polymorphic type. We describe here a similar approach based on parameterized modules. For each skeleton, we present a theory that requires the sorts and operations that the skeleton needs, a module defining the messages interchanged by objects in different hosts, and a parameterized object-oriented module defining the different classes envolved in the skeleton and their behavior by means of rewrite rules.

We show in the following sections three kinds of skeletons:

Data-parallel skeletons: The source of parallelism is the distribution of data between processors and the application of the same operation to all portions of the data. We apply our methodology to the Farm skeleton (Section 4.3).

Systolic skeletons: The systolic skeletons are used in algorithms in which parallel computation and global synchronization steps alternate [44]. As an example of a systolic skeleton we show the Ring skeleton (Section 4.4).

Task-parallel skeletons: The source of parallelism is the decomposition of a task into different subtasks which can be done in parallel. These subtasks need not be identical [44]. The task-parallel skeletons shown here are:

- Divide and conquer skeleton (Section 4.5).
- Branch and bound skeleton (Section 4.6).
- Pipeline skeleton (Section 4.7).

4.3 Farm skeleton

We show here how to implement a skeleton with *replicated workers* and *fixed data* [44]. In this kind of skeleton, a *master* initially sends the fixed data and some subproblems to all the *workers*. Each time a task is finished by a worker, the subresult is sent to the master where it is combined with the partial result already computed, and a new work is given to that worker, reducing the initial problem. Thus, the tasks are delivered on demand, obtaining an even distribution of the work to be done.

In order to use the parameterized module that implements this skeleton, each concrete application must define a module that satisfies the following `RW_FD-PROBLEM` theory, where

- the sort `Problem` refers to the initial problem;
- `SubProblem` represents the smaller problems solved by the workers;
- `SubResult` corresponds to the results obtained by the workers;
- `Result` is the final result to the original problem; and
- `FixData` contains the data shared by all the workers needed to compute their subproblems.

```
fth RW_FD-PROBLEM is
 inc BOOL .
 sorts Problem SubProblem SubResult Result FixData .
```

The operators required by the theory are:

- `do-work`, that given a subproblem and the fixed data solves the former;

```
    op do-work : SubProblem FixData -> SubResult .
```

- `combine`, that merges the current (partial) result with a new subresult, given the subproblem that was solved. Notice that this operator must be commutative (in the sense that the final result cannot depend on the order in which the combinations are performed) because the subresults may arrive unordered;

```
    op combine : Result SubProblem SubResult -> Result .

    var  R : Result .
    vars SP SP' : SubProblem .
    vars SR SR' : SubResult .
    eq combine(combine(R, SP, SR), SP', SR') =
              combine(combine(R, SP', SR'), SP, SR) [nonexec] .
```

- `new-work`, that extracts a new subproblem from the current problem;

```
    op new-work : Problem -> SubProblem .
```

- `reduce`, that updates the current problem making it smaller;

```
    op reduce : Problem -> Problem .
```

- `finished?`, that checks if the problem has already been solved;

```
    op finished? : Problem -> Bool .
   endfth
```

We need messages for sending the fixed data and new tasks to the workers, and for communicating the subresults to the master. We use a parameterized module because we need the sorts defined in the theory.

```
fmod RW_FD-TRANSMITTED-SYNTAX{P :: RW_FD-PROBLEM} is
 pr TRAVELING-CONTENTS .
 pr OID .

 op fixData : P$FixData -> TravelingContents .
 op new-work : P$SubProblem -> TravelingContents .
 op finished : Loc P$SubProblem P$SubResult -> TravelingContents .
endfm
```

We need lists of subproblems (for the unfinished tasks of each worker). We use the predefined parameterized module `LIST` which is first instantiated with the view `Subproblem` from the theory `TRIV` to the theory `RW_FD-PROBLEM`, and then instantiated with the parameter `P`. The lists sorts are renamed. We use `LIST` with the view `Oid` too, with one difference: the `Oid` view has no free parameters and it does not need the parameter `P`. At start-up, the workers have not fixed data, so we use the `MAYBE` parameterized module (see Section 3.1).

```
view SubProblem from TRIV to RW_FD-PROBLEM is
 sort Elt to SubProblem .
endv

view FixData from TRIV to RW_FD-PROBLEM is
 sort Elt to FixData .
endv

omod RW_FD-SKELETON{P :: RW_FD-PROBLEM} is
 pr LIST{SubProblem}{P} * (sort List{SubProblem}{P} to SubProblemList) .
 pr LIST{Oid} * (sort List{Oid} to OidList, op nil to mtOidList) .
 pr MAYBE{FixData}{P} * (sort Maybe{FixData}{P} to DefFixData,
                        op maybe to null) .
 pr ARCHITECTURE-MSGS .
 pr TRANSMITTED-SYNTAX{P} .
```

First the classes `RW_FD-Worker` and `RW_FD-Master` are defined. The workers have the list with unfinished subproblems (`nextWorks`), the fixed data (`fixData`), that initially is `null`, and the `master` identifier.

```
 class RW_FD-Worker | nextWorks : SubProblemList, fixData : DefFixData,
                      master : Oid .
```

The master stores the initial `problem` (that is reduced each time a new task is sent to a worker), the partial `result`, the list of idle `workers`, and the number of initial tasks assigned to each worker (`numWorks`).

```
class RW_FD-Master | problem : P$Problem, result : P$Result,
                     workers : OidList, numWorks : Nat .
```

The first action the master must take is to deliver the fixed data and the initial tasks to the workers.

```
var  N : Nat .
var  OL : OidList .
var  SR : P$SubResult .
var  R : P$Result .
var  SPL : SubProblemList .
vars W M : Oid .
var  FD : P$FixData .
vars SP SP1 : P$SubProblem .
var  P : P$Problem .

rl [new-worker] :
   < M : RW_FD-Master | fixData : FD, problem : PR, workers : W OL,
                        numWorks : N >
=> < M : RW_FD-Master | problem : update(PR, N), workers : OL  >
   to W : fixData(FD)
   sendTasks(W, PR, N) .
```

where sendTasks and update are functions that generate the initial tasks and reduce the problem.

```
op sendTasks : Oid P$Problem Nat -> Configuration .
ceq sendTasks(W, P, s(N)) = (to W : new-work(new-work(P)))
                            sendTasks(W, reduce(P), N)
 if not finished?(P) .
eq sendTasks(W, P, N) = none [owise] .

op update : P$Problem Nat -> P$Problem .
ceq update(P, s(N)) = update(reduce(P), N)
 if not finished?(P) .
eq update(P, N) = P [owise] .
```

We define now the rules for the worker dealing with the fixed data and new work arrivals, that are kept in the corresponding attributes.

```
rl [rec-fixData] :
   to W : fixData(FD)
   < W : RW_FD-Worker | fixData : null >
=> < W : RW_FD-Worker | fixData : FD > .

rl [new-work] :
   to W : new-work(SP)
   < W : RW_FD-Worker | nextWorks : SPL >
=> < W : RW_FD-Worker | nextWorks : SPL SP > .
```

While the list of unfinished tasks is not empty, the worker must do the following one and send the subresult to the master.

```
 rl [do-work] :
    < W : RW_FD-Worker | fixData : FD, master : M, nextWorks : SP SPL >
 => < W : RW_FD-Worker | nextWorks : SPL >
    to M : finished(W, SP, do-work(SP, FD)) .
```

Notice that the Maude's default bottom-up strategy for reducing terms will apply first the equations defining the operation `do-work`, and then the message containing the result will be transmitted.

The other tasks of the master are to compose the subresults from the workers and give them more work (if possible).

```
 crl [new-work] :
     to M : finished(W, SP, SR)
     < M : RW_FD-Master | fixData : FD, result : R, problem : PR >
  => < M : RW_FD-Master | result : combine(R, SP, SR), problem : reduce(PR) >
     to W : new-work(SP1)
  if not finished?(PR) /\
     SP1 := new-work(PR) .

 crl [no-more-work] :
     to M : finished(W, SP, SR)
     < M : RW_FD-Master | fixData : FD, result : R, problem : PR >
  => < M : RW_FD-Master | result : combine(R, SP, SR) >
  if finished?(PR) .
endom
```

Once the skeleton has been defined, we can instantiate it with concrete applications.

4.3.1 Ray tracing instantiation

In order to obtain a concrete application by using the parameterized skeleton we must know the sorts and operators related to those in the theory `RW_FD-PROBLEM`. We illustrate now how to do it with the ray tracing example [8]. We define the following module, where the included module `ROWTRACER` is shown in Section 3.5.1. The sort `Pair` is declared to define the initial problem (the highest and the lowest `y`), while `World` defines the rest of the problem (the width, the camera, and the list of figures). Partial functions (declared in the predefined module `MAP`) from `Floats` (identifying rows) to `ColorList` are used to represent the final result.

```
fmod RAYTRACING-PROBLEM is
 pr ROWTRACER .
 pr MAP{Float, ColorList} .
 sorts Pair World .

 op pair : Float Float -> Pair .
 op world : FigureList Point Float Float Float -> World .

 op traceRow : Float World -> ColorList .
 op sub-problem : Pair -> Float .
 op reduce : Pair -> Pair .
 op finished? : Pair -> Bool .

 eq traceRow(Y, world(FL, P, XL, XR, N)) = traceRow(P, XL, XR, Y, N, FL) .
```

```
  eq sub-problem(pair(YT, YB)) = YT .
  eq reduce(pair(YT, YB)) = pair(YT - 1.0, YB) .
  eq finished?(pair(YT, YB)) = YT < YB .
 endfm
```

To instantiate the module we create a view and define the mapping between sorts and operators with different name from those in the theory:

```
 view RayTracer from RW_FD-PROBLEM to RAYTRACING-PROBLEM is
  sort Problem to Pair .
  sort SubProblem to Float .
  sort Result to Map{Float, ColorList} .
  sort SubResult to ColorList .
  sort FixData to World .
  op combine(R:Result, SP:SubProblem, SR:SubResult) to
       term insert(SP:Float, SR: ColorList, R:Map{Float, ColorList}) .
  op do-work to traceRow .
  op new-work to sub-problem .
 endv
```

Finally, we instantiate the module RW_FD-SKELETON, where ray tracing can be executed. As in the implementation of the ray tracing problem without skeleton shown in Section 3.5.1, we use the star architecture, placing the master with the center and the workers with nodes. Thus, we must define a view `RT-Complement` that encapsulates the syntax of transmitted messages.

```
 fmod RT-COMPLEMENT is
  pr META-LEVEL .
  pr ARCHITECTURE-MSGS .
  pr RW_FD-TRANSMITTED-SYNTAX{RayTracer} .
 endfm

 view RT-Complement from ARCH-COMPLEMENT to RT-COMPLEMENT is
  op MOD to term upModule('RT-COMPLEMENT, false) .
 endv
```

The initial term of a master with three workers, the point of view located in $(0, 0, 0)$, a screen of size 201×151, and figures traced from $z = 100$ is:

```
 mod RAYTRACING-CENTER is
  pr RW_FD-SKELETON{RayTracer} .
  pr STAR-CENTER{RT-Complement} .
  pr RANDOM .

  op figs : -> FigureList .
  eq figs = ... .
 endm

 erew <> < l(ip0, 0) : StarCenter |
            state : idle,
            neighbors : empty,
            defNeighbor : null,
            port : 60039 >
        < o(l(ip0, 0), 0) : RW_FD-Master |
            fixData : world(figs, < 0.0, 0.0, 0.0 >, -75.0, 75.0, 100.0),
            problem : pair(100.0, -100.0),
            result : empty,
            workers : o(l(ip1, 1), 0) o(l(ip2, 2), 0),
            numWorks : 3,
            counter : 0 > .
```

where `figs` is a list of ten random spheres filtered if they are farther than 1000 and the `ipi` are IP addresses.

The unique difference among the workers is their names (which depend on an IP address). The initial term for one of the workers is:

```
mod RAYTRACING-NODE is
 pr RW_FD-SKELETON{RayTracer} .
 pr STAR-NODE{RT-Complement} .
endm

erew <> < l(ip1, 0) : StarNode |
           state : idle,
           neighbors : empty,
           defNeighbor : null,
           center : ip0,
           port : 60039 >
        < o(l(ip1, 0), 0) : RW_FD-Worker |
           master : o(l(ip0, 0), 0),
           nextWorks : nil,
           counter : 0,
           fixData : null > .
```

Once the module has been instantiated, its behavior is equivalent to the sequential module described in Section 3.5, as we will prove in Section 6.6.

4.3.2 Mandelbrot instantiation

A similar example of farm skeleton is the computation of the Mandelbrot set [45]. Using the `COLOR` module described in the ray tracing example (Section 3.5.1) the sequential implementation of the functions needed to instantiate the skeleton is almost straightforward. The function `mandelbrot*` below traverses a row and calculates for each pixel its color in function of the iterations used (by using the auxiliary function `getColor`). The module also defines the maximum number of iterations and the precision of the set.

```
fmod MANDELBROT-ROW is
 pr FLOAT .
 pr INT .
 pr COLOR .

 op maxIt : -> Nat .
 eq maxIt = 1000 .

 op precision : -> Float .
 eq precision = 0.008 .

 sort ColorList .
```

```
  --- Color union
  op __ : ColorList ColorList -> ColorList [assoc] .

  --- Xleft Xright Y
  op mandelbrot* : Float Float Float -> ColorList .
  op mandelbrot* : Float Float Float Float Nat -> Color .

  var  N : Nat .
  vars Xl Xr Ymax Ymin X0 Y0 X Y : Float .

  ceq mandelbrot*(Xl, Xr, Y) =
   mandelbrot*(Xl, Y, Xl, Y, maxIt) mandelbrot*(Xl + precision, Xr, Y)
   if Xl < Xr .
  ceq mandelbrot*(Xl, Xr, Y) = mandelbrot*(Xl, Y, Xl, Y, maxIt)
   if Xl >= Xr .

  --- Iterations of the algorithm from maxIt to 0, if we have not
  --- reached the precision in all the iterations, we assign
  --- the default color (d).
  eq mandelbrot*(X, Y, X0, Y0, 0) = d .
  --- When we reach the precision required, we decide the color
  --- depending on the number of iterations used.
  eq mandelbrot*(X, Y, X0, Y0, s(N)) =
   if (X * X) + (Y * Y) < 4.0 then
    mandelbrot*(((X * X) - (Y * Y)) + X0, 2.0 * X * Y + Y0, X0, Y0, N)
   else decideColor(s(N))
   fi .

  op decideColor : Nat -> Color .
  ceq decideColor(N) = d if N < maxIt quo 4 .
  ceq decideColor(N) = r if N >= maxIt quo 4 /\ N < maxIt quo 2 .
  ceq decideColor(N) = b if N >= maxIt quo 2 /\ N < (3 * maxIt) quo 4 .
  ceq decideColor(N) = m if N >= (3 * maxIt) quo 4 /\ N <= maxIt .
 endfm
```

We can now specify the module MANDELBROT-PROBLEM to implement the skeleton. We define again the sorts Pair (that keeps the current row and the lowest row) and World (that keeps the width of the screen). We adapt the other functions to use the same arity that those defined in the theory.

```
 fmod MANDELBROT-PROBLEM is
  pr MANDELBROT-ROW .
  pr MAP{Float, ColorList} .

  sort Pair World .

  var Xl Xr Y Ymax Ymin : Float .

  op pair : Float Float -> Pair .
  op world : Float Float -> World .

  op mandelbrot : Float World -> ColorList .
  op sub-problem : Pair -> Float .
  op reduce : Pair -> Pair .
  op finished? : Pair -> Bool .

  eq mandelbrot(Y, world(Xl, Xr)) = mandelbrot*(Xl, Xr, Y) .
  eq sub-problem(pair(Ymax, Ymin)) = Ymax .
  eq reduce(pair(Ymax, Ymin)) = pair(Ymax - 1.0, Ymin) .
  eq finished?(pair(Ymax, Ymin)) = Ymax < Ymin .
 endfm
```

Now we can define the view from the theory

```
 view Mandelbrot from RW_FD-PROBLEM to MANDELBROT-PROBLEM is
  sort Problem to Pair .
  sort SubProblem to Float .
  sort Result to Map{Float, ColorList} .
  sort SubResult to ColorList .
  op combine(R:Result, SP:SubProblem, SR:SubResult) to
       term insert(SP:Float, SR: ColorList, R:Map{Float, ColorList}) .
  op do-work to mandelbrot .
  op new-work to sub-problem .
 endv
```

and instantiate the skeleton with it

```
 mod MANDELBROT-EXAMPLE is
  pr RW_FD-SKELETON{Mandelbrot} .
 endm
```

4.3.3 Euler instantiations

In some problems the fixed data is not needed; we have implemented a slightly modified skeleton to deal with this situation. It is very similar to the replicated workers skeleton above, but all the rules dealing with the fixed data are either not needed or simplified.

We show here a simple example using this new skeleton: the distributed implementation of the Euler numbers problem shown in Section 4.1.1. In this problem we consider as a single work to calculate each $\varphi(i)$.

The only sort involved in this problem is Nat, so every sort in the skeleton is mapped to it.

```
 view Euler from RW-PROBLEM to EULER is
  sort Problem to Nat .
  sort SubProblem to Nat .
  sort Result to Nat .
  sort SubResult to Nat .
```

The operations are very simple too: combining two results is just adding them; the work that must be done is the function euler from module EULER in Section 4.1.1 (the target module in the view); a new work given the problem N is just N; we reduce the problem by subtracting 1; and we have finished when the number reaches 0.

```
  op combine(R:Result, S:SubProblem, SR:SubResult) to term (R:Nat + SR:Nat) .
  op do-work to euler .
  op new-work(N:Problem) to term N:Nat .
  op reduce(N:Problem) to term sd(N:Nat, 1) .
  op finished?(N:Problem) to term (N:Nat == 0) .
 endv
```

Calculating $\varphi(x)$ may be quite faster than the communication with the master, so it is possible that most of the computation time is used in communication. To avoid this problem we can make the granularity of the works coarser by computing more than one number in each step. To do this we only need to make small changes in the instantiation module, while obviously the skeleton remains unmodified. We show here an example where we calculate 20 numbers in each step.

```
 fmod EULER20 is
  pr EULER .

  vars N N' : Nat .

  op euler20 : Nat -> Nat .
  op euler20* : Nat Nat Nat -> Nat .

  eq euler20(N) = euler20*(N, if N > 20 then sd(N, 20) else 1 fi, 0) .
  ceq euler20*(N, N', N'') = N''
   if N' >= N .
  eq euler20*(s(N), N', N'') = euler20*(N, N', N'' + euler(s(N))) [owise] .
 endfm
```

Now the view only needs two changes:

```
 view Euler20 from RW-PROBLEM to EULER20 is
    ...
  op do-work to euler20 .
  op reduce(N:Problem) to term (if N:Nat > 20 then sd(N:Nat, 20) else 0 fi) .
    ...
 endv
```

4.4 Systolic ring skeleton

In this skeleton, a master splits the problem among all the workers, that are organized in a circular list because they must share some data through it. When the workers have both initial and shared data, they do their work, combine the partial result, and give the new shared data to the next worker. In order to have the first shared data, it must be produced by the worker itself. When a worker finishes all its tasks, it sends its subresult to the master, that will combine them in order.

We define a theory `SYSTOLIC-RING-PROBLEM` with the following sorts:

- `Problem`, that defines the sort of the initial data;

- `Result`, that describes the sort of the final data;

- `ProblemList` and `ResultList` are respectively lists of `Problem` and `Result`;[1]

[1]We cannot use here the predefined module `LIST` because we would need a parametric theory, that is not allowed in Core Maude yet.

- SharedProblem is the sort of the data that is passed by all the workers; and
- Pair builds pairs of Result and SharedProblem.

```
fth SYSTOLIC-RING-PROBLEM is
 inc BOOL .
 inc NAT .

 sorts Problem Result ProblemList ResultList SharedProblem Pair .
 subsort Problem < ProblemList .
 subsort Result < ResultList .
```

The theory defines the following operators:

- mtRL, mtPL, and __ are the list constructors;

```
    op mtPL : -> ProblemList .
    op __ : ProblemList ProblemList -> ProblemList [assoc id: mtPL] .

    op mtRL : -> ResultList .
    op __ : ResultList ResultList -> ResultList [assoc id: mtRL] .
```

- pair is the Pair constructor;

```
    op pair : Result SharedProblem -> Pair .
```

- divide splits the initial problem into a list of problems;

```
    op divide : Problem Nat -> ProblemList .
```

- initialSharedProblem extracts from the initial data the shared one;

```
    op initialSharedProblem : Problem -> SharedProblem .
```

- do-work computes a pair given the initial and the shared data;

```
    op do-work : Problem SharedProblem -> Pair .
```

- combine, used by the workers, merges the current partial result with a new one:

```
    op combine : Result Result -> Result .
```

- combineAll, used by the master, merges all the partial results from the workers:

```
    op combineAll : ResultList -> Result .
```

- finished? checks if a worker has finished all its tasks:

```
    op finished? : Problem SharedProblem -> Bool .
   endfth
```

We need the following messages:

- `initial-work`, that communicates the initial data;
- `shared-data`, that delivers the shared problem from one worker to the next one; and
- `finished`, that sends a (numbered) result to the master.

```
fmod SYSTOLIC-RING-TRANSMITTED-SYNTAX{P :: SYSTOLIC-RING-PROBLEM} is
 pr TRAVELING-CONTENTS .

 op initial-work : P$Problem -> TravelingContents .
 op shared-data : Nat P$SharedProblem -> TravelingContents .
 op finished : Nat P$Result -> TravelingContents .
endfm
```

We define views to `Problem`, `Result`, and `SharedProblem`, that will be needed by the skeleton.

```
view Problem from TRIV to SYSTOLIC-RING-PROBLEM is
 sort Elt to Problem .
endv

view Result from TRIV to SYSTOLIC-RING-PROBLEM is
 sort Elt to Result .
endv

view SharedProblem from TRIV to SYSTOLIC-RING-PROBLEM is
 sort Elt to SharedProblem .
endv
```

The module defining the skeleton receives `SYSTOLIC-RING-PROBLEM` as a parameter.

```
omod SYSTOLIC-RING-SKELETON{P :: SYSTOLIC-RING-PROBLEM} is
 pr MAYBE{Problem}{P} * (sort Maybe{Problem}{P} to DefProblem,
                         op maybe to nullP) .
 pr MAYBE{Result}{P} * (sort Maybe{Result}{P} to DefResult,
                        op maybe to nullR) .
 pr MAYBE{SharedData}{P} * (sort Maybe{SharedData}{P} to DefSharedData,
                            op maybe to nullSD) .
 pr ARCHITECTURE-MSGS .
 pr SYSTOLIC-RING-TRANSMITTED-SYNTAX{P} .
 pr CONVERSION .
 pr LIST{Oid} * (sort List{Oid} to OidList,
                 op nil to mtOidList) .
```

The class `SWorker` has the following attributes:

- `problem`, that keeps the initial data;
- `shared`, that contains the shared data;
- `result`, that stores the partial result;

- `numWorker`, that identifies the worker;
- `nextWorker`, that saves the identifier of the next worker;
- `master`, that keeps the master identifier; and
- `counter`, that keeps track of the number of shared data messages that have been received from the previous worker.

```
class SWorker | problem : DefProblem, shared : DefSharedData,
                result : Result, numWorker : Nat, nextWorker : Oid,
                master : Oid, counter : Nat .
```

The class `SMaster` has the following attributes:

- `initialProblem`, that stores the initial problem;
- `problems`, that keeps the list of problems once the initial data has been divided;
- `results`, that is a list of results where the partial results from the workers are kept in order;
- `counter`, that keeps track of the number of partial results that have been accepted (appended in the list `results`) from the workers;
- `result`, that contains the final result;
- `workers`, that stores the list of workers; and
- `numWorkers`, that keeps the number of workers.

```
class SMaster | initialProblem : DefProblem, problems : ProblemList,
                results : ResultList, counter : Nat, result : Result,
                workers : OidList, numWorkers : Nat .
```

The first thing that must be done is to divide the initial data into a list of problems, that can be delivered to all the workers.

```
vars N N' : Nat .
vars R R' : P$Result .
var  RL : P$ResultList .
vars W W' M : Oid .
vars P P' : P$Problem .
var  PL : P$ProblemList .
vars SD SD' : P$SharedData .
var  OL : OidList .

rl [divide] :
   < M : SMaster | initialProblem : P, problems : mtPL, numWorkers : N >
=> < M : SMaster | initialProblem : nullP, problems : divide(P, N) > .
```

Once the data has been split, the master sends it to the workers, that store the data and extract the shared data.

```
  rl [new-worker] :
     < M : SMaster | problems : P PL, workers : W OL >
  => < M : SMaster | problems : PL, workers : OL >
     to W : initial-work(P) .

  rl [initial-data] :
     to W : initial-work(P)
     < W : SWorker | problem : nullP, shared : nullSP >
  => < W : SWorker | problem : P, shared : initialSharedData(P), counter : 0 > .
```

When the shared attribute is not `nullSD`, the worker can do a new work and send the updated shared data to the next worker.

```
  crl [working] :
      < W : SWorker | problem : P, nextWorker : W', shared : SD,
                      counter : N, result : R >
   => < W : SWorker | shared : nullSD, result : combine(R, R') >
      to W' : shared-data(s(N), SD')
   if pair(R', SD') := do-work(P, SD) .
```

Notice that once the work is computed the shared data becomes `nullSD`, in order to allow the arrival of the next shared data from the previous worker.

When the next shared data that a worker must keep arrives, it checks whether the work is not finished, and the shared data must be kept, or the work is finished and the shared data can be deleted.

```
  crl [new-work] :
      to W : shared-data(s(N), SD)
      < W : SWorker | counter : N, problem : P, shared : nullSD >
   => < W : SWorker | counter : s(N), shared : SD >
   if not finished?(P, SD) .

  crl [finished] :
      to W : shared-data(s(N), SD)
      < W : SWorker | counter : N, problem : P, master : M, result : R,
                      numWorker : N' >
   => < W : SWorker | >
      to M : finished(N', R)
   if finished?(P, SD) .
```

The master keeps in `results` the partial results in an ordered fashion, using the `counter` attribute.

```
  rl [partial-result] :
     to M : finished(s(N), R)
     < M : SMaster | results : RL, counter : N >
  => < M : SMaster | results : RL R, counter : s(N) > .
```

When all the results have arrived, the master combines them.

```
  rl [completed] :
     < M : SMaster | results : RL, numWorkers : N, counter : N >
  => < M : SMaster | results : mtRL, result : combineAll(RL), counter : 0 > .
```

4.4.1 Force interaction instantiation

Using the module PARTICLES from Section 4.1.2 we can easily implement a distributed version of the atoms problem. First, we define a wrapper for the particle lists, in order to define a list of lists. We use a list of floats to keep the results.

```
fmod WRAPPER is
 pr LIST{Particle} * (sort List{Particle} to ParticleList,
                      op nil to mtParticleList) .

 sort WrappedParticles .
 op w : ParticleList -> WrappedParticles .
endfm

view WrappedParticles from TRIV to WRAPPER is
 sort Elt to WrappedParticles .
endv

fmod PARTICLES-PROBLEM is
 pr PARTICLES .
 pr LIST{WrappedParticles} .
 pr LIST{Float} * (op nil to nilFL) .
```

We define for ParticleList the functions size (that returns the length), take (that returns the first N elements), and drop (that deletes the first N elements).

```
 var  P : Particle .
 var  F : Float .
 var  FL : List{Float} .
 vars PL PL' : ParticleList .
 vars N N' : Nat .
 var  WPL : WrappedParticles .

 op size : ParticleList -> Nat .
 op take : ParticleList Nat -> ParticleList .
 op drop : ParticleList Nat -> ParticleList .

 eq size(mtParticleList) = 0 .
 eq size(P PL) = s(size(PL)) .

 eq take(P PL, s(N)) = P take(PL, N) .
 eq take(PL, N) = mtParticleList [owise] .

 eq drop(P PL, s(N)) = drop(PL, N) .
 eq drop(PL, N) = PL [owise] .
```

The function divide splits the particle list in N lists. The last one may be larger than the others, because the number of particles could be not a multiple of N.

```
 op divide : WrappedParticles Nat -> List{WrappedParticles} .
 op take : WrappedParticles Nat Nat -> List{WrappedParticles} .
 ceq divide(w(PL), N) = take(w(PL), N, N')
  if N' := size(PL) quo N .
 eq take(WPL, N, 1) = WPL .
 eq take(w(PL), N, s(s(N'))) = w(take(PL, N)) take(w(drop(PL, N)), N, s(N')) .
```

The `combineAll` function for float lists adds all the elements.

```
 op combineAll : List{Float} -> Float .
 eq combineAll(nilFL) = 0.0 .
 eq combineAll(F FL) = F + combine(FL) .
```

We redefine the `attraction` function for wrapped lists.

```
 op attraction : WrappedParticles WrappedParticles -> Pair .
 eq attraction(w(PL), w(PL')) = < attraction(PL, PL'), w(PL') > .
```

Finally, we define the `Pair` sort.

```
 sort Pair .
 op <_,_> : Float WrappedParticles -> Pair .
endfm
```

Now we can instantiate the skeleton with the module above.

```
view Particles from SYSTOLIC-RING-PROBLEM to PARTICLES-PROBLEM is
 sort Problem to WrappedParticles .
 sort Result to Float .
 sort SharedProblem to WrappedParticles .
 sort ProblemList to List{WrappedParticles} .
 sort ResultList  to List{Float} .

 op mtPL to nil .
 op mtRL to nilFL .
 op pair to <_,_> .
 op initialSharedProblem(P:Problem) to term P:WrappedParticles .
 op combine(R:Result, R':Result) to term R:Float + R':Float .
 op do-work to attraction .
 op finished? to _==_ .
endv
```

We also define the view `Particles-Complement`.

```
fmod PARTICLES-COMPLEMENT is
 pr OID .
 pr META-LEVEL .
 pr SYSTOLIC-RING-TRANSMITTED-SYNTAX{Particles} .
 pr ARCHITECTURE-MSGS .
endfm

view Particles-Complement from ARCH-COMPLEMENT to PARTICLES-COMPLEMENT is
 op MOD to term upModule('PARTICLES-COMPLEMENT, false) .
endv
```

The most suitable architecture for this skeleton is the centralized ring (see Section 3.4). The master is located in the center while the workers are placed in the nodes of the ring (one with a `CRingLast` object and the rest with the `CRingNode` objects).

The initial configuration for the location with the master is:

```
mod PARTICLES-CENTER is
 pr SYSTOLIC-RING-SKELETON{Particles} .
 pr STAR-CENTER{Particles-Complement} .
 pr PARTICLES-GENERATOR .
endm

erew <> < l(ip0, 0) : StarCenter |
                 state : idle,
                 neighbors : empty,
                 defNeighbor : null,
                 port : 60038 >
        < o(l(ip0, 0), 0) : SMaster |
           workers : o(l(ip1, 0), 0) o(l(ip2, 0), 0)
                     o(l(ip3, 0), 0),
           numWorkers : 3,
           result : nullR,
           counter : 1,
           initialProblem : w(atomGenerator(20)),
           problems : nil,
           results : nilFL > .
```

The initial configurations for the locations with workers are very similar. For example, we show below a configuration with a worker located in a ring node that is not the last one.

```
mod PARTICLES-CRNODE is
 pr SYSTOLIC-RING-SKELETON{Particles} .
 pr CENTRALIZED-RING-NODE{Particles-Complement} .
endm

erew <> < l(ip1, 0) : CRingNode |
                     state : idle,
                     neighbors : empty,
                     defNeighbor : null,
                     port : 60039,
                     nextIP : ip2,
                     nextPort : 60040,
                     centerIP : ip0,
                     centerPort : 60038 >
        < o(l(ip1, 0), 0) : SWorker |
                     problem : nullP,
                     master : o(l(ip0, 0), 0),
                     shared : nullSD,
                     result : 0.0,
                     numWorker : 1,
                     nextWorker : o(l(ip2, 0), 0),
                     counter : 0 > .
```

4.5 Divide and conquer skeleton

Divide and conquer algorithms clearly offer good potential for parallel evaluation. It is not difficult to see that recursively defined subproblems may be evaluated in parallel if sufficient processors are available. The whole execution of a divide and conquer algorithm amounts to the evaluation of a dynamically evolving tree of processes, one for each subproblem generated.

However, we show here an implementation based on the replicated workers scheme (see Section 4.3), that allows a balanced distribution of the leaves of the problem tree. This implementation is suitable when the decomposition of the problems and the composition of the results are irrelevant compared to the resolution of the subproblems. The master divides the initial problem into subproblems, that are delivered to the workers. The structure of the subproblems is kept in a tree in order to be able to combine their subresults in the appropriate order and get the final result.

First we define the ID module, that defines the identifiers of the nodes in the tree as lists of natural numbers.

```
fmod ID is
 pr NAT-LIST * (op nil to mtNatList) .

 sort Id .
 op id : NatList -> Id .
endfm
```

The TREE module is defined as follows:

```
fmod TREE{X :: TRIV} is
 pr EXT-BOOL .
 pr ID .
 pr MAYBE{X} * (op maybe to null) .

 var T : Tree .
 var F : Forest .
 var D : Maybe{X} .
```

We define generic trees, so we need a sort Forest to allow any number of siblings in each node. As seen above, a node identifier is a list of natural numbers. These numbers specify the number of sibling we must go through in each level of the tree to reach the identified node. Notice that the data in the node may be null.

```
 sorts Tree Forest .
 subsort Tree < Forest .

 op mtForest : -> Forest .
 op __ : Forest Forest -> Forest [assoc id: mtForest] .

 op empty : -> Tree .
 op tree : Maybe{X} Forest -> Tree .
```

We define the following operations over trees and forests: size returns the number of trees in a forest; getData returns the value in the root of the tree; and allWithValues checks if all the trees in a forest have data in their roots.

```
 op size : Forest -> Nat .
 eq size(mtForest) = 0 .
 eq size(T F) = s(size(F)) .
```

```
  op getData : Tree -> Maybe{X} .
  eq getData(empty) = null .
  eq getData(tree(D, F)) = D .

  op allWithValues? : Forest -> Bool .
  eq allWithValues?(mtForest) = true .
  eq allWithValues?(T F) = getData(T) =/= null and-then allWithValues?(F) .
 endfm
```

We define now a theory with operators that allow the skeleton to generate and solve the problem tree. The sorts `Problem` and `Result` define the initial and final data, while `ProblemList` and `ResultList` are, respectively, lists of `Problem` and `Result` with the corresponding operators for empty lists and composition. The function `divide` splits a problem into a list of (sub)problems, finishing when the problem `isTrivial`. Each trivial task is computed with `solve`. The function `combine` merges a list of (sub)results into a new (sub)result.

```
 fth DC-PROBLEM is
  inc BOOL .

  sorts Problem Result ProblemList ResultList .
  subsort Problem < ProblemList .
  subsort Result < ResultList .

  op mtProblemList : -> ProblemList .
  op mtResultList : -> ResultList .
  op __ : ProblemList ProblemList -> ProblemList [assoc id: mtProblemList] .
  op __ : ResultList ResultList -> ResultList [assoc id: mtResultList] .

  op divide : Problem -> ProblemList .
  op isTrivial : Problem -> Bool .
  op combine : ResultList -> Result .
  op solve : Problem -> Result .
 endfth
```

In the skeleton we will need a tree instantiated with results, so we define now the appropriate view.

```
 view Result from TRIV to DC-PROBLEM is
  sort Elt to Result .
 endv
```

Only two messages are used: `new-work` transmits new tasks to the workers, while `finished` is used to communicate new results to the master.

```
 fmod DC-TRANSMITTED-SYNTAX{P :: DC-PROBLEM} is
  pr ID .
  pr TRAVELING-CONTENTS .
  pr OID .

  op new-work : Id P$Problem -> TravelingContents .
  op finished : Oid Id P$Result -> TravelingContents .
 endfm
```

```
omod DC-SKELETON{P :: DC-PROBLEM} is
 pr ARCHITECTURE-MSGS .
 pr TRANSMITTED-SYNTAX{P} .
 pr TREE{Result}{P} .
 pr LIST{Oid} * (sort List{Oid} to OidList, op nil to mtOidList) .
```

We keep together each problem and its identifier with the operator `<_,_>`, that have sort `Task`. We define lists of `Task`s by using the juxtaposition operator.

```
 sorts Task TaskList .
 subsort Task < TaskList .
 op <_,_> : Id P$Problem -> Task .
 op mtTaskList : -> TaskList .
 op __ : TaskList TaskList -> TaskList [assoc id: mtTaskList] .

 var  N : Nat .
 var  OL : OidList .
 vars R R' : P$Result .
 var  RL : P$ResultList .
 var  PL : P$ProblemList .
 vars W O : Oid .
 var  P : P$Problem .
 var  NL : NatList .
 vars F F' : Forest .
 vars T T' : Tree .
 vars ID I I' : Id .
 vars TL TL' : TaskList .
 var  Tk : Task .
```

We define the classes for the master and the workers. A worker keeps information about the `master` identifier and the list of unfinished `tasks`:

```
 class DCWorker | master : Oid, tasks : TaskList .
```

The master can be in two states. At the start, it is in `initial` state, and once the problem has been divided and it can deliver the subproblems to the workers, it reaches the `working` state.

```
 sort DCMasterState .
 ops initial working : -> DCMasterState .
```

The master keeps information about the initial problem (`initialData`); the `workers` identifiers; the task list (`problems`); the `resultTree`; its state (`masterState`); and the number of tasks initially dispatched to each worker (`numWorks`).

```
 class MasterRO | initialData : P$Problem, workers : OidList,
                  problems : TaskList, resultTree : Tree,
                  masterState : MasterState, numWorks : Nat .
```

First, the master transforms the initial problem in a list of subproblems, and creates the initial result tree.

```
crl [start] :
    < O : DCMaster | problems : mtTaskList, initialData : P,
                     resultTree : empty, masterState : initial >
 => < O : DCMaster | problems : TL, resultTree : T, masterState : working >
 if p(TL, T) := getInitialData(P) .
```

where getInitialData uses the operations divide and isTrivial from the theory to obtain a pair with the list of tasks and the result tree (that initially has all its nodes without data).

```
sort Pair .
op p : TaskList Forest -> Pair .

op getInitialData : P$Problem -> Pair .
op getInitialData : P$Problem Id -> Pair .
op getInitialData* : P$ProblemList Id Nat -> Pair .

eq getInitialData(P) = getInitialData(P, id(mtNatList)) .

ceq getInitialData(P, ID) = p(< ID, P >, tree(noData, mtForest))
 if isTrivial(P) .

ceq getInitialData(P, ID) = p(TL, tree(noData, F))
 if not isTrivial(P) /\
    PL := divide(P) /\
    p(TL, F) := getInitialData*(PL, ID, 0) .

eq getInitialData*(mtProblemList, ID, N) = p(mtTaskList, mtForest) .
ceq getInitialData*(P PL, id(NL), N) = p(TL' TL, T F)
 if p(TL', T) := getInitialData(P, id(NL N)) /\
    p(TL, F) := getInitialData*(PL, id(NL), s(N)) .
```

Once the list of problems has been calculated, the master must transmit the initial tasks to the workers, which is calculated with sendWorks. The workers will keep them in the task list.

```
rl [new-worker] :
   < O : DCMaster | problems : TL, workers : O' OL, masterState : working,
                    numWorks : N >
=> < O : DCMaster | problems : update(TL, N), workers : OL >
   sendWorks(O', TL, N) .

rl [new-work] :
   to W : new-work(I, P)
   < W : DCWorker | tasks : TL >
=> < W : DCWorker | tasks : TL < I, P > > .
```

where sendWorks and update just extract tasks from the list.

```
op sendWorks : Oid TaskList Nat -> Configuration .
eq sendWorks(O, < I, P > TL, s(N)) = to O : new-work(I, P)
                                     sendWorks(O, TL, N) .
eq sendWorks(O, TL, N) = none [owise] .
```

```
  op update : TaskList Nat -> TaskList .
  eq update(Tk TL, s(N)) = update(TL, N) .
  eq update(TL, N) = TL [owise] .
```

Eventually, a task is finished and sent to the server, that inserts it in the result tree. While the whole problem is not solved, new subproblems are delivered.

```
  rl [do-work] :
     < W : DCWorker | master : O, tasks : < I, P > TL >
  => < W : DCWorker | tasks : TL >
     to O : finished(W, I, solve(P)) .

  rl [new-work] :
     to O : finished(W, I, R)
     < O : DCMaster | resultTree : T, problems : < I', P > TL,
                      masterState : working >
  => < O : DCMaster | resultTree : insert*(I, R, T), problems : TL >
     to W : new-work(I', P) .

  rl [no-more-work] :
     to O : finished(W, I, R)
     < O : DCMaster | resultTree : T, problems : mtTaskList,
                      masterState : working >
  => < O : DCMaster | resultTree : insert*(I, R, T) > .
```

Notice the use of `insert*`, an operator that inserts a new element in the tree and then tries to recursively `combine` the leaves, checking if all the siblings of a node have already a value.

```
  ---- Merging insert
  op insert* : Id P$Result Tree -> Tree .
  eq insert*(id(mtNatList), R, tree(null, F)) = tree(R, F) .
  ceq insert*(id(N NL), R, tree(null, F)) =
            if allWithValues?(F') then
            tree(combine(getResults(F')), mtForest)
            else tree(null, F')
            fi
   if F' := insertF*(id(N NL), R, F) .

  op insertF* : Id P$Result Forest -> Forest .
  eq insertF*(ID, R, mtForest) = mtForest .
  eq insertF*(id(s(N) NL), R, T F) = T insertF*(id(N NL), R, F) .
  eq insertF*(id(0 NL), R, T F) = insert*(id(NL), R, T) F .
 endom
```

4.5.1 Mergesort instantiation

We show how to instantiate the skeleton with the mergesort algorithm. We define the sort `List`, that encapsulates the lists of natural numbers with the operator `l`, and we define lists of lists of natural numbers in the sort `ListList`. We use the module `SORT` shown in Section 4.1.3.

```
 fmod MERGESORT-PROBLEM is
  pr SORT .
  sort List ListList .
  subsort List < ListList .
  op l : NatList -> List .

  op mtListList : -> ListList .
  op __ : ListList ListList -> ListList [assoc id: mtListList] .
```

In the instantiation, `List` will represent `Problem` and `Result`, while `ListList` the corresponding lists.

We define now the operations needed by the theory. The operator `divide` splits a list into two sublists, while we consider as trivial a list with at most 50 elements. These trivial cases are solved by using the sequential version of `mergesort` shown in Section 4.1.3. Finally, we use `merge` to combine the elements from the subresults.

```
 op divide : List -> ListList .
 ceq divide(l(NL)) = l(NL') l(NL'')
  if pair(NL', NL'') := halfDivide(NL) .

 op trivial : List -> Bool .
 op trivial* : NatList Nat -> Bool .
 eq trivial(l(NL)) = trivial*(NL, 0) .
 eq trivial*(mtNatList, N) = N <= 50 .
 eq trivial*(NL, 51) = false .
 eq trivial*(N NL, N') = trivial*(NL, s(N')) [owise] .

 op merge : ListList -> List .
 eq merge(l(NL) l(NL')) = l(merge(NL, NL')) .

 op mergesort : List -> List .
 eq mergesort(l(NL)) = l(mergesort(NL)) .
```

We can now declare a view from the skeleton to the module with the operators above:

```
view Mergesort from DC-PROBLEM to MERGESORT-PROBLEM is
 sort Problem to List .
 sort Result to List .
 sort ProblemList to ListList .
 sort ResultList to ListList .

 op mtProblemList to mtListList .
 op mtResultList to mtListList .

 op isTrivial to trivial .
 op solve to mergesort .
 op combine to merge .
endv
```

Note that although we map `solve` to `mergesort`, we could use any other sort method.

We define a view `Mergesort-Complement` to instantiate the architecture.

```
fmod MERGESORT-COMPLEMENT is
 pr DC-TRANSMITTED-SYNTAX{Mergesort} .
 pr ARCHITECTURE-MSGS .
 pr META-LEVEL .
endfm

view Mergesort-Complement from ARCH-COMPLEMENT to MERGESORT-COMPLEMENT is
 op MOD to term upModule('MERGESORT-COMPLEMENT, false) .
endv
```

We can now instantiate the skeleton with this view and execute some examples. We use the star architecture, with the master placed in the center and the workers in the nodes. The initial configuration for the master with four workers and a list of 1000 elements (generated with the gen function below) is:

```
mod MERGESORT-CENTER is
 pr RANDOM .
 pr DC-SKELETON{Mergesort} .
 pr STAR-CENTER{Mergesort-Complement} .

 op gen : Nat -> List .
 op gen* : Nat -> NatList .

 var N : Nat .
 eq gen(N) = l(gen*(N)) .
 eq gen*(0) = random(0) .
 eq gen*(s(N)) = random(s(N)) gen*(N) .
endm

erew <> < l(ip0, 0) : StarCenter |
                 state : idle,
                 neighbors : empty,
                 defNeighbor : null,
                 port : 60039 >
        < o(l(ip0, 0), 0) : DCMaster |
                 masterState : initial,
                 initialData : l(gen(1000)),
                 workers : o(l(ip1, 0), 0) o(l(ip2, 0), 0)
                           o(l(ip3, 0), 0) o(l(ip4, 0), 0),
                 numWorks : 3,
                 resultTree : empty,
                 problem : mtTaskList > .
```

where the ipi are IP addresses. The initial configuration for one of the workers is:

```
mod MERGESORT-NODE is
 pr DC-SKELETON{Mergesort} .
 pr STAR-NODE{Mergesort-Complement} .
endm

erew <> < l(ip1, 0) : StarNode |
           state : idle,
           neighbors : empty,
           defNeighbor : null,
           center : ip0,
           port : 60039 >
        < o(l(ip1, 0), 0) : DCWorker |
           master : o(l(ip0, 0), 0),
           tasks : mtTaskList > .
```

4.6 Branch and bound skeleton

Branch and bound algorithms [39] traverse a search tree looking for the best solution, as shown in Section 4.1.4. We define the theory BB-PROBLEM with the sorts and the operations involved in the skeleton:

- We consider each node of the tree as a partial solution of sort PartialResult.
- The lists of partial results are represented with the sort PRList, and is constructed with mtPRList and __.
- The sort FixData is used to represent the data shared by all nodes.
- The sort Value is the renaming of Elt from the predefined theory STRICT-TOTAL-ORDER, so these values must have defined a _<_ relation.
- The function expand expands a node of the tree and returns a list of partial solutions. Since this operation is defined by the user, it can establish the number of levels that the node is expanded.
- The function isResult? checks if a partial result is a final result.
- The function getBound extracts the upper bound of a node, that must be a Value.

```
fth BB-PROBLEM is
 inc STRICT-TOTAL-ORDER * (sort Elt to Value) .

 sort PartialResult PRList FixData .
 subsort PartialResult < PRList .

 op mtPRList : -> PRList .
 op __ : PRList PRList -> PRList [assoc id: mtPRList] .

 op expand : FixData PartialResult Value -> PRList .
 op isResult? : PartialResult FixData -> Bool .

 op getBound : PartialResult FixData -> Value .
endfth
```

The skeleton needs messages for:

- communicating the fixed data and the new tasks;

```
omod BB-TRANSMITTED-SYNTAX{P :: BB-PROBLEM} is
 pr OID .
 pr TRAVELING-CONTENTS .

 op fixData : P$FixData -> TravelingContents .
 op new-task : P$PartialResult P$Value -> TravelingContents .
```

- reporting the identifier of the worker and the work it has just finished; and

```
 op finished : Oid P$PRList -> TravelingContents .
```

- asking for new work.

```
   op work-needed : Oid -> TravelingContents .
  endom
```

The skeleton uses a priority queue to keep the nodes and their precedence. We define a sort `Pair` that puts together those values, and an operation `insert` that inserts a problem list into the queue.

```
fmod PQUEUE{P :: BB-PROBLEM} is
 vars PQ : PQueue .
 vars V V' : Value .
 var  PL : P$ProblemList .
 vars PR PR' PR'' : P$Problem .
 var  FD : P$FixData .
 var  P : Pair .

 sort PQueue Pair .
 subsort Pair < PQueue .

 op pair : P$Problem Value -> Pair .

 op mtPQueue : -> PQueue .
 op _._ : PQueue PQueue -> PQueue [assoc id: mtPQueue] .

 op insert : P$ProblemList PQueue P$FixData -> PQueue .
 op insert* : Pair PQueue -> PQueue .

 eq insert(mtProblemList, PQ, FD) = PQ .
```

We use the `getBound` function to obtain the value used to sort the queue.

```
 eq insert(PR PL, PQ, FD) =
     insert*(pair(PR, getBound(PR, FD)), insert(PL, PQ, FD)) .

 eq insert*(P, mtPQueue) = P .
 eq insert*(pair(PR', V), pair(PR'', V') . PQ) =
           if (V < V') then pair(PR', V) . pair(PR'', V') . PQ
                       else pair(PR'', V') . insert*(pair(PR', V), PQ)
           fi .
endfm

omod BB-SKELETON{P :: BB-PROBLEM} is
 pr ARCHITECTURE-MSGS .
 pr TRANSMITTED-SYNTAX{P} .
 pr PQUEUE{P} .
 pr MAYBE{FixData}{P} * (sort Maybe{FixData}{P} to DefFixData ) .
 pr LIST{Oid} * (sort List{Oid} to OidList, op nil to mtOidList) .
```

First, we define the class `BBMaster`. It saves information about:

- the `initial` problem;

- the current `result`, that is, the best solution found so far;

- the current upper bound (bestResult);
- the workers that have not been assigned tasks yet; and
- the state, that can be initial or working.

```
    sort MasterState .
    ops initial working : -> MasterState .
```

- The priority queue where tasks are kept.

```
class BBMaster | initial : P$Problem, result : P$Result, bestResult : P$Value,
                 workers : OidList, st : MasterState, queue : PQueue .
```

The BBWorker has the following attributes:

- the identifier of the master;
- the list of unfinished tasks (nextWorks);
- the current upperBound; and
- the fixed data (fixData).

```
class BBWorker | master : Oid, nextWorks : P$ProblemList,
                 upperBound : P$Value, fixData : P$FixData .
```

Given the initial problem, the master expands it and keeps the nodes in the priority queue, changing its state from initial to working. Note that the master uses the function expand, that has been defined to be used by the workers. We decide to use it here in order to obtain enough nodes to deliver them to the workers quickly (the other option would consist in delivering the initial data to one worker, that expands it, and wait for the results).

```
vars PQ PQ' : PQueue .
vars N N' : Nat .
vars V V' : P$Value .
var  OL : OidList .
vars PL PL' : P$PRList .
vars W M : Oid .
vars PR PR' PR'' : P$PartialResult .
var  FD : P$FixData .
var  P : Pair .

crl [start] :
    < M : BBMaster | initial : PR, fixData : FD, st : initial,
                     bestResult : V, queue : mtPQueue, result : PR' >
 => < M : BBMaster | st : working, bestResult : V', queue : PQ,
                     result : PR'' >
 if PL := expand(FD, PR, V) /\
    tern(PR'', PQ, V') := traverse(PL, FD, tern(PR', mtPQueue, V)) .
```

where traverse is a function that traverses a list of partial results, checking for each one if its upper bound is lower than the best current result, and then examines whether it is a final result, updating the current best result, or a promising node, inserting it in the queue.

```
sort Tern .
op tern : P$PartialResult PQueue P$Value -> Tern .

op traverse : P$PRList P$FixData Tern -> Tern .
eq traverse(mtPRList, FD, T) = T .
ceq traverse(PR PL, FD, tern(PR', PQ, V)) =
   if V' < V then
    if isResult?(PR, FD) then traverse(PL, FD, tern(PR, prune(PQ, V'), V'))
    else traverse(PL, FD, tern(PR', insert(PR, PQ, FD), V))
    fi
   else traverse(PL, FD, tern(PR', PQ, V))
   fi
 if V' := getBound(PR, FD) .

op prune : PQueue P$Value -> PQueue .
eq prune(mtPQueue, V) = mtPQueue .
eq prune(PQ . pair(PR, V), V') = if not (V' < V) then PQ . pair(PR, V)
                                 else prune(PQ, V') fi .
```

While the queue is not empty, the master delivers the initial tasks and the fixed data to the workers, that keep them in the corresponding attributes.

```
crl [new-worker] :
    < M : BBMaster | workers : W OL, bestResult : V, st : working,
                     queue : PQ, fixData : FD, numWorks : N >
 => < M : BBMaster | workers : OL, bestResult : V, st : working,
                     queue : update(PQ, N),  fixData : FD, numWorks : N >
    sendInitialTasks(W, PQ, N, V)
    to W : fixData(FD)
 if PQ =/= mtPQueue .

rl [fixData] :
   to W : fixData(FD)
   < W : BBWorker | fixData : null >
=> < W : BBWorker | fixData : FD > .

rl [new-task] :
   to W : new-task(PR, V)
   < W : BBWorker | nextWorks : PL, upperBound : V' >
=> < W : BBWorker | nextWorks : PL PR, upperBound : minimum(V, V') > .

op minimum : P$Value P$Value -> P$Value .
eq minimum(V,V') = if V < V' then V else V' fi .
```

where `sendInitialTasks` and `update` extract tasks from the queue.

```
op sendInitialTasks : Oid PQueue Nat P$Value -> Configuration .
eq sendInitialTasks(W, pair(PR, V') . PQ, s(N), V) = to W : new-task(PR, V)
                                       sendInitialTasks(W, PQ, N, V) .
eq sendInitialTasks(W, PQ, N, V) = none [owise] .

op update : PQueue Nat -> PQueue .
eq update(pair(PR, V') . PQ, s(N)) = update(PQ, N) .
eq update(PQ, N) = PQ [owise] .
```

When the priority queue is not empty and a worker needs a new task, the master delivers it.

```
  rl [work-needed] :
     to M : work-needed(W)
     < M : BBMaster | queue : pair(PR, V') . PQ, st : working, bestResult : V >
  => < M : BBMaster | queue : PQ >
     to W : new-task(PR, V) .
```

While the list of unfinished tasks is not empty, the worker expands the first node and sends the result to the master, or asks for more tasks if the result from `expand` is `mtPRList`. If the node upper bound was higher than the best current one, the worker asks for more work without expanding.

```
  crl [do-work] :
      < W : BBWorker | nextWorks : PR PL, upperBound : V, fixData : FD,
                       master : M >
   => < W : BBWorker | nextWorks : PL >
      if (PL' == mtPRList) then to M : work-needed(W)
                           else to M : finished(W, PL') fi
   if getBound(PR, FD) < V /\
      PL' := expand(FD, PR, V) .

  crl [do-work] :
      < W : BBWorker | nextWorks : PR PL, upperBound : V, fixData : FD,
                       master : M >
   => < W : BBWorker | nextWorks : PL >
      to M : work-needed(W)
   if not (getBound(PR, FD) < V) .
```

When new nodes arrive, the master traverses the list and puts a `work-needed` message in the configuration in order to assign a new task to the worker.

```
  crl [partial-results] :
      to M : finished(W, PL)
      < M : BBMaster | result : PR, bestResult : V, st : working, queue : PQ,
                       fixData : FD >
   => < M : BBMaster | result : PR', bestResult : V', queue : PQ'>
      to M : work-needed(W)
   if tern(PR', PQ', V') := traverse(PL, FD, tern(PR, PQ, V)) .
 endom
```

4.6.1 Traveling salesman instantiation

Since the skeleton hides the features related with the queue, we cannot use directly the functions from Section 4.1.4. First, we define the sorts associated with the fixed data and the nodes required by the theory. The fixed data keeps the cost map, the cities, and the cheapest edge, that will be used to estimate the lower bound of the nodes.

```
 fmod TRAVELER-PROBLEM is
  pr GREEDY-TRAVELER .
  sort FixData .
  ---- Cost, Cities (from 0, so the number of cities is N + 1), Cheapest edge
  op fixData : Map{CityPair, Nat} Nat Nat -> FixData .

  sorts Node NodeList .
  subsort Node < NodeList .

  ---- Path, Current cost
  op node : Path Nat -> Node .

  op mtNodeList : -> NodeList .
  op __ : NodeList NodeList -> NodeList [assoc id: mtNodeList] .
```

We define now the operator getBound. We estimate a lower bound supposing that the edges to all the cities not visited yet have minimum cost.

```
  op getBound : Node FixData -> Nat .
  eq getBound(node(P, N), fixData(G, N', N'')) = N +
                                               sd(s(s(N')), size(P)) * N''  .
```

To expand a node, we must check if it is admissible. We consider admissible a node if the cities are not repeated in the path (unless all cities are visited, when we return to the initial one) and the bound of the node is lower than the current upper bound.

```
  ---- FixData, Node, UpperBound
  op expand : FixData Node Nat -> NodeList .
  ---- FixData, Node, UpperBound, CurrentCity
  op expand : FixData Node Nat Nat -> NodeList .

  eq expand(FD, ND, N) = expand(FD, ND, N, 0) .

  ceq expand(fixData(G, N, N'), ND, UB, N'') = mtNodeList
  if N'' > N .

  ceq expand(fixData(G, N, N'), node(P CT, C), UB, N'') =
           ND' expand(fixData(G, N, N'), node(P CT, C), UB, s(N''))
   if N'' <= N /\
      admissible(P CT, city(N''), N) /\
      ND' := node(P CT city(N''), C + (G [pair(CT, city(N''))])) /\
      getBound(ND', fixData(G, N, N')) < UB .

  eq expand(FD, ND, UB, N) = expand(FD, ND, UB, s(N)) [owise] .

  ---- We can add the new city (i.e., is admissible) if the city is not
  ---- in the path or the path is complete and we are coming back home.
  op admissible : Path City Nat -> Bool .
  eq admissible(P, CT, N) = not in(CT, P) or
                            (size(P) == s(N) and CT == city(0)) .
```

Finally, we define the operator result?, that checks if a node is a final result.

```
  op result? : Node FixData -> Bool .
  eq result?(node(P, N), fixData(G, N', N'')) = result?(P, N') .

  op result? : Path Nat -> Bool .
  eq result?(P, N) = size(P) == s(s(N)) .
 endfm
```

We can now define the Traveler view.

```
view Traveler from BB-PROBLEM to TRAVELER-PROBLEM is
 sort Value to Nat .
 sort PartialResult to Node .
 sort PRList to NodeList .

 op isResult? to result? .
 op mtPRList to mtNodeList .
endv
```

We also show the view from ARCH-COMPLEMENT.

```
fmod TRAVELER-COMPLEMENT is
 pr META-LEVEL .
 pr BB-TRANSMITTED-SYNTAX{Traveler} .
 pr ARCHITECTURE-MSGS .
endfm

view Traveler-Complement from ARCH-COMPLEMENT to TRAVELER-COMPLEMENT is
 op MOD to term upModule('TRAVELER-COMPLEMENT, false) .
endv
```

We use the star architecture. The master is located in the center, while the workers are located in the nodes. We use the greedy algorithm from Section 4.1.4 in the initial configuration to calculate the first upper bound. The initial term for the master in an example with three workers and seven cities (from 0 to 6) is:

```
mod TRAVELER-CENTER is
 pr BB-SKELETON{Traveler} .
 pr STAR-CENTER{Traveler-Complement} .
endm

erew <> < l(ip0, 0) : StarCenter |
                    state : idle,
                    neighbors : empty,
                    defNeighbor : null,
                    port : 60039 >
        < o(l(ip0, 0), 0) : BBMaster |
                    result : node(getCity(R), getCost(R)),
                    bestResult : getCost(R),
                    fixData : fixData(G, 6, cheapestEdge(G)),
                    initial : node(city(0), 0),
                    workers : o(l(ip1, 0), 0) o(l(ip2, 0), 0) o(l(ip3, 0), 0),
                    st : initial,
                    queue : mtPQueue,
                    numWorks : 3 > .
```

where R stands for greedyTravel(city(0), 6, generateCostMatrix(6)) and G stands for generateCostMatrix(6), a random cost matrix for seven cities. The initial term for one of the workers is:

```
mod TRAVELER-NODE is
 pr BB-SKELETON{Traveler} .
 pr STAR-NODE{Traveler-Complement} .
endm

erew <> < l(ip1, 0) : StarNode |
                          state : idle,
                          neighbors : empty,
                          defNeighbor : null,
                          center : ip0,
                          port : 60039 >
        < o(l(ip1, 0), 0) : BBWorker |
                          master : o(l(ip0, 0), 0),
                          nextWorks : mtNodeList,
                          upperBound : 100000,
                          fixData : null > .
```

4.6.2 Graph bipartitioning instantiation

Given a graph $G = (N, E)$, where N is the set of nodes (or vertices) and E is the set of edges, the graph partitioning problem consists in choosing a partition $N = N_1 \cup N_2 \cup \ldots \cup N_p$ such that the number of edges connecting all different pairs N_j and N_k is minimized and the size of each N_i is similar (differs at most in one). We show here the special case where $p = 2$, so we must decide just two partitions.

First we implement a module with the problem (approximately) solved by a greedy algorithm. This module includes `VERTEX`, that defines the sorts `Vertex`, `VertexPair` (for pairs of vertices), and `VertexSet` (for sets of vertices, with functions `size` and `delete`). We define a graph as a (partial) function from pairs of vertices to natural numbers. A solution consists in the two set of vertices and the number of edges between them.

```
fmod GRAPH-PARTITIONING is
 pr VERTEX .
 pr MAP{VertexPair, Nat}  * (sort Map{VertexPair, Nat} to Graph) .

 sort Solution .
 op sol : VertexSet VertexSet Nat -> Solution .
```

In the algorithm, we first choose randomly one vertex for each set.

```
 vars N N' MinEdges MaxEdges : Nat .
 vars V V' : Vertex .
 vars VS VS' VS'' VS''' Candidates NVS : VertexSet .
 var  G : Graph .
 var  S : Solution .

 ---- Remaining Vertices, Cost
 op greedy-gpp : VertexSet Graph -> Solution .
 op greedy-gpp : VertexSet Graph Solution -> Solution .

 eq greedy-gpp((V, V', VS), G) =
    greedy-gpp(VS, G, sol(V, V', if (G [pair(V, V')]) > 0 then 1 else 0 fi)) .
```

Then, we take the vertices not selected yet with minimum number of edges with vertices in the second set. Then, we take the vertex with more edges with vertices in the first set, and we add that vertex to the set. Finally, we interchange the sets. The algorithm finishes when the set of remaining sets is empty.

```
eq greedy-gpp(mtVertexSet, G, S) = S .
ceq greedy-gpp(VS, G, sol(VS', VS'', N)) =
         greedy-gpp(NVS, G, sol(VS'', (V, VS'), N + MinEdges))
 if size(VS) > 0 /\
    MinEdges := getMinIntersection(VS, VS'', G) /\
    Candidates := getVerticesWith(VS, VS'', G, MinEdges) /\
    MaxEdges := getMaxIntersection(Candidates, VS', G) /\
    (V, VS''') := getVerticesWith(Candidates, VS', G, MaxEdges) /\
    NVS := delete(V, VS) .
```

The function `getMinIntersection` just traverses all the vertices not used yet, gets the number of vertices connected with them and keeps the minimum (the `getMaxIntersection` function is symmetrical).

```
op getMinIntersection : VertexSet VertexSet Graph -> Nat .
op getMinIntersection : VertexSet VertexSet Graph Nat -> Nat .

ceq getMinIntersection((V, VS), VS', G) =
                              getMinIntersection(VS, VS', G, N)
 if VS'' := getConnectedVertices(V, VS', G) /\
    N := size(VS'') .
eq getMinIntersection(mtVertexSet, VS, G, N) = N .
ceq getMinIntersection((V, VS), VS', G, N) =
    if (N <= N') then
      getMinIntersection(VS, VS', G, N)
    else
      getMinIntersection(VS, VS', G, N')
    fi
 if VS'' := getConnectedVertices(V, VS', G) /\
    N' := size(VS'') .

op getConnectedVertices : Vertex VertexSet Graph -> VertexSet .
eq getConnectedVertices(V, (V', VS), G) = if ((G [pair(V, V')]) > 0) then V'
                                    else mtVertexSet
                                    fi, getConnectedVertices(V, VS, G) .
eq getConnectedVertices(V, mtVertexSet, G) = mtVertexSet .
```

Finally, the function `getVerticesWith` obtains the set of vertices with the indicated number of edges in the other set.

```
 op getVerticesWith : VertexSet VertexSet Graph Nat -> VertexSet .
 eq getVerticesWith(mtVertexSet, VS, G, N) = mtVertexSet .
 ceq getVerticesWith((V, VS), VS', G, N) =
  if N == N' then V
            else mtVertexSet fi, getVerticesWith(VS, VS', G, N)
  if VS'' := getConnectedVertices(V, VS', G) /\
     N' := size(VS'') .
endfm
```

We can now instantiate the skeleton. First, we write a module defining the sorts and operations that then will be related with those in the theory.

```
fmod GPP-PROBLEM is
 pr GRAPH-PARTITIONING .
```

```
  vars VS VS' VS'' : VertexSet .
  vars N N' N'' N''' : Nat .
  var  G : Graph .
  vars V V' V'' : Vertex .

  sorts Node NodeList .
  subsort Node < NodeList .

  op mtNodeList : -> NodeList .
  op __ : NodeList NodeList -> NodeList [assoc id: mtNodeList] .
```

The nodes contain the set of remaining vertices, the two sets we are creating, and the number of edges between them. We obtain the bound for both unfinished tasks and results by getting the current cost.

```
  ---- Remaining nodes, Set 1, Set 2, Cost
  op node : VertexSet VertexSet VertexSet Nat -> Node .

  ---- Result
  op getBound : Node -> Nat .
  eq getBound(node(VS, VS', VS'', N)) = N .

  ---- Unfinished task
  op getBound : Node Graph -> Nat .
  eq getBound(node(VS, VS', VS'', N), G) = N .
```

The expand function generates two new nodes, one where the current vertex has been added to the first set and another with the vertex added to the second.

```
  op expand : Graph Node Nat -> NodeList .

  ceq expand(G, node((V, VS), VS', VS'', N), N') =
      node(VS, (V, VS'), VS'', N + N''') node(VS, VS', (V, VS''), N + N'')
   if N'' := size(getConnectedVertices(V, VS', G)) /\
      N''' := size(getConnectedVertices(V, VS'', G)) .
  eq expand(G, node(mtVertexSet, VS, VS', N), N') = mtNodeList [owise] .
```

Finally, the `result?` function checks if a node is a valid solution.

```
  op result? : Node -> Bool .
  ceq result?(node(mtVertexSet, VS, VS', N)) = true
   if similarSize(size(VS), size(VS')) .
  eq result?(ND) = false [owise] .

  op similarSize : Nat Nat -> Bool .
  eq similarSize(N, N) = true .
  eq similarSize(N, s(N)) = true .
  eq similarSize(s(N), N) = true .
  eq similarSize(N, N') = false [owise] .
 endfm
```

We can now define the view from BB-PROBLEM, with a mapping similar to the one shown in the traveler instantiation.

```
view Gpp from BB-PROBLEM to GPP-PROBLEM is
 sort Value to Nat .
 sort PartialResult to Node .
 sort PRList to NodeList .
 sort Result to Node .
 sort FixData to Graph .

 op mtProblemList to mtNodeList .
 op mtPRList to mtNodeList .
 op isResult?(PR:PartialResult, FD:FixData) to term result?(PR:Node) .
endv
```

The initial term for the master in an example with two workers and a graph with six vertices (from `0` to `5`) is:

```
mod GPP-CENTER is
 pr BB-SKELETON{Gpp} .
 pr STAR-CENTER{Gpp-Complement} .
endm

erew <> < l(ip0, 0) : StarCenter |
                     state : idle,
                     neighbors : empty,
                     defNeighbor : null,
                     port : 60039 >
        < o(l(ip0, 0), 0) : BBMaster |
                     result : initialResult(5),
                     bestResult : initialCost(5),
                     fixData : generateCostMatrix(5),
                     initial : initialNode(5),
                     workers : o(l(ip1, 0), 0) o(l(ip2, 0), 0),
                     counter : 0,
                     st : initial,
                     queue : mtPQueue,
                     numWorks : 3 > .
```

where `initialResult` and `initialCost` are extracted from the greedy algorithm, the cost graph is generated with `generateCostMatrix`, `initialNode` constructs the initial node with the whole vertex set followed by two empty sets and `0` as initial cost, and `Gpp-Complement` is defined as usual.

```
fmod GPP-COMPLEMENT is
 pr META-LEVEL .
 pr BB-TRANSMITTED-SYNTAX{Gpp} .
 pr ARCHITECTURE-MSGS .
endfm

view Gpp-Complement from ARCH-COMPLEMENT to GPP-COMPLEMENT is
 op MOD to term upModule('GPP-COMPLEMENT, false) .
endv
```

4.7 Pipeline skeleton

A pipeline consists of a list of stages, where each stage applies a different function to the results obtained in the previous stage. The aim is to apply a function $f = f_n \circ f_{n-1} \circ \ldots \circ f_2 \circ f_1$, where f_i is the function applied in the i^{th} stage of the pipeline, to a list of problems. Thus, we will have a master in charge of deliver the list of problems to the first stage and collect the results from the final stage, and n workers that will apply the corresponding function f_i to the values received from the previous worker.

The theory for this skeleton defines the sort `Problem`, that is the sort of the values received in each step (so it must be the returning sort too),[2] and `ProblemList`, defined with the usual notation, used to transmit the initial list of tasks from the master to the worker in the first stage of the pipeline. The operation `step` receives a number i that identifies the stage and a value of sort `Problem` and applies the function f_i to it. This number i must be equal or greater than 1 and equal or less than `numSteps` in order to obtain a correct result. These requirements are expressed by a membership and by declaring `numSteps` as a `NzNat` (a non-zero natural number).

```
fth PPL-PROBLEM is
 inc NAT .

 sorts Problem ProblemList .
 subsort Problem < ProblemList .

 op mtProblemList : -> ProblemList .
 op __ : ProblemList ProblemList -> ProblemList [assoc id: mtProblemList] .

 op numSteps : -> NzNat .
 op step : Nat Problem ~> Problem .

 var N : NzNat .
 var P : Problem .

 cmb step(N, P) : Problem if N <= numSteps .
endfth
```

We only need two messages:

- `tasks` delivers the list of tasks from the master to the first worker of the pipeline.
- `result` delivers a numbered result to the next stage of the pipeline. Notice that in the pipeline skeleton the order matters, so the results must be attended in a ordered way.

```
fmod PPL-TRANSMITTED-SYNTAX{P :: PPL-PROBLEM} is
 pr TRAVELING-CONTENTS .

 op tasks : P$ProblemList -> TravelingContents .
 op result : Nat P$Problem -> TravelingContents .
endfm
```

The module `PPL-SKELETON` implementing the skeleton is parameterized by the `PPL-PROBLEM` theory.

[2] If the different stages return different types, we can use an union type. For example, if $f_1 : A \to B$ and $f_2 : B \to C$, then we would define
```
sorts A B C Problem .
op a :  A -> Problem .
op b :  B -> Problem .
op c :  C -> Problem .
```

```
omod PPL-SKELETON{P :: PPL-PROBLEM} is
 pr TRANSMITTED-SYNTAX{P} .
 pr ARCHITECTURE-MSGS .
 pr OID .
```

The class `PPLMaster` has the following attributes:

- `first` contains the identifier of the worker in the first stage of the pipeline, that must receive the initial list of problems.
- `result` keeps the results from the last stage of the pipeline in a list.
- `masterState` specifies the state of the master. It is in `initial` state at the beginning. When it delivers the initial tasks it reaches the `waiting` state, and waits until it receives all the tasks from the final stage, when it reaches the `finished` state.

  ```
  sort PPLMasterState .
  ops initial waiting finished : -> PPLMasterState .
  ```

- `numTasks` keeps the number of tasks to be developed, in order to know when all the results have been received.
- `tasks` contains the initial list of problems, that will be transmitted to the worker in the first stage of the pipeline.
- `counter` keeps the number of results already received.

```
class PPLMaster | first : Oid, result : P$ProblemList,
                  masterState : PPLMasterState, numTasks : Nat,
                  tasks : P$ProblemList, counter : Nat .

vars N N' : Nat .
vars P P' : P$Problem .
var  PL : P$ProblemList .
vars W O O' : Oid .
```

The first thing the master must do is to send the list of tasks to the worker on the first stage of the pipeline, make the size of this list the number of tasks to be solved, and change its state to `waiting`:

```
rl [start] :
   < O : PPLMaster | tasks : PL, first : O', masterState : initial >
=> < O : PPLMaster | tasks : mtProblemList, masterState : waiting,
                     numTasks : size(PL) >
   to O' : tasks(PL) .
```

Then, the master only waits for results from the last stage of the pipeline. The results must be attended in order; the attribute `counter` is used to know the next result to be taken.

```
rl [result] :
   to O : result(s(N), P)
   < O : PPLMaster | counter : N, result : PL >
=> < O : PPLMaster | counter : s(N), result : PL P > .
```

When all the results have arrived, the master reaches the `finished` state.

```
rl [no-more-work] :
   < O : PPLMaster | counter : N, numTasks : N, masterState : waiting >
=> < O : PPLMaster | masterState : finished > .
```

We distinguish between the first worker (of class `PPLFirstWorker`), that receives the whole list of problems, and all the other workers (of class `PPLWorker`), that receive the problems step by step (that is, one value of sort `Problem` in each step).

The class `PPLWorker` has the following attributes:

- `numStage`, that keeps the stage number of the worker.
- `counter`, that stores the number of tasks that the worker has finished, in order to deliver to the next stage the result numbered.
- `next`, that keeps the identifier of the worker that must do the next stage, or the identifier of the master in the case of the last stage of the pipeline.

```
class PPLWorker | numStage : Nat, counter : Nat, next : Oid .
```

When the next problem arrives to the worker, it does the work and sends the result to the next stage.

```
rl [do-work] :
   to W : result(s(N), P)
   < W : PPLWorker | next : O, counter : N, numStage : N' >
=> < W : PPLWorker | counter : s(N) >
   to O : result(s(N), step(N', P)) .
```

The class `PPLFirstWorker` has also attributes `numStage`, `counter`, and `next` and it has a new attribute `tasks` where the list of unfinished problems is kept. The class is defined as follows:

```
class PPLFirstWorker | numStage : Nat, counter : Nat, next : Oid,
                       tasks : P$ProblemList .
```

The worker in the first stage receives the `tasks` message, and keeps the list of problems in the `tasks` attribute.

```
rl [tasks] :
   to W : tasks(PL)
   < W : PPLFirstWorker | tasks : mtProblemList >
=> < W : PPLFirstWorker | tasks : PL > .
```

While the list of tasks is not empty, the first worker computes another one and sends the result to the next worker, identifying each work with the `counter` attribute.

```
rl [do-work] :
   < W : PPLFirstWorker | next : O, counter : N, numStage : N', tasks : P PL >
=> < W : PPLFirstWorker | counter : s(N), tasks : PL >
   to O : result(s(N), step(N', P)) .
```

4.7.1 Airport instantiation

We show here an example of a high security airport, where the travelers pass through several controls in order to be filtered of pernicious belongings.

We first define the THING module with the description of all the objects that can be carried by someone.

```
fmod THING is
 pr NAT .
 pr STRING .

 sorts Thing Drug Weapon Liquid .
 subsort Drug Weapon Liquid < Thing .

 --- Name and capacity
 op liquid : String Nat -> Liquid .

 ops cocaine lsd : -> Drug .
 ops gun knife : -> Weapon .
 ops book computer apple : -> Thing .
endfm
```

Now we define a view from TRIV to THING in order to instantiate the predefined module LIST for representing the belongings as a list.

```
view Thing from TRIV to THING is
 sort Elt to Thing .
endv

fmod PERSON is
 pr LIST{Thing} * (op nil to mtTL).

 sort Person .
 op person : String List{Thing} -> Person .
endfm

view Person from TRIV to PERSON is
 sort Elt to Person .
endv
```

We can now describe the behavior of our airport security system in a module already prepared to instantiate the skeleton. The function check receives as a parameter the number of the filter that must be applied to the traveler.

```
fmod AIRPORT is
 pr LIST{Person} .

 var  NAME : String .
 var  TL : List{Thing} .
 var  T : Thing .
 var  W : Weapon .
 var  D : Drug .
 var  N : Nat .
```

```
  op check : Nat Person -> Person .
  --- Weapon detector
  eq check(0, person(NAME, TL)) = person(NAME, weaponFilter(TL)) .
  --- Police dogs
  eq check(1, person(NAME, TL)) = person(NAME, drugFilter(TL)) .
  --- New laws
  eq check(2, person(NAME, TL)) = person(NAME, liquidFilter(TL)) .

  ops weaponFilter drugFilter liquidFilter : List{Thing} -> List{Thing} .
```

Each filter checks if the traveler owns any of the items not allowed by the airport, and removes them.

```
  eq weaponFilter(mtTL) = mtTL .
  eq weaponFilter(W TL) = weaponFilter(TL) .
  eq weaponFilter(T TL) = T weaponFilter(TL) [owise] .

  eq drugFilter(mtTL) = mtTL .
  eq drugFilter(D TL) = drugFilter(TL) .
  eq drugFilter(T TL) = T drugFilter(TL) [owise] .

  eq liquidFilter(mtTL) = mtTL .
  eq liquidFilter(liquid(NAME, N) TL) = if (N > 100) then mtTL
                                                      else liquid(NAME, N) fi
                                       liquidFilter(TL) .
  eq liquidFilter(T TL) = T liquidFilter(TL) [owise] .
 endfm
```

Finally the skeleton can be instantiated. Since the master sends information to the first stage, and collects the results from the last one, the most suitable architecture is the ring shown in Section 3.3. We place the master with the `RingLast` object and the workers with the `RingNode` ones. This configuration is the best one because the first connection is made by the `RingLast` object, which allows the transmission of data from the master to the first worker while the architecture is not completely established.

```
 view Airport from PPL-PROBLEM to AIRPORT is
  sort Problem to Person .
  sort ProblemList to List{Person} .

  op mtProblemList to nil .
  op numSteps to term 3 .
  op step to check .
 endv

 fmod AIRPORT-COMPLEMENT is
  pr OID .
  pr META-LEVEL .
  pr PPL-TRANSMITTED-SYNTAX{Airport} .
  pr ARCHITECTURE-MSGS .
 endfm

 view Airport-Complement from ARCH-COMPLEMENT to AIRPORT-COMPLEMENT is
  op MOD to term upModule('AIRPORT-COMPLEMENT, false) .
 endv
```

The initial configuration for the location with the master is:

```
mod AIRPORT-RLAST is
 pr PIPELINE-SKELETON{Airport} .
 pr RING-LAST{Airport-Complement} .

 op persons : -> List{Person} .
 eq persons = person("A", lsd knife computer)
              person("B", liquid("cologne", 110) liquid("toothpaste", 50))
              person("C", liquid("deodorant", 30) book cocaine)
              person("D", apple computer book) .
endm

erew <> < l(ip0, 0) : RingLast |
                     state : idle,
                     neighbors : empty,
                     defNeighbor : null,
                     port : 60039,
                     nextIP : ip1,
                     nextPort : 60044 >
        < o(l(ip0, 0), 0) : PPLMaster |
                     result : nil,
                     masterState : initial,
                     tasks : persons,
                     numTasks : 0,
                     counter : 0,
                     first : o(l(ip1, 0), 0) > .
```

The initial configuration for the location with the first worker is:

```
mod AIRPORT-RNODE is
 pr PIPELINE-SKELETON{Airport} .
 pr RING-NODE{Airport-Complement} .
endm

erew <> < l(ip1, 0) : RingNode |
                     state : idle,
                     neighbors : empty,
                     defNeighbor : null,
                     port : 60044,
                     nextIP : ip2,
                     nextPort : 60041 >
        < o(l(ip1, 0), 0) : PPLFirstWorker |
                     counter : 0,
                     numStage : 1,
                     next : o(l(ip2, 0), 0),
                     tasks : nil > .
```

The initial configurations for the rest of the locations is very similar, and looks as follows:

```
mod AIRPORT-RNODE is
 pr PIPELINE-SKELETON{Airport} .
 pr RING-NODE{Airport-Complement} .
endm
```

```
erew <> < l(ip2, 0) : RingNode |
                    state : idle,
                    neighbors : empty,
                    defNeighbor : null,
                    port : 60041,
                    nextIP : ip3,
                    nextPort : 60042 >
        < o(l(ip2, 0), 0) : PPLWorker |
                    counter : 0,
                    numStage : 2,
                    next : o(l(ip3, 0), 0) > .
```

Chapter 5

Mobile Maude

Mobile Maude is a mobile agent language extending Maude and supporting mobile computation. Mobile Maude uses reflection to obtain a simple and general declarative mobile language design and makes possible strong assurances about mobile agent behavior. The formal semantics of Mobile Maude is given by an executable rewrite theory in Maude, in which mobile agent systems can be simulated.

A short version of the work described in this chapter has been published in the *Proceedings of the Sixth International Workshop on Rewriting Logic and its Applications, WRLA 06* as the paper [29], and a longer version as the Chapter 16 of the Maude book [18].

5.1 Mobile Maude main features

Mobile Maude's key characteristics include:

- reflection as a way of endowing mobile objects with "higher-order" capabilities;
- object-orientation and asynchronous message passing; and
- a simple semantics without any loss in the expressive power of application code.

Mobile Maude was first introduced in [28], where the authors presented a 'simulator' of Mobile Maude, an executable Maude specification on top of Maude 1.0.5, in which the system code was written entirely in Maude, and thus locations and processes were encoded as Maude terms. In the same paper it was also sketched a development plan including two further development efforts: a first step in which a single-host executable system would be implemented, and a second implementation effort focussing on true distributed execution.

The release of Maude 2.0 allowed them to take the first step. This implementation effort was completed in a very short time, using the built-in object system for object/message fairness, just by simplifying and extending the previous specification. This new version was developed by Francisco Durán and Alberto Verdejo, and was used in several examples, one of which was reported in [30].

The built-in string handling and internet socket module available in Maude 2.2 has allowed us to build a truly distributed implementation, thus advancing the second development effort. The Maude 2.2 `SOCKET` module support non-blocking client and server TCP sockets (see Section 2.7). In this implementation effort, a Mobile Maude server runs on top of a Maude interpreter and performs the following tasks:

1. keeps track of the current locations of mobile objects created on a host;
2. handles change of location messages;
3. reroutes messages to mobile objects; and
4. runs the code of mobile objects by invoking the metalevel.

We explain below the design of processes and mobile objects and their rewriting semantics, based on a formal specification of Mobile Maude written in Maude. How to formally analyze Mobile Maude applications will be shown in Section 6.7.

5.2 Processes and mobile objects

The two key notions of Mobile Maude are *processes* and *mobile objects*. A *distributed configuration* is made up of located configurations, each of them executed in a Maude process (connected by an architecture as those shown in Chapter 3). Such processes can therefore be seen as *located* computational environments *inside which* mobile objects can reside, execute, and send and receive messages to and from other mobile objects located in different processes. We assume that each located configuration has exactly one *home*, of class `Home`, which keeps information about the mobile objects in such a configuration and the whereabouts of the mobile objects created in it, which may have moved to other processes. The *names* of homes range over the sort `Hid`, and have the form `h(L)` with `L` the identifier of the location where the object resides. We assume uniqueness of home names in a distributed configuration. The syntax for home identifiers and the corresponding `getLoc` function (see Section 3.1) are defined in the `HID` module.

```
fmod HID is
 pr LOC .
 sorts Hid .
 subsort Hid < Oid .
 op h : Loc -> Hid .                 *** Home Oid

 op getLoc : Hid -> Loc .
 eq getLoc(h(L:Loc)) = L:Loc .
endfm
```

Mobile objects are modeled as distributed objects in the class `MobileObject`. The names of mobile objects range over the sort `Mid` and have the form `o(L, N)`, with `L` the name of the location in which it was created and `N` a number used to distinguish objects created in the same location.

```
fmod MID is
 pr LOC .
 sorts Mid .
 subsort Mid < Oid .
 op o : Loc Nat -> Mid .             *** Mobile-Object Oid
endfm
```

Once we have defined the syntax of mobile objects and homes, we can define the class `Home` as follows:

```
view Mid from TRIV to MID is
 sort Elt to Mid .
endv

fmod PAIR{X :: TRIV, Y :: TRIV} is
 sort Pair{X,Y} .

 op (_,_) : X$Elt Y$Elt -> Pair{X,Y} .
 op p1_ : Pair{X,Y} -> X$Elt .
 op p2_ : Pair{X,Y} -> Y$Elt .
 eq p1 (V1:X$Elt, V2:Y$Elt) = V1:X$Elt .
 eq p2 (V1:X$Elt, V2:Y$Elt) = V2:Y$Elt .
endfm

view Pair{Loc,Nat} from TRIV to PAIR{Loc, Nat} is
 sort Elt to Pair{Loc, Nat} .
endv

omod HOME is
 pr SET{Mid} .
 pr MAP{Nat, Pair{Loc,Nat}} .
 pr HID .

 class Home | cnt : Nat,                          *** counter to generate names
              guests : Set{Mid},                  *** objects in the location
              forward : Map{Nat, Pair{Loc,Nat}} . *** forwarding information
endom
```

The home's cnt attribute stores a counter to generate unique names for new mobile objects. The home of each process keeps information about the mobile objects currently in it in the guests attribute.

Since mobile objects may move from one process to another, reaching them by messages is nontrivial. The solution adopted in Mobile Maude is that, when a message's addressee is not in the current process, the message is forwarded to the addressee's home. Each home stores forwarding information about the whereabouts of its children in its forward attribute, a partial function in Map{Nat, Pair{Loc,Nat}} that maps each child (identified by its number) to a pair consisting of the name of the location in which the object currently resides, and the number of "hops" to different processes that the mobile object has taken so far. The number of hops is important in disambiguating situations when old messages (containing old location information) arrive after newer ones containing the current location. The most recent location is that associated with the largest number of hops. Whenever a mobile object moves to a new process, the object's parent process is always notified. Note that this mechanism does not guarantee message delivery in the case that objects move more rapidly than messages.

Mobile objects carry their own internal state and code (an object-oriented module) with them, can move from one process to another, and can communicate with each other by asynchronous message passing. Figure 5.1 shows several mobile objects in two processes,[1] with (mobile) object o(l(IP, 0), 1) moving from the process with home h(l(IP, 0)) to the process of home h(l(IP', 0)), and with object o(l(IP, 0), 0) sending a message to o(l(IP', 0), 0).

[1] Although the router objects has been omitted in the figure, remember that Mobile Maude is executed on top of one of the architectures shown in Chapter 3.

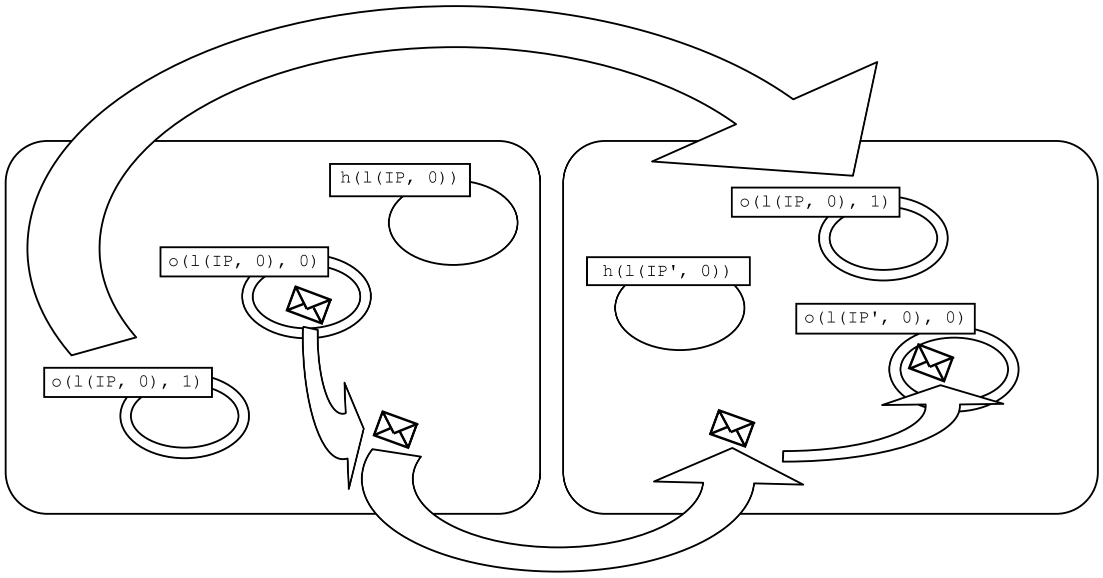

Figure 5.1: Object and message mobility

Mobile objects are specified as objects of the class `MobileObject`:

```
omod MOBILE-OBJECT is
 pr META-LEVEL .
 pr MID .

 class MobileObject |
    mod : Module,           *** rewrite rules of the mobile object
    s : Term,               *** current state
    gas : Nat,              *** bound on resources
    hops : Nat .            *** number of hops
endom
```

The sorts `Module` and `Term`, associated with the attributes `mod` and `s`, respectively, are sorts in the module `META-LEVEL` (see Section 2.5). The value of a mobile object's `mod` attribute is the metarepresentation of an object-oriented module. The mobile object's *state* `s` must be the metarepresentation of a pair of configurations meaningful for the module in `mod` and having the form `C & C'`, where `C` is a configuration of objects and messages representing unprocessed incoming messages and inter-inner-objects messages, and `C'` is a multiset of messages representing the *outgoing* messages tray. One of the objects in `C` is supposed to have the same identifier as the mobile object it is in. We sometimes refer to this object as the *main* one, which in most cases will be the only one. Therefore, we can think of a mobile object as a *wrapper* that encapsulates the state and code of its inner object and mediates its communication with other objects. For this reason, Figure 5.1 depicts mobile objects by two concentric circles, with the inner object and its incoming and outgoing messages contained in the inner circle.

A `MobileObject` includes the attribute `hops`, which stores the number of "hops" a mobile object has performed while moving from one process to another. As we have seen, this information is used to disambiguate the arrival of messages with obsolete information. To guarantee that all mobile objects eventually have some activity, and to impose a bound on the resources they can consume, they have a `gas` attribute, that will be reduced as the mobile object evolves.

5.3 Mobile Maude interface

Mobile Maude *system code* is specified by a relatively small number of rules for homes, mobile objects, mobility, and message passing. Such rules work in an *application-independent* way. Application code, on the other hand, can be written as Maude object-oriented modules with great freedom, except for being aware that, as explained in Section 5.2, the top level of the internal state of a mobile object has to be a pair of configurations, with the first, called the *inner configuration*, containing the inner object(s) and incoming messages, and the second component, called *outgoing tray*, containing outgoing messages. Such a pair is built with the constructor `_&_`

```
fmod MOBILE-OBJECT-INTERFACE is
 pr META-LEVEL .
 inc MID .

 sort MobObjState .
 op _&_ : Configuration Configuration -> MobObjState [ctor] .
```

The messages sent by a mobile object may in fact be understood as commands that the main object—or one of the other objects—in the internal state of a mobile object gives to its wrapper object. Thus, an object may

1. request to move from its current location to a given location `L` with the `go(L)` message;
2. request going to the location in which the mobile object `O` resides, which is possibly `L`, with the message `go-find(O, L)`;
3. request creating a new mobile object with module `Mod`, initial state `Conf`, and temporal identifier of the main object in such a configuration `O`, with the message `newo(Mod, Conf, O)`;
4. send a message with contents `C` to the object `O` with the message `to O : C`; and
5. request the destruction of the mobile object it resides into with the message `kill`.

```
 sort Contents .

 msg go : Loc -> Msg .
 msg go-find : Mid Loc -> Msg .
 msg newo : Module Configuration Oid -> Msg .
 msg to_:_ : Mid Contents -> Msg .
 msg kill : -> Msg .
```

Note that messages being sent to other mobile objects must be of the form `to_:_`, with the addressee of the message as first argument and a term of sort `Contents` as second argument.[2] The definition of the `Contents` sort is left to each particular application, which

[2]Notice that this message has the same syntax that the operator to interchange messages between routers in the architecture, but different arguments (`Contents` in Mobile Maude and `TravelingContetns` in the architectures).

in fact gives the user the freedom to define any kind of message, an example is shown in page 125.

The newo message takes a module (a term of sort Module metarepresenting a module), a term of sort Configuration (which will be the initial configuration in the belly of the mobile object to be created, so it should make sense in the module given as first argument), and the provisional identifier of the main object in the configuration given as second argument. As we shall see in Section 5.5.4, the first action accomplished by a mobile object when it detects the newo command is creating a new mobile object with the metarepresentation of the configuration given as second argument to the newo message, and then sending a start-up message to the main object with its new name, so that it coincides with the name of the mobile object it is in. Let us recall that the name of a mobile object depends on the home object in its process, and on the number of mobile objects already created in it. Therefore, such a name cannot be known when the creation is requested. Thus, the main object in the configuration will be created with a provisional identifier—usually tmp-id—that will be changed by its mobile object once it is created.

```
  op tmp-id : -> Mid [ctor] .
  op start-up : Mid -> Contents [ctor] .
 endfm
```

The MOBILE-OBJECT-INTERFACE module is assumed to be imported by the user in all Mobile Maude applications.

5.4 Mobile Maude's syntax

Once the messages shown above are "pulled out" of the outgoing message tray, they must be placed in the located configuration at the level of mobile objects. Some of these messages will travel through the different locations, while others will be used in the same location where they are pulled out. The messages that will travel are defined of sort TravelingContents, and will be encapsulated in the to_:_ message from the architecture (see Section 3.1). We define the syntax of these messages in the following module:

```
 fmod MOBILE-MAUDE-TRANSMITED-SYNTAX is
  pr MID .
  pr MOBILE-OBJECT .
  pr PAIR{Loc, Nat} .
  pr SET{Mid} .
  pr TRAVELING-CONTENTS .
  pr MAYBE{Nat} * (op maybe to null) .
```

When a go or go-find message is pulled out, the object is encapsulated in the message while the location will be used to send the object to the proper home object.

```
  op go : Object -> TravelingContents [frozen] .
  op go-find : Mid Maybe{Nat} Object -> TravelingContents [frozen] .
```

Notice that the operators are declared frozen, so the mobile object inside cannot evolve while they are in transit.

When an object arrives to a new location, it sends a message to its home in order to keep the forwarding information updated. The object_@_ message notifies the number of the object as first parameter and the location and number of hops as second one.

```
  op object_@_ : Nat Pair{Loc, Nat} -> TravelingContents .
```

Messages between objects may travel too. They contain the addressee identifier, the number of hops of the forwarding information that they have received, and the information the object wants to communicate (where the `Term` is a metarepresented value of sort `Contents`).

```
  op to_hops_{_} : Mid Maybe{Nat} Term -> TravelingContents .
```

Finally, when an object is destroyed, the information is sent to its home.

```
  op dead{_} : Nat -> TravelingContents .
 endfm
```

All the Mobile Maude applications can use the same view, from a module that contains home objects (needed because they will be the addressees of the messages), the syntax of Mobile Maude, and the messages from the architecture.

```
 fmod MM-COMPLEMENT is
  pr HID .
  pr ARCHITECTURE-MSGS .
  pr MOBILE-MAUDE-TRANSMITED-SYNTAX
 endfm

 view MM-Complement from ARCH-COMPLEMENT to MM-COMPLEMENT is
  op MOD to term upModule('MM-COMPLEMENT, false) .
 endv
```

In addition to the syntax of messages that will travel between locations, we define the syntax of messages that will not be transmitted. The set of all these messages defines the syntax of Mobile Maude.

```
 fmod MOBILE-MAUDE-SYNTAX is
  pr MOBILE-MAUDE-TRANSMITED-SYNTAX .

  *** Inter-object message inside a process configuration.
  msg to_{_} : Mid Term -> Msg .

  *** Mobile object creation.
  msg newo : Module Term Term -> Msg .
 endfm
```

5.5 Mobile Maude's rewriting semantics

The semantics of Mobile Maude is specified by an object-oriented rewrite theory containing the definitions of the classes `Home` and `MobileObject` and rewrite rules that describe the behavior of the different primitives: object mobility, message passing, and object creation and destruction. This specification is the *system code* of Mobile Maude, which works in an application-independent way as a prototype on which to execute Mobile Maude applications. The rules discussed in the next subsections specify the way in which the different Mobile Maude commands are handled.

```
omod MOBILE-MAUDE-SEMANTICS is
 pr MOBILE-MAUDE-SYNTAX .
 pr HOME .
 pr ARCHITECTURE-MSGS .
```

5.5.1 Letting mobile objects do something

To allow the mobile objects to evolve, and therefore to allow the application code, which "lives" inside the mobile objects, to invoke such commands, the state of the mobile objects must be rewritten. In the `do-something` rule below, the internal state of a mobile object is rewritten using the rules of the module in its `mod` attribute.

As a fairness condition, or, more concretely, to make sure that no mobile object consumes all the resources—to avoid, for example, that when rewriting the state of a mobile object we get into an infinite computation—and to try to balance such consumption, we establish a bound on the number of rewrites for each of the mobile objects. Such a bound is given in their `gas` attribute. Each time the `do-something` rule is applied, the mobile object's gas value is decremented. Note that the `gas` attribute gives the number of rewrites a mobile object can perform. If no rewriting step can be taken, the `do-something` rule cannot be applied.

```
 vars O O' : Mid .
 vars T T' T'' T''' T'''' : Term .
 var  MOD : Module .
 vars N N' N'' : Nat .
 var  TS : TermSet .
 var  RST? : ResultTriple? .

 sort TermSet .
 subsort Term < TermSet .
 op emptyTS : -> TermSet .
 op _._ : TermSet TermSet -> TermSet [assoc comm id: emptyTS] .
 eq T . T = T .

 crl [do-something] :
     < O : MobileObject | mod : MOD, s : T, gas : s(N) >
  => < O : MobileObject | s : T', gas : N >
  if T' . TS :=  possibleSteps(MOD, T) /\
     T' =/= T .
```

Instead of using the deterministic functions `metaRewrite` or `metaFrewrite`, we use the `metaSearch` function in the equations for the operation `possibleSteps` to obtain the set of terms that can be reached in one rewriting step. From all these possible rewrites one is chosen (modulo commutativity) in the condition of the `do-something` rule; this allows us to explore all possible executions as we shall discuss in Section 6.7.

```
 op possibleSteps : Module Term Nat -> TermSet .

 var Ty : Type .
 var Sb : Substitution .

 ceq possibleSteps(MOD, T, N) = T' . possibleSteps(MOD, T, N + 1)
  if {T', Ty, Sb} := metaSearch(MOD, T, 'M:MobObjState, nil, '+, 1, N) .
 eq possibleSteps(MOD, T, N) = emptyTS [owise]
```

5.5.2 Object communication

There are three kinds of communication between objects. Objects inside the same mobile object can communicate with each other by means of messages with any format, and such communication may be synchronous or asynchronous. Objects in different mobile objects may communicate when such mobile objects are in the same process and when they are in different processes; in these cases, the actual kind of communication is transparent to the mobile objects, but such communication must be asynchronous and by means of messages the form `to O : C`. That is, the minimum information needed to dispatch a message is the receiver's identity; if the sender wants to communicate its identifier, it has to include it in the message contents. If the addressee is an object in a different mobile object, then the message must be put by the sender object in the second component of its state (the outgoing messages tray). The system code will then send the message to the addressee object.

An important issue when managing messages is that the rewriting rules and state of mobile objects are metarepresented, that is, the system code of Mobile Maude is at the metalevel of the application code. Therefore, before dealing with such messages, they must be moved up, or, as we say, they must be *pulled out*. The internal state of a mobile object will have the form `'_&_[T, T']`, with `T` and `T'` the terms metarepresenting, respectively, the inner configuration and the outgoing messages. In the case of messages of the form `to_:_` we will have a term of the form `'to_:_[T1, T2]`, which may be alone or with more messages in the outgoing tray. Since we must leave the tray in a valid state we need to include the following three rules.[3]

```
op downMid : Term -> Mid .
eq downMid(T) = downTerm(T, o(l("null", 0), 0)) .

rl [message-out-to] :
   < O : MobileObject | s : '_&_[T, 'to_:_[T', T'']] >
=> < O : MobileObject | s : '_&_[T, 'none.Configuration] >
   to downMid(T') { T'' } .

rl [message-out-to] :
   < O : MobileObject | s : '_&_[T, '__['to_:_[T', T''], T''']] >
=> < O : MobileObject | s : '_&_[T, T'''] >
   to downMid(T') { T'' } .

var  TL : TermList .

crl [message-out-to] :
    < O : MobileObject | s : '_&_[T, '__['to_:_[T', T''], T''', TL]] >
 => < O : MobileObject | s : '_&_[T, '__[T''', TL]] >
    to downMid(T') { T'' }
 if TL =/= empty .
```

Notice that, although the contents of messages are left at the metalevel, i.e., as found, the identifier of the addressee object is moved down to the object level, so that the message can be correctly delivered. The rules pull out a message `to` O : C, which is metarepresented

[3]Although in general two cases are enough to deal with associative lists (one element and more than one element), at the metalevel, since the engine is giving the list in flattened form and expects it in flattened form, we must make sure that when we have more than one element the top operator is __.

as `'to_:_[`$\overline{O}$`, `$\overline{C}$`]`, into a message `to` O `{` $\overline{C}$ `}`. We will find similar pull-out rules for each of the commands.

Once the message is out of the mobile object, it will be appropriately delivered.

```
var  OS : Set{Mid} .
vars L L' L'' L''' : Loc .
var  F : Map{Nat, Pair{Loc, Nat}} .

crl [msg-send] :
    to o(L, N) { T }
    < h(L) : Home  | guests : OS, forward : F >
 => < h(L) : Home  | >
    to h(p1(F[N])) : to o(L, N) hops p2(F[N]) { T }
 if not o(L, N) in OS /\
    F[N] =/= undefined /\
    p1(F[N]) =/= L .

crl [msg-send] :
    to o(L, N) { T }
    < h(L') : Home  | guests : OS >
 => < h(L') : Home  | >
    to h(L) : to o(L, N) hops null { T }
 if L =/= L' /\
    not o(L, N) in OS .
```

Note that the conditions of the rules make sure not only that the object is not in the current process (`not o(L, N) in OS`), but also that the forwarding info does not point to the location itself (`p1(F[N]) =/= L`), which would mean that the mobile object has not arrived to its destination yet. If the location in which the message is generated is the parent location of the mobile object the message is addressed to (first rule), then the message is forwarded to the location indicated by the forwarding info with the corresponding number of hops; otherwise, the message is forwarded to the parent location with the number of hops set to `null` (second rule). We will see below how the hops information is used in the `msg-arrive-to-loc` rules to avoid unnecessary forwarding of messages when the destination object is in transit.

The arrival of an inter-object message to a location is handled by the following four rules. We explain the case handled by each of the rules separately.

If the addressee object is at the location, then the message is just left at the location so the object can get it.

```
var  H : Maybe{Nat} .

rl [msg-arrive-to-loc] :
   to h(L') : to o(L, N) hops H { T }
   < h(L') : Home  | guests : (o(L, N), OS) >
=> < h(L') : Home  |  >
   to o(L, N) { T } .
```

If the object is not at the location and the number of hops is `null`, then the message is being sent to the mobile object's parent location. If the forwarding information is pointing to its home location, then the object is in transit, and the forwarding information has not been updated with its new location, and therefore the message is not handled; otherwise, the message is sent to the location indicated by the forwarding information with the corresponding number of hops.

```
crl [msg-arrive-to-loc] :
    to h(L) : to o(L, N) hops null { T }
    < h(L) : Home | guests : OS, forward : F >
 => < h(L) : Home | >
    to h(p1(F[N])) : to o(L, N) hops p2(F[N]) { T }
 if not o(L, N) in OS /\ p1(F[N]) =/= L .
```

If the object is not at the location and the location is not its home location, then the message is forwarded back to the parent location with the same hops number. Note that, since it is not its home location, the number of hops is not `null`, that is, it is a natural number. Note also that, since the forwarding information is updated once the object has arrived to a location, it cannot be the case that the message has arrived before the object. If the object to which the message is addressed is not at the location registered in the forwarding information, it is because the object has already left the location and the message must be returned to its home location.

```
crl [msg-arrive-to-loc] :
    to h(L') : to o(L, N) hops N' { T }
    < h(L') : Home | guests : OS >
 => < h(L') : Home | >
    to h(L) : to o(L, N) hops N' { T }
 if not o(L, N) in OS /\ L =/= L' .
```

Finally, if the message is being returned from a location to which the message was forwarded from its home location because the object already left it, then the message will be forwarded again by its home location only if its forwarding information has been updated since the message was forwarded the first time, that is, if the number of hops in the message is smaller than the number of hops in the forwarding information in its home location. Note that we do not check whether the forwarding information points to the parent location itself anymore, since in this case the hops would have been appropriately incremented.

```
crl [msg-arrive-to-loc] :
    to h(L) : to o(L, N) hops N' { T }
    < h(L) : Home  | guests : OS, forward : F >
 => < h(L) : Home  | >
    to h(p1(F[N])) : to o(L, N) hops p2(F[N]) { T }
 if not o(L, N) in OS /\ N' < p2(F[N]) .
```

Once the message reaches its addressee object, the message must be inserted in—*pushed into*—the internal state of such a mobile object. To make sure that the mobile object will remain in a valid state, we check that the metarepresentation of the corresponding message is a valid message in the module of the object. We can assume that, since the previous state was a valid one, adding a valid message will result in a new valid state.

```
rl [msg-in] :
   to O { T }
   < O : MobileObject | mod : MOD, s : '_&_[T', T''] >
=> if sortLeq(MOD, leastSort(MOD, 'to_:_[upTerm(O), T]), 'Msg)
   then < O : MobileObject | s : '_&_['__['to_:_[upTerm(O), T], T'], T''] >
   else < O : MobileObject | s : '_&_[T', T''] >
   fi .
```

5.5.3 Object mobility

We explain in this section the rules that govern object mobility. Such mobility is initiated by the mobile object's inner object, which puts the `go` or `go-find` messages in the second component (i.e., as an outgoing message) of the state. The rules for both cases are quite similar; the main difference is that a `go-find` message tries to reach a particular object that can be itself on movement; that is, we may reach the tentative location and not find the object there, in which case we must go on looking for it in a different location.

The `go` message

When a mobile object wants to move to another process it puts in its outgoing messages tray a `go(L)` message, where `L` is the target location. When a mobile object has an outgoing `go` message, a new *inter-mobile-objects* `go` message is sent, with the mobile object as its argument, after removing the outgoing message. Since the `go` message is declared to be frozen, the mobile object is inactive while on the move.

If there is a `go` message in the outgoing message tray, we observe that the state of the corresponding mobile object has the form `'_&_[T, 'go[T']]`, where the term `T'` metarepresents the name of the location where the object wants to go. Notice that in this case it must be the only message in the tray, it is assumed that any other message has already been handled. The rule `message-out-go` indicates how such a name is decoded by the `downLoc` function, and shows in its righthand side the mobile object ready to go—which is indicated by being enclosed inside a `go` operator.

```
op downLoc : Term -> Loc .
eq downLoc(T) = downTerm(T, l("null", 0)) .

rl [message-out-go] :
   < h(L) : Home | guests : (O, OS) >
   < O : MobileObject | s : '_&_[T, 'go[T']] >
=> < h(L) : Home | guests : OS >
   to h(downLoc(T')) : go(< O : MobileObject |
                                          s : '_&_[T, 'none.Configuration] >) .
```

If the message and the addressee are in different locations, then this message will be sent through the appropriate socket by the router of the location. When the message reaches the destination location, the home is informed, so it can update its forwarding information; if the object has reached its home location, such information is directly updated.

When a `go` message reaches the location it is addressed to, the mobile object that it carries as an argument is put into the configuration. Depending on whether the location is the home location of such a mobile object or not, the forwarding information is updated or a message `object_@_` is sent to its home location so that the home in it can update its forwarding information.

```
rl [arrive-loc] :
   to h(L) : go(< o(L', N) : MobileObject | hops : N' >)
   < h(L) : Home | guests : OS, forward : F >
=> < o(L', N) : MobileObject | hops : N' + 1 >
   if L == L'
   then < h(L) : Home | guests : (o(L', N), OS),
                        forward : insert(N, (L, N' + 1), F) >
   else < h(L) : Home | guests : (o(L', N), OS) >
        to h(L') : object N @ (L, N' + 1)
   fi .
```

The following rule specifies the update of a mobile object's forwarding information in the home upon the reception of an `object_@_` message. Note that, since the message to update the forwarding information is sent when the object arrives to its destination location, the forwarding information is not valid during the transit of the mobile objects. However, thanks to the guests lists we still have enough information to guide messages appropriately. Notice also that the forwarding information about a mobile object may be `undefined` upon the reception of an `object_@_` message if the corresponding mobile object was destroyed and the message communicating its destruction arrives before a message communicating a previous move.

```
  rl [forwarding-update] :
     to h(L) : object N @ (L', N')
     < h(L) : Home  | forward : F >
  => if F[N] == undefined
     then < h(L) : Home | >
     else if p2(F[N]) < N'
          then < h(L) : Home  | forward : insert(N, (L', N'), F) >
          else < h(L) : Home  |  >
          fi
     fi .
```

The `go-find` message

In the `go` message, the mobile object indicates the location it wants to go to. However, sometimes, a mobile object wants to reach another object, but it only knows the identifier of the object it wants to catch up with, not the location it is at. In this case, the `go-find` message can be used, which takes as arguments the identifier of the mobile object to be reached, and the identifier of a tentative location, where it may be.

The rules for the `go-find` messages are very similar to those for the `go` messages just described. However, in this case we do not only want to reach a location, but also to find a mobile object, which may move from one place to another. Although the message includes a tentative location for the object, such information may be incorrect, or obsolete.

When a mobile object has a `go-find` message in its state it is pulled out with the following rule.

```
  rl [message-out-go-find] :
     < h(L) : Home | guests : (O, OS) >
     < O : MobileObject | s : '_&_[T, 'go-find[T', T'']] >
  => < h(L) : Home | guests : OS >
     to h(downLoc(T'')) : go-find(downMid(T'), null,
                     < O : MobileObject | s : '_&_[T, 'none.Configuration] >) .
```

When a `go-find` message reaches the tentative location it was addressed to, depending on whether the object the message is trying to find is at such a location or not, the mobile object will be put into the configuration or forwarded. As in the `arrive-loc` rule shown above, the forwarding information is then updated. In addition, if the message requires to be forwarded, this will be done towards the location the mobile object is at according to the forwarding information in its home location, or to such a home location depending on whether it is at its home location or not.

```
rl [arrive-find-loc] :
   *** the object has been reached in the traveling object's home location
   to h(L') : go-find(o(L, N), H, < o(L', N') : MobileObject | hops : N'' >)
   < h(L') : Home | guests : (o(L, N), OS), forward : F >
=> < h(L') : Home | guests : (o(L, N), o(L', N'), OS),
                    forward : insert(N', (L', N'' + 1), F) >
   < o(L', N') : MobileObject | hops : N'' + 1 > .

crl [arrive-find-loc] :
    *** the object has been reached in the tentative location,
    *** and this is not its home location
    to h(L') : go-find(o(L, N), H, < o(L'', N') : MobileObject | hops : N'' >)
    < h(L') : Home | guests : (o(L, N), OS) >
 => < h(L') : Home | guests : (o(L, N), o(L'', N'), OS) >
    < o(L'', N') : MobileObject | hops : N'' + 1 >
    to h(L'') : object N @ (L', N'' + 1)
 if L' =/= L'' .

crl [arrive-find-loc] :
    *** The message is redirected for the first time by using
    *** the forward attribute
    to h(L) : go-find(o(L, N), null, < o(L'', N') : MobileObject | >)
    < h(L) : Home | guests : OS, forward : F >
 => < h(L) : Home | >
    to h(p1(F[N])) : go-find(o(L, N), p2(F[N]),
                     < o(L'', N') : MobileObject | >)
 if not o(L, N) in OS /\ F[N] =/= undefined .

crl [arrive-find-loc] :
    *** The message is redirected by using the new information of
    *** the forward attribute
    to h(L) : go-find(o(L, N), N'', < o(L'', N') : MobileObject | >)
    < h(L) : Home | guests : OS, forward : F >
 => < h(L) : Home | >
    to h(p1(F[N])) : go-find(o(L, N), p2(F[N]),
                     < o(L'', N') : MobileObject | >)
 if not o(L, N) in OS /\ p2(F[N]) > N'' .

crl [arrive-find-loc] :
    *** Redirection to the home location
    to h(L') : go-find(o(L, N), H, < o(L'', N') : MobileObject | >)
    < h(L') : Home | guests : OS >
 => < h(L') : Home | >
    to h(L) : go-find(o(L, N), null, < o(L'', N') : MobileObject | >)
 if not o(L, N) in OS /\ L =/= L' .
```

5.5.4 The creation of mobile objects

When an object (in the inner configuration of a mobile object), as part of the application code, wants to create a new mobile object, it sends a newo message to the system (by putting it in the second component of its state, the outgoing tray). The first action accomplished by the system when it detects the `newo(C, M, O)` message is to create a new mobile object with the configuration `C` as its state and the module `M` as its code, and then to send a `start-up` message to the object `O` with its new name. The application code will be in charge of attending the `start-up` message as shown in page 133.

First, as for the other Mobile Maude commands, we need to provide rules for pulling out `newo` commands. As for the `to_:_` message in Section 5.5.2, we need three rules to cover the different cases.

```
op errorModule : -> [Module] [ctor] .

op downModule : Term -> Module .
eq downModule(T) = downTerm(T, errorModule) .

rl [message-out-newo] :
   < O : MobileObject | s : '_&_[T, 'newo[T', T'', T''']] >
=> < O : MobileObject | s : '_&_[T, 'none.Configuration] >
   newo(downModule(T'), T'', T''') .

rl [message-out-newo] :
   < O : MobileObject | s : '_&_[T, '__['newo[T', T'', T'''], T'''']] >
=> < O : MobileObject | s : '_&_[T, T''''] >
   newo(downModule(T'), T'', T''') .

crl [message-out-newo] :
    < O : MobileObject | s : '_&_[T, '__['newo[T', T'', T'''], T'''', TL]] >
 => < O : MobileObject | s : '_&_[T, '__[T'''', TL]] >
    newo(downModule(T'), T'', T''')
 if TL =/= empty .
```

Before creating the mobile object, we check that the initial state given to the `newo` command as second argument together with the `start-up` message is a valid configuration.

When a mobile object is created, its number of hops is set to zero, and the forwarding information in the home at its parent location is initialized as expected—with its home location as the location it is at and zero as its number of hops. Note that the value initially given to the `gas` attribute of the new mobile object is 100, and that its identifier is included in the set of guests of its home.

```
rl [create-object] :
   newo(MOD, T, T')
   < h(L) : Home | cnt : N, guests : OS, forward : F >
=> if sortLeq(MOD,
          leastSort(MOD, '__[T, 'to_:_[T', 'start-up[upTerm(o(L, N))]]]),
          'Configuration)
   then < h(L) : Home | cnt : N + 1,
                        guests : (o(L, N), OS),
                        forward : insert(N, (L, 0), F) >
          < o(L, N) : MobileObject | mod : MOD,
                      s : '_&_['__[T, 'to_:_[T', 'start-up[upTerm(o(L, N))]]],
                                 'none.Configuration],
                      gas : 100,
                      hops : 0 >
   else < h(L) : Home | >
   fi .
```

5.5.5 Mobile object destruction

After it has completed its task, a mobile object's inner object may request the death of its container mobile object. The rule `message-out-kill` directly destroys the mobile object with the kill message in its outgoing messages tray (it must be the only one, so that all other messages have been previously handled). However, its home location must be informed, so that the forwarding information is appropriately updated.

```
  rl [message-out-kill] :
     < o(L, N) : MobileObject | s : '_&_[T, 'kill.Msg] >
     < h(L') : Home | guests : (o(L, N), OS) >
  => < h(L') : Home | guests : OS >
     to h(L) : dead { N } .
```

The `mobile-object-dead` rule updates the `forward` attribute of the destroyed object's home location.

```
  rl [mobile-object-dead] :
     to h(L) : dead { N }
     < h(L) : Home | forward : ((N |-> (L', N')), F) >
  => < h(L) : Home | forward : F >  .
 endom
```

5.6 A buying printers example

In this section we present a simple application to illustrate how mobile *application code* can be written in Maude and can be wrapped in mobile objects. In this example we have printers, buyers, and sellers; a buyer agent visits several printer sellers, who provide him information on their printers. The buyer looks for the cheapest printer, and once he has visited all the sellers, he goes back to the location of the seller offering the best price.

From the previous description, we can identify different actors, which may move freely from one process to another, and therefore could be represented as mobile objects. In the Mobile Maude approach the specification of the system consists of objects embedded inside mobile objects, which communicate with each other via messages. In addition to the term representing its state, each mobile object carries the *code* managing the behavior of the configuration of objects and messages representing such a state. The main difference with the usual specification of systems in Maude is that these objects must be aware of the fact that they are inside mobile objects, and that in order to communicate with (objects in) other mobile objects or to use some of the system messages available, they must follow the appropriate protocol.

In our sample application we have two different classes of mobile objects: sellers and buyers. Although in the simple example modeled here sellers do not move, they must be mobile objects, because they communicate with other mobile objects, and therefore have to be recognized as mobile objects by the Mobile Maude system. A buyer visits several sellers, and he asks each seller he visits for the description of the seller's printer, represented here only by its price. The seller sends back this information, which the buyer keeps if it corresponds to a better (cheaper) printer, otherwise he discards it. Once the buyer has visited all the sellers he knows, he goes back to the location of the best offer.

We represent sellers and buyers as objects of respective classes `Seller` and `Buyer`. Such objects in the application code will then be embedded as *inner objects* of their corresponding mobile objects.

Sellers receive messages of the form `get-printer-price(B)`, with `B` the identifier of the buyer mobile object sending the message. A seller can send messages of the form `printer-price(N)`, with `N` a natural number representing the printer's price. Both are defined of sort `Contents`, declared in the module `MOBILE-OBJECT-INTERFACE` (see Section 5.3).

```
fmod MESSAGES is
 ex MOBILE-OBJECT-INTERFACE .

 op get-printer-price : Mid -> Contents .
 op printer-price : Nat -> Contents .
endfm
```

The class `Seller` has an attribute `description` with the printer price.

```
omod SELLER is
 pr MESSAGES .

 class Seller | description : Nat .
```

A seller's behavior is represented by the following single rewrite rule: when a seller receives a description (price) request, he sends the description back to the seller.

```
 vars S B : Mid .
 var  N : Nat .
 var  Conf : Configuration .

 rl [get-des] :
    Conf (to S : get-printer-price(B))
    < S : Seller | description : N > & none
 => Conf < S : Seller | > & to B : printer-price(N) .
endom
```

Note the use of the `_&_` constructor. Since the printer description is sent to an object outside the mobile object in which the `Seller` object is located, the message is placed in the righthand outgoing tray. The rule `get-des` is applied only if the outgoing messages tray is empty, making sure in this way that any previous outgoing message has been handled. The `_&_` operator is the top operator of the term representing the state of the mobile object. Therefore, since there may be other objects and messages in the configuration in its lefthand side component, we include a variable `Conf` of sort `Configuration` to match the rest. Note also how an object may communicate with objects in other mobile objects, which may be in different processes, in a completely transparent way.

A buyer has an attribute `sellers` with a list of the identifiers of the known seller mobile objects. It also has an attribute `status` with its current state: `onArrival`, `asking`, `done`, or `buying`. Finally, the buyer keeps information about the printer with the best price in the attributes `price` and `bestSeller` of sorts, respectively, `Maybe{Nat}` and `Maybe{Oid}`. Initially, these two last attributes are `null`.

```
omod BUYER is
 pr LIST{Mid} * (op __ to _._, op nil to no-id) .
 pr MAYBE{Nat} * (op maybe to null) .
 pr MAYBE{Mid} * (op maybe to null) .
 pr MESSAGES .

 sort Status .
 ops onArrival asking done buying : -> Status .

 class Buyer | sellers : List{Mid},
               status : Status,
               price : Maybe{Nat},
               bestSeller : Maybe{Oid} .
```

The first rewrite rule, move, handles the travels of the buyer to request information on printers: if it is not in the middle of a request (its status is done) and there is at least one seller name in the sellers attribute, it asks the system to take it to the host where the next seller is located.

```
 var  S S' B : Mid .
 var  OS : List{Mid} .
 vars N N' : Nat .
 var  L : Loc .

 rl [move] :
    < B : Buyer | sellers : o(L, N) . OS, status : done > Conf & none
 => < B : Buyer | status : onArrival > Conf & go-find(o(L, N), L) .
```

Since Mobile Maude guarantees that mobile objects moving from one process to another are frozen (see Section 5.5.3), we know that, once the go-find command is given in the move rule, the buyer object will not be able to do anything until the mobile object in which it is embedded has reached the seller's process. Therefore, since there is no rule taking a Buyer object in onArrival state and a non-empty outgoing messages tray, this object will not do anything until it reaches its destination.

On arrival, the buyer asks the seller for the printer description.

```
 rl [onArrival] :
    < B : Buyer | sellers : S . OS, status : onArrival > Conf & none
 => < B : Buyer | status : asking > Conf & (to S : get-printer-price(B)) .
```

When the printer price arrives, if it corresponds to the first time the buyer is asking for a price (the attributes price and bestSeller are null) the buyer keeps it as the best known price; otherwise, it compares it with the best known printer and updates its information if needed. Notice that the first identifier in the list of known sellers gives us the identifier of the seller it is currently interacting with.

```
 rl [new-des] :
    (to B : printer-price(N))
    < B : Buyer | sellers : S . OS, price : null, status : asking,
                  bestSeller : null >
 => < B : Buyer | sellers : OS, price : N, status : done, bestSeller : S > .
```

```
  rl [new-des] :
     (to B : printer-price(N))
     < B : Buyer | sellers : S . OS, price : N', bestSeller : S',
                   status : asking >
  => if (N < N') then
          < B : Buyer | sellers : OS, price : N, bestSeller : S,
                        status : done >
     else < B : Buyer | sellers : OS, status : done >
     fi .
```

Notice that since these last rules do not imply the sending of any message out of the mobile object, we do not need to use the _&_ operator and the variable `Conf` to encompass the whole state.

Finally, when the list of remaining sellers is empty, the buyer travels to find the best buyer and reaches the `buying` status.

```
  rl [buy-it] :
     < B : Buyer | sellers : no-id, bestSeller : o(L, N), status : done >
     Conf & none
  => < B : Buyer | status : buying > Conf & go-find(o(L, N), L) .
 endom
```

Let us see an example of a distributed configuration, and how we can rewrite it by using the `erewrite` command. Our sample buyers/sellers configuration, shown in Figure 5.2, is constituted by three located configurations, each one to be executed in a Maude process (as we have already seen in previous chapters). The architecture is transparent to the application and thus it is not shown in the figure, although we have to take care of it in the initial configurations. The first located configuration (shown in the middle of the figure) contains a `Home` with identifier `h(l(ip0, 0))`, a mobile object with identifier `o(l(ip0, 0), 0)` with a `Seller` in its belly, and the `StarCenter` (so it must be executed before the other two ones). The Maude command to introduce the initial state of this configuration is as follows:

```
 mod PRINTERS-CENTER is
  pr SELLER .
  pr MOBILE-MAUDE-SEMANTICS .
  pr STAR-CENTER{MM-Complement} .
 endm

 erew  <> < l(ip0, 0) : StarCenter |
            neighbors : empty,
            state : idle,
            defNeighbor : null,
            port : 60039 >
          < h(l(ip0, 0)) : Home |
            cnt : 1,
            guests : o(l(ip0, 0), 0),
            forward : 0 |-> (l(ip0, 0), 0) >
          < o(l(ip0, 0), 0) : MobileObject |
            mod : upModule('SELLER, false),
            s : upTerm(< o(l(ip0, 0), 0) : Seller | description : 30 >
                       & none),
            gas : 200,
            hops : 0 > .
```

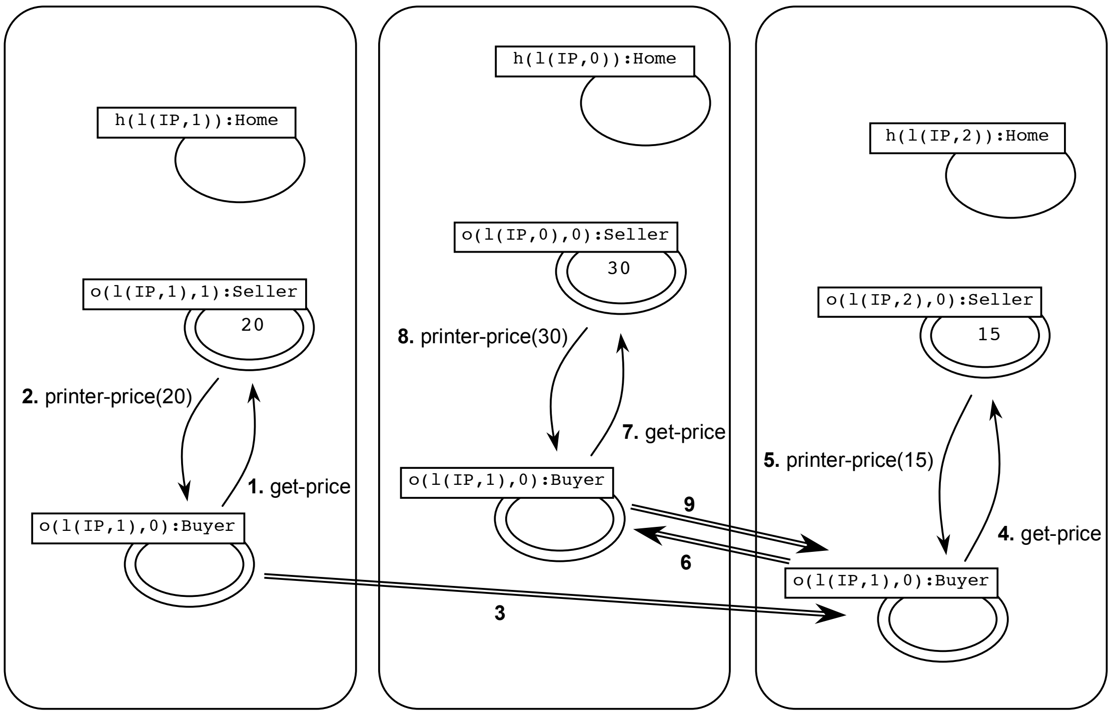

Figure 5.2: Buyers and sellers configuration

Note how the function `upModule` is used to obtain the metarepresentation of the module `SELLER`, and how the function `upTerm` is used to metarepresent the initial state of the inner object.

The second located configuration (on the left) contains a `StarNode`, a `Home`, a `Buyer` and a `Seller` with cheaper printers. The Maude command, introduced in a different Maude process, is the following one:

```
erew  <> < l(ip1, 0) : StarNode |
          neighbors : empty,
          state : idle,
          defNeighbor : null,
          port : 60039,
          center : ip0 >
      < h(l(ip1, 0)) : Home |
          cnt : 1,
          guests : (o(l(ip1, 0), 0), o(l(ip1, 0), 1)),
          forward : ((0 |-> (l(ip1, 0), 0)), (1 |-> (l(ip1, 0), 0))) >
      < o(l(ip1, 0), 0) : MobileObject |
          mod : upModule('BUYER, false),
          s : upTerm(< o(l(ip1, 0), 0) : Buyer |
                        price : null,
                        status : done,
                        bestSeller : null,
                        sellers : (o(l(ip1, 0), 1) .
                                   o(l(ip2, 0), 0) .
                                   o(l(ip0, 0), 0)) >
                    & none),
          gas : 200,
          hops : 0 >
        < o(l(ip1, 0), 1) : MobileObject |
          mod : upModule('SELLER, false),
          s : upTerm(< o(l(ip1, 0), 1) : Seller | description : 20 >
                     & none),
          gas : 200,
          hops : 0 > .
```

Finally, the third located configuration (on the right in the figure) contains another `StarNode`, a `Home`, and a `Seller` with the cheapest printers.

```
  erew <> < l(ip2, 0) : StarNode |
                          neighbors : empty,
                          state : idle,
                          defNeighbor : null,
                          port : 60039,
                          center : ip0 >
        < h(l(ip2, 0)) : Home |
                          cnt : 1,
                          guests : o(l(ip2, 0), 0),
                          forward : 0 |-> (l(ip2, 0), 0) >
        < o(l(ip2, 0), 0) : MobileObject |
          mod : upModule('SELLER, false),
          s : upTerm(< o(l(ip2, 0), 0) : Seller | description : 15 >
                     & none),
          gas : 200,
          hops : 0 > .
```

Figure 5.2 shows the order in which the different actions occur. First, the buyer asks the seller at his same location (price 20). Then, the buyer travels to the location on the right and asks the seller who sells printers costing 15. After that, the buyer travels to the middle location and asks the seller there (price 30). Finally, the buyer travels to the right location to find the seller with the best offer.

An execution of a Mobile Maude application is not intended to terminate, since the located configurations are always waiting for messages or mobile objects to come in from other configurations. As we showed in Section 2.7, we can stop the execution when it seems to be finished by typing ^C.

In the first Maude process we obtain the following configuration:

```
  result Configuration:
    <> receive(socket(5), b(socket(5)))
    receive(socket(6), b(socket(6)))
    receive(socket(7), b(socket(7)))
    Receive(b(socket(6)), l(ip0, 0))
    Receive(b(socket(7)), l(ip0, 0))
  < h(l(ip0, 0)) : Home |
        cnt : 1,
        guests : o(l(ip0, 0), 0),
        forward : 0 |-> l(ip0, 0), 0 >
  < b(socket(5)) : BufferedSocket |
        read : "",
        complete : notFound >
  < b(socket(6)) : BufferedSocket |
        read : "",
        complete : notFound >
  < b(socket(7)) : BufferedSocket |
        read : "",
        complete : notFound >
  < l(ip0, 0) : StarCenter |
        state : active,
        neighbors : (l(ip1, 0) |-> b(socket(6)),
                     l(ip2, 0) |-> b(socket(7))),
        defNeighbor : null,
        port : 60039 >
  < o(l(ip0, 0), 0) : MobileObject |
        mod : mod_is_sorts_._____endm(...),
        s : ('_&_['<_:_|_>['o['l['ip0.String,'0.Zero],'0.Zero],
    'Seller.Seller,'description`:_['s_^30['0.Zero]]],'none.Configuration]),
        gas : 199,
        hops : 0 >
```

In the second Maude process we obtain:

```
result Configuration:
  <> receive(socket(5), b(socket(5)))
  Receive(b(socket(5)), l(ip1, 0))
< h(l(ip1, 0)) : Home |
      cnt : 1,
      guests : o(l(ip1, 0), 1),
      forward : (0 |-> l(ip2, 0),4, 1 |-> l(ip1, 0),0) >
< b(socket(5)) : BufferedSocket |
      read : "",
      complete : notFound >
< l(ip1, 0) : StarNode |
      state : active,
      neighbors : empty,
      defNeighbor : b(socket(5)),
      port : 60039,
      center : ip0 >
< o(l(ip1, 0), 1) : MobileObject |
      mod : mod_is_sorts_._____endm(...),
      s : ('_&_['<_:_|_>['o['l['ip1.String,'0.Zero],'s_[
  '0.Zero]],'Seller.Seller,'description`:_['s_^20['0.Zero]]],
  'none.Configuration]),
      gas : 199,
      hops : 0 >
```

And in the third Maude process we obtain:

```
result Configuration:
  <> receive(socket(5), b(socket(5)))
  Receive(b(socket(5)), l(ip2, 0))
< h(l(ip2, 0)) : Home |
      cnt : 1,
      guests : (o(l(ip1, 0), 0), o(l(ip2, 0), 0)),
      forward : 0 |-> l(ip2, 0),0 >
< b(socket(5)) : BufferedSocket |
      read : "",
      complete : notFound >
  < l(ip2, 0) : StarNode |
        state : active,
        neighbors : empty,
        defNeighbor : b(socket(5)),
        port : 60039,
        center : ip0 >
  < o(l(ip1, 0), 0) : MobileObject |
        mod : mod_is_sorts_._____endm(...),
        s : ('_&_['<_:_|_>['o['l['ip1.String,'0.Zero],'0.Zero],
    'Buyer.Buyer,'_`,_['bestSeller`:_['o['l['ip2.String,
    '0.Zero],'0.Zero]],'price`:_['s_^15['0.Zero]],'sellers`:_[
    'no-id.List`{Mid`}],'status`:_['buying.Status]]],'none.Configuration]),
        gas : 190,
        hops : 4 >
< o(l(ip2, 0), 0) : MobileObject |
        mod : mod_is_sorts_._____endm(...),
        s : ('_&_['<_:_|_>['o['l['ip2.String,'0.Zero],'0.Zero],
    'Seller.Seller,'description`:_['s_^15['0.Zero]]],'none.Configuration]),
        gas : 199,
        hops : 0 >
```

Note that the buyer has finished his travel at the same location as that of the best seller.

5.7 An auction example

We show here an example where several auction centers send invitations offering their items to some buyers. If these buyers want any item, they send an agent with some money to buy them. The agent tries to buy the items and returns with the bought one, returning the remaining money so the buyer can send another agent to other auction.

We first identify all the actors in this example, that will interact with each other. We have buyers, agents, and auction centers. Although some of them do not travel, they interchange messages, so they must be mobile objects.

To increase the non-determinism we define a MONEY module, that uses the RANDOM predefined module to return a (pseudo) random amount of money, always less or equal than the natural number given as argument of the money operator.

```
 mod MONEY is
  pr RANDOM .
  inc NAT .

  op money : Nat -> [Nat] .

  var  N : Nat .

  rl [money1] :
     money(N)
  => s(random(N) rem N) .

  rl [money2] :
     money(N)
  => s(random(N + 1) rem N) .

  rl [money3] :
     money(N)
  => s(random(N + 2) rem N) .
 endm
```

We define the sort for the items to be sold. We define a view `Item` to be able to use it with the predefined parametric modules. We also specify the sort `Answer`, that represents the possible answers of a buyer to an invitation.

```
fmod ITEM is
 sort Item .
 ops item1 item2 item3 : -> Item .
endfm

view Item from TRIV to ITEM is
 sort Elt to Item .
endv

fmod ANSWER is
 sort Answer .
 ops ok refuse : -> Answer .
endfm
```

Now we define the messages that are going to be exchanged by the mobile objects.

```
fmod MESSAGES is
 pr ANSWER .
 inc MOBILE-OBJECT-ADDITIONAL-DEFS .
 pr SET{Item} * (sort Set{Item} to ItemSet, op empty to noItem) .
```

- The auction center notifies a seller about a new auction.

  ```
  op auction in_with_ : Mid ItemSet -> Contents .
  ```

- The buyer answers to the center if it will send an agent to the auction.

  ```
  op _answers_ : Mid Answer -> Contents .
  ```

- An agent informs the identifier of its client to the auction the center.

  ```
  op _represents to_ : Mid Mid -> Contents .
  ```

- A center offers an item to be auctioned.

  ```
  op item_ : Item -> Contents .
  ```

- For each item offered in the auction, the agent can make a bid.

  ```
  op no offer : -> Contents .
  op _offers_ : Oid Nat -> Contents .
  ```

- The auction center informs whenever an agent obtains an item (with the price it must pay), and when the auction is over.

```
    op winner : Item Nat -> Contents .
    op auction end : -> Contents .
```

- Finally, the agent returns with its client, giving him the bought items, communicating the items that it could not buy, and returning the remaining money.

```
    op I bought_remaining_and_ : ItemSet ItemSet Nat -> Contents .
   endfm
```

Agents must travel to the auction center location, try to obtain the items the buyer is looking for, and return with these items and the remaining money. The `Agent` class has the following attributes:

- the `money` that can be used to pay the items;
- the identifier of its `client`;
- the items `wanted` by its client and not obtained yet;
- the items that has been already `bought`;
- the identifier of the `auction-center` where the agent must work; and
- the current `state` of the agent, of sort `AgentState`.

```
omod AGENT is
 pr MESSAGES .

 sort AgentState .
 ops initial working finishing travelling end : -> AgentState .

 class Agent | money : Nat, client : Oid, wanted : ItemSet, bought : ItemSet,
              auction-center : Oid, state : AgentState .
```

Since agents are created by the buyers, the first thing they must do is to receive the `start-up` message, that Mobile Maude sends them when they are created. When an agent receives this message, it updates its name and sets its state to `initial`.

```
 var  I : Item .
 vars IS IS' : ItemSet .
 vars N N' N'' : Nat .
 var  L : Loc .
 vars O O' O'' : Mid .
 var  Conf : Configuration .

rl [start-up] :
   (to tmp-id : start-up(O))
   < tmp-id : Agent | state : ST >
=> < O : Agent | state : initial > .
```

The second action that the agent takes is to travel to the auction center location.

```
rl [travel] :
   < O : Agent | state : initial, auction-center : o(L, N) > Conf & none
=> < O : Agent | state : travelling > Conf & go-find(o(L, N), L) .
```

When the agent arrives to this location, it notifies the name of its client to the auction center, reaching the working state.

```
rl [agent-arrival] :
   < O : Agent | state : travelling, client : O', auction-center : O'' >
   Conf & none
=> < O : Agent | state : working > Conf &
   (to O'' : O represents to O') .
```

When a new item is offered, the agent checks whether its client wants the item or not. When the client wants the item, the agent makes an offer *non-deterministically* by using the money operator from module MONEY, so we can obtain different results depending on the selected rule.

```
crl [new-item] :
    (to O : item I)
    < O : Agent | wanted : IS, auction-center : O' > Conf & none
 => < O : Agent | > Conf & (to O' : no offer)
 if not I in IS .

crl [new-item] :
    (to O : item I)
    < O : Agent | money : N, wanted : (I, IS), auction-center : O' > Conf & none
 => < O : Agent | > Conf & (to O' : O offers N')
 if N > 0 /\
    money(N) => N' .
```

If the agent makes the best offer, it receives a message with the item and the price.

```
rl [winner] :
   (to O : winner(I, N'))
   < O : Agent | money : N, wanted : (I, IS), bought : IS' >
=> < O : Agent | money : sd(N, N'), wanted : IS, bought : (I, IS') > .
```

When the auction finishes, the agent goes to the client location.

```
rl [auction-end] :
   (to O : auction end)
   < O : Agent | client : o(L, N), state : working > Conf & none
=> < O : Agent | state : finishing >
   Conf & go-find(o(L, N), L) .
```

When the agent arrives to the client location, it gives to the buyer the bought items, the list of items that it could not buy, and the remaining money, and reaches the end state, to be finally deleted.

```
rl [end] :
   < O : Agent | money : N, client : O', wanted : IS, bought : IS',
                 state : finishing > Conf & none
=> < O : Agent | state : end > Conf &
   (to O' : bought IS' remaining IS and N) .

rl [end] :
   < O : Agent | state : end > Conf & none
=> < O : Agent | > Conf & kill .
endom
```

A buyer just waits while the agents it has created travel through the auction centers and get the items it wants. The `Buyer` class has the following attributes:

- the `money` it can give to the agents to buy its desired items;
- the items `wanted`, that the agents must try to obtain;
- the items `bought` by the agents so far; and
- the current `state` of the buyer.

```
omod BUYER is
 pr AGENT .

 sort BuyerState .
 ops buying finished : -> BuyerState .

 class Agent | money : Nat, wanted : ItemSet, bought : ItemSet,
               state : BuyerState .
```

When a new auction is announced to the buyer it checks if it is interested in any of the offered items. Then it creates an agent that sends to the auction center. The items that the buyer wants from this auction are removed from the `wanted` attribute. If finally the agent does not get them, the buyer can to try obtain them in another auction. The money received by the agent is decided non-deterministically by means of the rules from module `MONEY`.

```
 vars IS IS' IS'' IS''' newWanted : ItemSet .
 vars N N' : Nat .
 vars O O' O'' : Mid .
 var  Conf : Configuration .

 crl [new-auction] :
     (to O : auction in O' with IS')
     < O : Buyer | money : N, wanted : IS, state : buying > Conf & none
  => < O : Buyer | money : sd(N, N'), wanted : IS \ IS', state : start > Conf &
        if intersection(IS, IS') =/= noItem then
          newo(upModule('AGENT, false),
               < tmp-id : Agent |
                     money : N', wanted : intersection(IS, IS'), client : O,
                     bought : noItem, auction-center : O',
                     state : initial >,
               tmp-id)
          to O' : O answers ok
        else
          to O' : O answers refuse
        fi
  if N > 0 /\
     money(N) => N' .
```

When the agent arrives with the bought items, the buyer checks if it wants more things or not, and changes its state accordingly.

```
  rl [new-items] :
     (to O : bought IS'' remaining IS''' and N')
     < O : Buyer | money : N, wanted : IS, bought : IS', state : buying >
  => < O : Buyer | money : (N + N'), wanted : (IS, IS'''),
                   bought : (IS', IS'') > .

  rl [new-items] :
     < O : Buyer | wanted : noItem, state : buying >
  => < O : Buyer | state : finished > .
 endom
```

An auction center is in charge of inviting all the possible buyers to the auction, waiting for the agents of the buyers that confirmed the assistance, and selling the articles. To achieve it, the class `Center` has the following attributes:

- the `items` to be sold;
- the `buyers` possibly interested in the auction;
- the buyers that have `accepted` the invitation to the auction;
- the `agents` that have come to the auction;
- the `current` auctioned item;
- the agent (`bestAgent`) that has made the highest offer (`bestOffer`);
- a `counter` to keep track of the number of buyer's answers and agent's offers; and
- its current `state`.

```
 omod AUCTION-CENTER is
  pr MESSAGES .
  pr SET{Mid} .
  pr MAYBE{Mid} .
  pr MAYBE{Nat} .
  pr MAYBE{Item} .

  sort CenterState .
  ops inviting waitingResponse waitingAgents waitingOffers
      working finished : -> CenterState .

  class Center | items : ItemSet, buyers : Set{Mid}, accepted : Set{Mid},
                 agents : Set{Mid}, current : Maybe{Item},
                 bestAgent : Maybe{Mid}, bestOffer: Maybe{Nat}, counter : Nat,
                 state : CenterState .
```

We have two kinds of auctions:

- *Sealed-bid first-price* auction, where all bidders simultaneously submit bids so that no bidder knows the bid of any other participant. The highest bidder pays the price they submitted.
- *Sealed-bid second-price* auction, where the bidders make the offers in the same way than the auction above, but the highest bidder pays the second highest price.

We distinguish between the two types of auction by creating two subclasses of `Center`. The centers that use the first type will be instances of the class `SBFP`, that has no new attributes, while the centers that use the second type are instances of `SBSP`, that has a new attribute `second`, where they keep the second highest price.

```
class SBFP | .
subclass SBFP < Center .
class SBSP | second : Maybe{Nat} .
subclass SBSP < Center .
```

Initially, the auction center sends an invitation to all the buyers it knows.

```
vars O O' O'' : Mid .
vars OS OS' : Set{Mid} .
vars N N' N'' : Nat .
var  Conf : Configuration .
var  I : Item .
var  IS : ItemSet .
var  A : Answer .
var  C : Contents .

rl [invite] :
   < O : Center | buyers : OS, state : inviting,
                  items : IS > Conf & none
=> < O : Center | state : waitingResponse > Conf &
   broadcast(OS, auction in O with IS) .

op broadcast : Set{Mid} Contents -> Configuration .
eq broadcast(empty, C) = none .
eq broadcast((O , OS), C) = broadcast(OS, C) (to O : C) .
```

When the auction center receives an answer from a buyer, it updates its `accepted` and `counter` attributes.

```
rl [buyer-answer] :
   (to O : O' says A)
   < O : Center | accepted : OS, counter : N, state : waitingResponse >
=> < O : Center | accepted : if (A == ok) then (O' , OS) else OS fi,
                  counter : s(N) > .
```

Once all the buyers have sent the response, the center waits for their agents.

```
crl [all-answers] :
    < O : Center | buyers : OS, counter : N, state : waitingResponse >
 => < O : Center | counter : 0, state : waitingAgents >
 if | OS | == N .
```

As before, the auction center waits for all the agents, and once they have arrived, the auction starts.

```
rl [new-agent] :
   (to O : O' represents to O'')
   < O : Center | accepted : (O'' , OS), agents : OS',
                  state : waitingAgents >
```

```
 => < O : Center | agents : (O' , OS') > .

 crl [all-agents] :
     < O : Center | accepted : OS, agents : OS', state : waitingAgents >
  => < O : Center | state : working >
  if | OS | == | OS' | .
```

While there are available items for the auction and there is no currently auctioned item (that is, current is maybe), a new one is announced.

```
  rl [new-item] :
     < O : Center | agents : OS, items : (I, IS), current : maybe,
                    state : working > Conf & none
  => < O : Center | items : IS, current : I, state : waitingOffers >
     Conf & broadcast(OS, item I) .
```

When an answer arrives and it is not an offer, the center only updates the counter.

```
  rl [new-response] :
     (to O : no offer)
     < O : Center | counter : N >
  => < O : Center | counter : s(N) > .
```

When a bid arrives, we distinguish between the classes of auctions. In the case of the SBFP objects, they only update the attributes bestAgent and bestOffer.

```
  rl [new-response] :
     (to O : O' offers N)
     < O : SBFP | bestAgent : maybe, bestOffer : maybe, counter : N' >
  => < O : SBFP | bestAgent : O', bestOffer : N, counter : s(N') > .

  rl [new-response] :
     (to O : O' offers N)
     < O : SBFP | bestAgent : O'', bestOffer : N', counter : N'' >
  => if N > N' then
         < O : SBFP | bestAgent : O', bestOffer : N, counter : s(N'') >
     else
         < O : SBFP | bestAgent : O'', bestOffer : N', counter : s(N'') >
     fi .
```

In the case of the SBSP objects, they also update the second attribute. The first time a bid is received bestOffer and second have the same value, but when other bids arrive these values differ.

```
  rl [new-response] :
     (to O : O' offers N)
     < O : SBSP | bestAgent : maybe, bestOffer : maybe, second : maybe,
                  counter : N' >
  => < O : SBSP | bestAgent : O', bestOffer : N, second : N, counter : s(N') > .

  rl [new-response] :
     (to O : O' offers N)
     < O : SBSP | bestAgent : O'', bestOffer : N', second : N'', counter : N''' >
  => if N > N' then
```

```
        < O : SBSP | bestAgent : O', bestOffer : N, second : N',
                     counter : s(N''') >
    else
        < O : SBSP | counter : s(N''') >
    fi .
```

When all the agents have sent their answers, the center checks if there is an offer (that is, the `bestOffer` is not `maybe`) and sends the item to the winner. The difference between the two classes is the price communicated in the `winner` message.

```
  crl [all-offers] :
      < O : SBFP | bestAgent : O', bestOffer : N, counter : N', current : I,
                     agents : OS, state : waitingOffers > Conf & none
   => < O : SBFP | bestAgent : maybe, bestOffer : maybe, counter : 0,
                   current : maybe, state : working >
      Conf & (to O' : winner(I, N))
   if | OS | == N' .

  crl [all-offers] :
      < O : SBSP | bestAgent : O', bestOffer : N, second : N', counter : N'',
                   current : I, agents : OS, state : waitingOffers, AtS >
      Conf & none
   => < O : SBSP | bestAgent : maybe, bestOffer : maybe, second : maybe,
                   counter : 0, current : maybe, state : working >
      Conf & (to O' : winner(I, N'))
   if | OS | == N'' .
```

If there were no offers, the item is discarded.

```
  crl [all-offers] :
      < O : Center | bestAgent : maybe, bestOffer : maybe, counter : N',
                     current : I, agents : OS, state : waitingOffers >
   => < O : Center | counter : 0, current : maybe, state : working >
   if | OS | == N' .
```

Finally, when the set of items to sell becomes empty, the auction finishes.

```
  rl [no-more-items] :
     < O : Center | items : noItem, agents : OS, state : working >
     Conf & none
  => < O : Center | state : finished > Conf & broadcast(OS, auction end) .
 endom
```

We can use the star architecture to execute an example. An initial configuration for the star center with an auction center of class SBFP is:

```
 erew <> < l(ip0, 0) : StarCenter |
           neighbors : empty,
           state : idle,
           defNeighbor : null,
           port : 60039 >
         < h(l(ip0, 0)) : Home |
           cnt : 1,
           guests : o(l(ip0, 0), 0),
           forward : 0 |-> (l(ip0, 0), 0) >
        < o(l(ip0, 0), 0) : MobileObject |
           mod : upModule('AUCTION-CENTER, false),
           s : upTerm(< o(l(ip0, 0), 0) : SBFP | buyers : (o(l(ip1, 0), 0),
                                                           o(l(ip2, 0), 0)),
                                                 accepted : empty,
                                                 agents : empty,
                                                 items : item1,
                                                 state : inviting,
                                                 current : maybe,
                                                 bestOffer : maybe,
                                                 bestAgent : maybe,
                                                 counter : 0 >
                      & none),
           gas : 200,
           hops : 0 > .
```

A star node with a buyer looks as follows:

```
 erew <> < l(ip1, 0) : StarNode |
           neighbors : empty,
           state : idle,
           defNeighbor : null,
           port : 60039,
           center : ip0 >
        < h(l(ip1, 0)) : Home |
           cnt : 1,
           guests : o(l(ip1, 0), 0),
           forward : 0 |-> (l(ip1, 0), 0) >
        < o(l(ip1, 0), 0) : MobileObject |
           mod : upModule('BUYER, false),
           s : upTerm(< o(l(ip1, 0), 0) : Buyer | money : 10,
                                                  bought : noItem,
                                                  wanted : item1,
                                                  state : buying >
                      & none),
           gas : 200,
           hops : 0 > .
```

An auction center of class SBFP located in other star node is defined as follows:

```
erew <> < l(ip3, 0) : StarNode |
            neighbors : empty,
            state : idle,
            defNeighbor : null,
            port : 60039,
            center : ip0 >
        < h(l(ip3, 0)) : Home |
            cnt : 1,
            guests : o(l(ip3, 3), 0),
            forward : 0 |-> (l(ip3, 0), 0) >
    < o(l(ip3, 0), 0) : MobileObject |
         mod : upModule('AUCTION-CENTER, false),
         s : upTerm(< o(l(ip3, 3), 0) : SBSP | buyers : (o(l(ip1, 0), 0),
                                                         o(l(ip2, 0), 0)),
                                                                 accepted : empty,
                                                                 agents : empty,
                                                                 items : (item1, item4),
                                                                 state : inviting,
                                                                 current : maybe,
                                                                 bestOffer : maybe,
                                                                 second : maybe,
                                                                 bestAgent : maybe,
                                                                 counter : 0 >
                         & none),
            gas : 200,
            hops : 0 > .
```

5.8 Mobile Maude skeletons

Another way in which the applications shown in Chapter 4 can be executed in a parallel way is by using mobile objects on top of Mobile Maude. The workers are mobile objects that are created by the master and travel to a free location to compute the solution to the subproblems they have been assigned. A better approach is to use skeletons, in this case by using "generic" mobile objects, that receive as data the module solving each concrete problem as initial information.

5.8.1 Euler numbers case study

We show an implementation of the Euler numbers problem (shown in Section 4.1.1) built with a master that distributes the work amongst several workers and combines their subresults. The concrete problems that must be solved in this application are Euler numbers.

We define first the messages that are going to be transmitted, that must have sort `Contents`, defined in `MOBILE-OBJECT-INTERFACE`. We just need two messages, one sending new problems to the workers and another communicating the (sub)result.

```
fmod MESSAGES is
 inc MOBILE-OBJECT-INTERFACE .

 op new-work : Nat -> Contents .
 op finished : Mid Nat -> Contents .
endfm
```

The `Worker` class has attributes that store the `master` identifier, the assigned `location` where it has to work, and the list of unfinished tasks (`next`).

```
omod WORKER is
 pr MESSAGES .
 pr EULER .

 class Worker | master : Mid, loc : Loc, next : NatList .
```

This module must import the messages shown above and the `EULER` module[4] described in Section 4.1.1. At start up, the worker travels to its assigned location.

[4]Note we use the `EULER` module, that contains only the functions solving single Euler numbers, not `SUM-EULER`, that contains the function computing the whole sum.

```
  var  M M' W O : Mid .
  vars N N' : Nat .
  var  L : Loc .
  var  NL : NatList .
  var  Conf : Configuration .

  rl [start-up] :
     (to tmp-id : start-up(O))
     < tmp-id : Worker | loc : L > Conf & none
  => < O : Worker | > Conf & go(L) .
```

When a new number arrives, the worker appends it to the list of unfinished tasks.

```
  rl [new-work] :
     to W : new-work(N)
     < W : Worker | next : NL >
  => < W : Worker | next : NL N > .
```

Finally, while the list of unfinished tasks is not empty, the worker calculates another Euler number and sends it to the master.

```
  crl [working] :
      < W : Worker | next : N NL, master : M > Conf & none
   => < W : Worker | next : NL > Conf & to M : finished(W, N')
   if N' := euler(N) .
 endom
```

The `Master` class keeps information about the number of workers it must create (`numWorkers`); the partial `result`; the `current` Euler number we must compute (which is decreased when new works are delivered); and the list of available locations where workers can be sent (`locs`).

```
 omod MASTER is
  pr WORKER .
  pr LIST{Oid} * (sort List{Oid} to LocList,
                  op nil to mtLocList) .

  class Master | numWorkers : Nat, result : Nat, current : Nat,
                 locs : LocList .
```

The first action the master takes is to create the objects that will work in the clients (so it imports the `WORKER` module), and assign three initial works[5] to each of them, in order to have the workers as occupied as possible, that is, computing another value while new tasks arrive.

```
  vars M O : Mid .
  vars N N' X Y N'' R C : Nat .
  var  Conf : Configuration .
  var  L : Loc .
  var  LL : LocList .

  rl [new-worker] :
```

[5]In the following section this initial assignment will be generalized.

```
    < O : Master | numWorkers : s(N), locs : L LL, current : N' > Conf & none
 => < O : Master | numWorkers : N, locs : LL L, current : sd(N', 3) > Conf &
    newo(upModule('WORKER, false),
            < tmp-id : Worker | master : O,
                                loc : L,
                                next : (N' sd(N', 1) sd(N', 2)) >,
            tmp-id) .
```

Note that the location L is moved from the beginning to the end of the loc list, so it could be used again if the initial number of workers is bigger that the length of this list.

When a new subresult arrives to the server, it is combined with the current result by adding them, and a new task is sent (if it is possible).

```
  rl [new-work] :
    to O : finished(M, N'') Conf & none
    < O : Master | current : s(N), result : N' >
 => < O : Master | current : N, result : N' + N'' > Conf &
    to M : new-work(s(N)) .

  rl [no-more-work] :
    to O : finished(M, N')
    < O : Master | current : 0, result : N >
 => < O : Master | result : N + N' > .
 endom
```

Like the Euler instantiation in Section 4.3.3, we use the star architecture in this example. The initial term for the master, that is located in the center of the star, in an example with two workers and 100 as initial number to compute is:

```
erew  <> < l(ip0, 0) : StarCenter |
                     neighbors : empty,
                     state : idle,
                     defNeighbor : null,
                     port : 60039 >
         < h(l(ip0, 0)) : Home |
                     cnt : 1,
                     guests : o(l(ip0, 0), 0),
                     forward : (0 |-> (l(ip0, 0), 0)) >
         < o(l(ip0, 0), 0) : MobileObject |
                     mod : upModule('MASTER, false),
                     s : upTerm(< o(l(ip0, 0), 0) : Master |
                                counter : 0,
                                numWorkers : 2,
                                result : 0,
                                locs : l(ip1, 0) l(ip2, 0),
                                current : 100 >
                            & none),
                     gas : 400,
                     hops : 0 > .
```

The unique difference between the two worker's locations is their name. The initial term for the first one is:

```
erew <> < l(ip1, 0) : StarNode |
                 state : idle,
                 neighbors : empty,
                 defNeighbor : null,
                 port : 60039,
                 center : ip0 >
         < h(l(ip1, 0)) : Home |
                 cnt : 0,
                 guests : empty,
                 forward : empty > .
```

5.8.2 The farm skeleton in Mobile Maude

In Chapter 4 we described how to specify algorithmic skeletons by simulating higher order functions by means of parameterization. We show here a different approach based in the `META-LEVEL` module, that allows to use (metarepresented) Maude modules as data. Our Mobile Maude farm skeleton will have a `Master` class with the number of workers (`numWorkers`); the partial `result`, that is now a `Term`, because it represents the result of all possible applications; the `current` subproblem (a `Term` too); the list of available locations (`locs`); the number of initial works assigned to each worker `numWorks`; and (the metarepresentation of) a `module`, that will store the concrete application code.

```
class Master | numWorkers : Nat, result : Term, current : Term,
               locs : LocList, numWorks : Nat, module : Module .
```

The metarepresented module must contain some fixed operators:

- `do-work`, that solves the subproblems;
- `reduce`, that updates the current problem by making it smaller;
- `next-work`, that gets the next subproblem from the current problem;
- `combine`, that merges the current (partial) result with a subresult; and
- `finished?`, that checks if there are more subproblems.

In the `Worker` class we still need the master identifier, the assigned `location`, and the list of unfinished tasks (`next`, that is now a term list), and we need a new attribute with (the metarepresentation of) the application `module`.

```
class Worker | master : Mid, loc : Loc, next : TermList, module : Module .
```

The messages must change in order to dispatch generic data. They transmit now data of type `Term`, that will represent the concrete data for each application.

```
mod MESSAGES is
 inc MOBILE-OBJECT-ADDITIONAL-DEFS .

 op new-work : Term -> Contents .
 op finished : Mid Term -> Contents .
endm
```

In the worker, the `start-up` and `new-work` rules remains almost unchanged, while the `working` rule uses the `metaReduce` operation (notice how the operator `do-work` defined in the application module is used).

```
crl [working] :
    < W : Worker | next : (T, TL), module : Mod, master : M > Conf & none
 => < W : Worker | next : TL > Conf & to M : finished(W, T')
 if T' := getTerm(metaReduce(Mod, 'do-work[T])) .
```

The main changes have been made in the master module. When a new worker is created, we assign it the tasks specified by `numWorks`.

```
crl [new-worker] :
    < O : Master | numWorkers : s(N), locs : L LL, module : Mod,
                   current : T, numWorks : N' > Conf & none
 => < O : Master | numWorkers : N, locs : LL L, current : T' > Conf &
    newo(upModule('WORKER, false),
           < tmp-id : Worker | module : Mod,
                               master : O,
                               next : getTasks(Mod, T, N'),
                               loc : L,
                               counter : 0,
                               st : tr >,
           tmp-id)
 if T' := update(Mod, T, N') .
```

where `getTasks` and `update` are operators that make use of `next-work`, `reduce`, and `finished?` at the metalevel to obtain the next tasks and update the current problem.

```
op getTasks : Module Term Nat -> TermList .
ceq getTasks(Mod, T, s(N)) = T', getTasks(Mod, T1, N)
 if getTerm(metaReduce(Mod, 'finished?[T])) == 'false.Bool /\
    T' := getTerm(metaReduce(Mod, 'next-work[T])) /\
    T1 := getTerm(metaReduce(Mod, 'reduce[T])) .
eq getTasks(Mod, T, N) = empty [owise] .

op update : Module Term Nat -> Term .
ceq update(Mod, T, s(N)) = update(Mod, T', N)
 if getTerm(metaReduce(Mod, 'finished?[T])) == 'false.Bool /\
    T' := getTerm(metaReduce(Mod, 'reduce[T])) .
eq update(Mod, T, N) = T [owise] .
```

When a new subresult arrives, we combine the results (by using `combine` at the metalevel) and distinguish again whether we have more tasks to send or not (by using `finished?`).

```
crl [new-work] :
    to O : finished(M, T'') Conf & none
    < O : Master | current : T, result : T', module : Mod >
 => < O : Master | current : T1, result : T2 > Conf &
    to M : new-work(T3)
 if getTerm(metaReduce(Mod, 'finished?[T])) == 'false.Bool /\
    T1 := getTerm(metaReduce(Mod, 'reduce[T])) /\
    T2 := getTerm(metaReduce(Mod, 'combine[T', T'']))  /\
    T3 := getTerm(metaReduce(Mod, 'next-work[T])) .

crl [no-more-work] :
    to O : finished(M, T'')
    < O : Master | current : T, result : T', module : Mod >
 => < O : Master | result : T1 >
 if getTerm(metaReduce(Mod, 'finished?[T])) == 'true.Bool /\
    T1 := getTerm(metaReduce(Mod, 'combine[T', T'']))  .
```

Euler numbers instantiation

For our Euler numbers example we define the following module:

```
fmod EULER-MM-PROBLEM is
 pr EULER .

 vars N N' : Nat .

 op do-work : Nat -> Nat .
 op reduce : Nat ~> Nat .
 op next-work : Nat -> Nat .
 op combine : Nat Nat -> Nat .
 op finished? : Nat -> Bool .

 eq do-work(N) = euler(N) .
 eq reduce(s(N)) = N .
 eq next-work(N) = N .
 eq combine(N, N') = N + N' .
 eq finished?(N) = N == 0 .
endfm
```

The initial configuration of the master, in an example with two workers and initial problem 100 is:

```
erew <> < l(ip0, 0) : StarCenter |
           neighbors : empty,
           state : idle,
           defNeighbor : null,
           port : 60039 >
        < h(l(ip0, 0)) : Home |
           cnt : 1,
           guests : o(l(ip0, 0), 0),
           forward : 0 |-> (l(ip0, 0),0) >
        < o(l(ip0, 0), 0) : MobileObject |
                mod : upModule('MASTER, false),
                s : upTerm(< o(l(ip0, 0), 0) : Master |
                           numWorkers : 2,
                           current : upTerm(100),
                           result : upTerm(0),
                           module : upModule('EULER-MM-PROBLEM, false),
                           locs : l(ip1, 0) l(ip2, 0),
                           counter : 0,
                           numWorks : 3 >
                       & none),
                gas : 2000,
                hops : 0 > .
```

where the `ipi` are strings denoting IP addresses.

The initial configuration for one of the locations that will receive workers is:

```
erew <> < l(ip1, 0) : StarNode |
           neighbors : empty,
           state : idle,
           defNeighbor : null,
           port : 60039,
           center : ip0 >
        < h(l(ip1, 0)) : Home |
           cnt : 0,
           guests : empty,
           forward : empty > .
```

Notice that the skeleton code (the master and the workers) will be metarepresented in the belly of the mobile objects, so the application code will have two reflection levels. Despite we have obtained an application that can be executed among several hosts, the two levels of reflection suppose a lot of "extra work", because the terms must move between them, thus performing additional work in order to be executed and obtaining almost the same time that the sequential applications obtain.

Chapter 6

Formal analysis of distributed applications

Formal verification is the process of checking whether a design satisfies some requirements (properties). In order to formally verify a distributed system, it must first be converted into a simpler "verifiable" format. To do that in Maude by using its provided tools, we must be able of represent the whole system in one single term in order to prove the properties over it.

Model checking is a method for formally verifying finite-state concurrent systems [16]. It has several important advantages over mechanical theorem provers or proof checkers; the most important is that the procedure is completely automatic. The main disadvantage is the *state space explosion*, that can occur if the system being verified has many components that can make transitions in parallel.

Maude's model checker [31] allows us to prove properties on Maude specifications when the set of states reachable from an initial state in such a Maude system module is finite. This is supported in Maude by its predefined `MODEL-CHECKER` module and other related modules, which can be found in the `model-checker.maude` file distributed with Maude.

The properties to be checked are described by using a specific property specification logic, namely *Linear Temporal Logic* (LTL) [46, 16], which allows specification of properties such as safety properties (ensuring that something bad never happens) and liveness properties (ensuring that something good eventually happens). Then, the model checker can be used to check whether a given initial state, represented by a Maude term, fulfills a given property.

Sometimes all the power of model checking is not needed. Another Maude's analysis tool is the `search` command (see Section 2.4), that allows to explore (following a breadth first search strategy) the reachable states in different ways. By using the `search` command we can check *invariants*. An invariant I is a predicate over a transition system defining a subset of states meeting two properties:

- it contains the initial state s_0.

- it contains any state reachable from s_0 through a finite number of transitions.

If an invariant holds, then we know that something "bad" can never happen, namely, the negation $\neg I$ of the invariant is impossible. Thus, if the command

```
search init =>* C:Configuration such that not I(C:Configuration) .
```

has no solution, then *I* holds.

Finally, we can also use the `search` command to check properties over final configurations.

We illustrate the use of all these techniques with a small example in the following section.

6.1 A taste of Maude analysis tools

We show here how to use the `search` command and the Maude model checker by means of some small examples. First, we explain the usage of `search` to check invariants with an example extracted from [18]. Consider a simple clock that marks the hours of the day. The implementation in Maude of such a clock is:

```
mod SIMPLE-CLOCK is
 pr INT .

 sort Clock .

 op clock : Int -> Clock [ctor] .
 var T : Int .
 rl clock(T) => clock((T + 1) rem 24) .
endm
```

An invariant property of this clock is that the hour marked will be greater or equal than `0` and less or equal than `24`. To check that this property is an invariant, we look for a configuration of the clock where it does not hold, that is

```
 search clock(0) =>* clock(T) such that T < 0 or T >= 24 .

 No solution.
```

Since Maude answers with `No solution.`, we can assert that the property is really an invariant.

As we have said, the `search` command can also be used to check properties about final states. Suppose we define a vending machine that dispenses apples for one dollar and cakes for three quarter dollars. The Maude specification of this machine (different from the one shown in Section 2.2) is defined as follows:

```
mod VENDING-MACHINE is

 sorts Coin Item Marking .
 subsorts Coin Item < Marking .

 op __ : Marking Marking -> Marking [assoc comm id: null] .
 op null : -> Marking .

 ops $ q : -> Coin .
 ops a c : -> Item .
```

```
  rl [buy-c] : $ => c .
  rl [buy-c] : q q q q => c .
  rl [buy-a] : $ => a q .
  rl [buy-a] : q q q => a .
```

We add a function `withApple` that checks if a marking contains an apple.

```
  var M : Marking .

  op withApple : Marking -> Bool .
  eq withApple(a M) = true .
  eq withApple(M) = false [owise] .
 endm
```

We can check that, if we start with the marking `$ $ $ q q q`, it is impossible to reach a final state (denoted with the notation `=>!`) that fulfills the condition that it contains no apples (denoted with the notation `such that`):

```
  search $ $ $ q q q =>! M:Marking such that not withApple(M:Marking) .

  No solution.
```

Now, we illustrate how to use the Maude model checker with a simple mutual exclusion example, also extracted from [18]. We define the module `MUTEX`, in which two processes, one named `a` and another named `b`, can be either waiting or in their critical section.

```
 mod MUTEX is

  sorts Name Mode Proc Token Conf .
  subsorts Token Proc < Conf .

  op none : -> Conf [ctor] .
  op __ : Conf Conf -> Conf [ctor assoc comm id: none] .
  ops a b : -> Name [ctor] .
  ops wait critical : -> Mode [ctor] .
  op [_,_] : Name Mode -> Proc [ctor] .
```

The processes take turns accessing their critical section by passing each other a different token (either $ or *).

```
  ops * $ : -> Token [ctor] .

  rl [a-enter] : $ [a, wait] => [a, critical] .
  rl [b-enter] : * [b, wait] => [b, critical] .
  rl [a-exit] : [a, critical] => [a, wait] * .
  rl [b-exit] : [b, critical] => [b, wait] $ .
 endm
```

Now we define the module defining the properties that will be checked about the system above. To use the model checker we just need to make explicit two things: the intended sort of states and the relevant *state predicates*, that is, the relevant LTL atomic propositions. The module `SATISFACTION` defines sorts for both the states (`State`) and the predicates (`Prop`), and an operator `_|=_`, that indicates when an state satisfies a predicate.

```
  op _|=_ : State Prop -> Bool [frozen] .
```

Our obvious sort for states is the sort `Conf` of configurations.

```
mod MUTEX-PREDS is
  protecting MUTEX .
  including SATISFACTION .

  subsort Conf < State .
```

In order to state the desired safety and liveness properties we need state predicates telling us whether a process is waiting or is in its critical section. We can make these predicates parametric on the name of the process and define their semantics.

```
  op crit : Name -> Prop .
  op wait : Name -> Prop .
```

We give now the equations, defining when each of the two parametric state predicates holds in a given state.

```
  var N : Name .
  var C : Conf .
  var P : Prop .

  eq [N, critical] C |= crit(N) = true .
  eq [N, wait] C |= wait(N) = true .
  eq C |= P = false [owise] .
 endm
```

Note the use of the `_|=_` to define the properties.

To check properties about the system above we must define an initial state. The module `LTL-SIMPLIFIER` tries to reduce the formulas in order to obtain a smaller automata, while the module `MODEL-CHECKER` defines the operator `modelCheck`, used to check if a state fulfills a formula. The result of reducing this function has sort `ModelCheckResult`, that can be either a Boolean or a counterexample.

```
  op modelCheck : State Formula ~> ModelCheckResult [special(...)] .
```

We use the following module:

```
 mod MUTEX-CHECK is
  protecting MUTEX-PREDS .
  including LTL-SIMPLIFIER .
  including MODEL-CHECKER .

  ops initial1 initial2 : -> Conf .
  eq initial1 = $ [a, wait] [b, wait] .
  eq initial2 = * [a, wait] [b, wait] .
 endm
```

The first obvious property to check is mutual exclusion:

```
red modelCheck(initial1, [] ~(crit(a) /\ crit(b))) .
result Bool: true

red modelCheck(initial2, [] ~(crit(a) /\ crit(b))) .
result Bool: true
```

If a property does not hold, a counterexample is shown. Suppose that we want to check if, beginning in the state `initial1`, process `b` will always be waiting.

```
 red modelCheck(initial1, [] wait(b)) .
```

Maude returns a sequence of pairs of states and rule names, where the process `b` obtains the `critical` state.

```
counterexample({$ [a, wait] [b, wait], a-enter}
 {[a, critical] [b, wait], a-exit}
 {* [a, wait] [b, wait], b-enter},
 {[a, wait] [b, critical], b-exit}
 {$ [a, wait] [b, wait], a-enter}
 {[a, critical] [b, wait], a-exit}
 {* [a, wait] [b, wait], b-enter})
```

6.2 Redefinition of the SOCKET module

In order to be able to represent a whole distributed configuration as a single term, we have provided an algebraic specification of sockets. We have redefined the SOCKET module, simulating the behavior of sockets on local configurations. This specification expresses processes as terms of a class `Process`, that has an attribute `conf`. Processes are identified with the name of the location, that corresponds with the name of the unique router in the configuration `conf`, in order to simplify the definition of the properties to be checked (see Section 6.5).

```
omod SOCKET is
 inc STRING .

 class Process | conf : Configuration .
```

Processes work as hosts in the distributed version, keeping their configuration separated from the others in their attribute. Message passing is then defined between processes instead of between hosts, where the messages are defined as follows:[1]

```
 msg createClientTcpSocket : Oid Oid String Nat -> Msg .
 msg createServerTcpSocket : Oid Oid Nat Nat -> Msg .
 msg createdSocket : Oid Oid Oid -> Msg .
 msg acceptClient : Oid Oid -> Msg .
 msg acceptedClient : Oid Oid String Oid -> Msg .
 msg send : Oid Oid String -> Msg .
 msg sent : Oid Oid -> Msg .
 msg receive : Oid Oid -> Msg .
 msg received : Oid Oid String -> Msg .
 msg closeSocket : Oid Oid -> Msg .
 msg closedSocket : Oid Oid String -> Msg .
```

[1]Since this centralized version does not have connection errors, messages dealing with these situations are not modeled.

Thus, we have specified the socket manager and client and server sockets to deal with processes:

- The socket manager is now an instance of a class `Manager`, with a `count` attribute to name new sockets.

  ```
  class Manager | count : Nat .

  op socketManager : -> Oid [ctor] .
  ```

- The client sockets are instances of a class `Socket` with attributes `source` (the source `Process`), `target` (the target `Process`), and `socketState` (the socket state). Notice that although we talk about source and target, sockets are bidirectional. Socket identifiers have the form `socket(N)` with `N` a natural number.

  ```
  class Socket | source : Oid, target : Oid, socketState : SocketState .

  sort SocketState .
  ops active closed : -> SocketState .

  op socket : Nat -> Oid [ctor] . ---- socket objects identifiers
  ```

- The server sockets are instances of the class `ServerSocket` with the attributes `address` (the server address), `port` (the server port), and `backlog` (the number of queue requests for connection that the server will allow). Server socket identifiers are constructed with the operator `server` and a natural number.

  ```
  class ServerSocket | address : String, port : Nat, backlog : Nat .

  op server : Nat -> Oid [ctor] . ---- server socket objects identifiers
  ```

If there is a `createServerTcpSocket` message in a configuration, we use the counter from the socket manager to create a `ServerSocket` with the values specified in the message, and a `createdSocket` message is put into the configuration.

```
vars SOCKET PID PID' O O' : Oid .
vars DATA ADDRESS S S' : String .
vars N M PORT BACKLOG : Nat .
vars CONF CONF' : Configuration .

rl [ServerTcpSocketCreation] :
   < PID : Process |
      conf : (createServerTcpSocket(socketManager, O, PORT, BACKLOG) CONF) >
   < socketManager : Manager | count : N >
=> < PID : Process |
      conf : (createdSocket(O, socketManager, server(N)) CONF) >
   < socketManager : Manager | count : s(N) >
   < server(N) : ServerSocket | address : "127.0.0.1", port : PORT,
                                backlog : BACKLOG > .
```

If there is an `acceptClient` message in one configuration and a server offers its services through the requested server socket, a new socket is created between the two processes, and the corresponding `acceptedClient` and `createdSocket` messages are put into the configurations.

```
  rl [SocketCreation] :
     < PID : Process | conf : (acceptClient(server(N), O) CONF) >
     < PID' : Process |
        conf : (createClientTcpSocket(socketManager, O', ADDRESS, PORT) CONF') >
     < socketManager : Manager | count : M >
     < server(N) : ServerSocket | address : ADDRESS, port : PORT >
  => < PID : Process |
           conf : (acceptedClient(O, server(N), ADDRESS, socket(M)) CONF) >
     < PID' : Process |
           conf : (createdSocket(O', socketManager, socket(M)) CONF') >
     < socketManager : Manager | count : s(M) >
     < server(N) : ServerSocket | >
     < socket(M) : Socket | source : PID, target : PID', socketState : active > .
```

If there is a send message in one configuration, the corresponding receive message in another one, and their processes are connected through a socket, then the message is delivered. We need two rules, depending on the values of source and target.

```
  rl [send] :
     < PID : Process | conf : (send(SOCKET, O, DATA) CONF) >
     < PID' : Process | conf : (receive(SOCKET, O') CONF') >
     < SOCKET : Socket | source : PID, target : PID', socketState : active >
  => < PID : Process | conf : (sent(O, SOCKET) CONF) >
     < PID' : Process | conf : (received(O', SOCKET, DATA) CONF') >
     < SOCKET : Socket | > .

  rl [send] :
     < PID : Process | conf : (send(SOCKET, O, DATA) CONF) >
     < PID' : Process | conf : (receive(SOCKET, O') CONF') >
     < SOCKET : Socket | source : PID', target : PID, socketState : active >
  => < PID : Process | conf : (sent(O, SOCKET) CONF) >
     < PID' : Process | conf : (received(O', SOCKET, DATA) CONF') >
     < SOCKET : Socket | > .
```

If an object requests to close a socket, its state changes to closed and a closedSocket message is put into the configuration.

```
  rl [closeSocket] :
     < PID : Process | conf : (closeSocket(SOCKET, O) CONF) >
     < SOCKET : Socket | socketState : active >
  => < PID : Process | conf : (closedSocket(O, socketManager, "") CONF) >
     < SOCKET : Socket | socketState : closed > .
```

If an object tries to send a message through a closed socket, it receives a closedSocket message.

```
  rl [closedSocket] :
     < PID : Process | conf : (send(SOCKET, O, MSG) CONF) >
     < SOCKET : Socket | socketState : closed >
  => < PID : Process | conf : (closedSocket(O, socketManager, "") CONF) >
     < SOCKET : Socket | > .
 endom
```

This module allows to represent in a single term a whole distributed configuration and formally analyze it without changing its specification. Thus, in order to prove a property

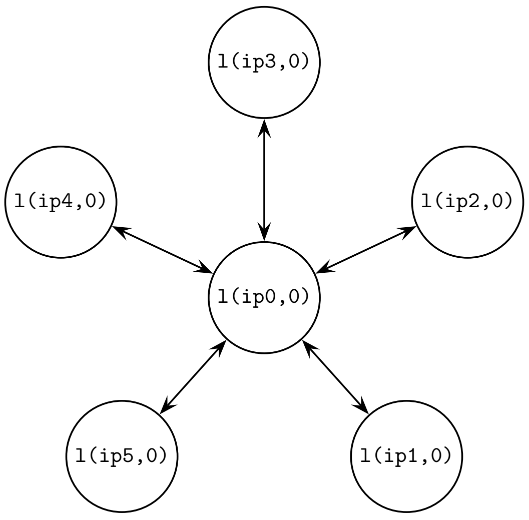

Figure 6.1: A concrete star architecture

about a distributed configuration we have to prove it on the corresponding "centralized" configuration. We show how one of these configurations looks by means of an example. The initial configuration for the star architecture (see Section 3.2) shown in Figure 6.1 is:

```
< socketManager : Manager | count : 0 >
< pid(0) : Process |
    conf : < l(ip0, 0) : StarCenter | state : idle,
                                       neighbors : empty,
                                       defNeighbor : null,
                                       port : 60039 > >
 < pid(1) : Process |
    conf : < l(ip1, 0) : StarNode | state : idle,
                                     neighbors : empty,
                                     defNeighbor : null,
                                     center : ip0,
                                     port : 60039 > >
 < pid(2) : Process |
    conf : < l(ip2, 0) : StarNode | state : idle,
                                     neighbors : empty,
                                     defNeighbor : null,
                                     center : ip0,
                                     port : 60039 > >
 < pid(3) : Process |
    conf : < l(ip3, 0) : StarNode | state : idle,
                                     neighbors : empty,
                                     defNeighbor : null,
                                     center : ip0,
                                     port : 60039 > >
 < pid(4) : Process |
    conf : < l(ip4, 0) : StarNode | state : idle,
                                     neighbors : empty,
                                     defNeighbor : null,
                                     center : ip0,
                                     port : 60039 > >
```

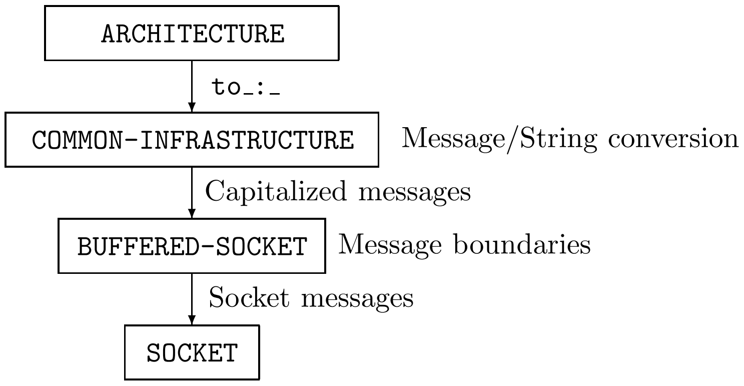

Figure 6.2: Initial design

```
< pid(5) : Process |
   conf : < l(ip5, 0) : StarNode | state : idle,
                                   neighbors : empty,
                                   defNeighbor : null,
                                   center : ip0,
                                   port : 60039 > >
```

6.3 Using different abstraction levels

In the previous section we have simulated Maude sockets in order to obtain the whole configuration of the system in one single term. This already allows us to use the Maude tools to analyze our specifications. However, we could trust some of the components of the applications in order to speed up their formal analysis. We show in the following sections how to incrementally abstract different parts of the design, until we finally abstract the whole architecture.

First we remind how the architecture works. Figure 6.2 shows the design we have described in the previous chapters. The algebraic socket module provides the messages and rules of the built-in SOCKET module; the buffered sockets assure the preservation of message boundaries and handle the capitalized messages shown in Section 2.8; the common infrastructure redirects the messages of the form to_:_, converting them first into string and back again to messages once they have arrived to their destination. We show in the following sections how we can remove each of these layers without changing the layers above.

6.3.1 First abstraction

In this first abstraction, we merge the functionality of the two lower layers (the sockets and the buffered sockets) in a transparent way for the common infrastructure. Since we know that in this centralized version the message boundaries are preserved, this architecture sends and receives complete messages, and the steps looking for the special character that separates different messages (see Section 2.8) are skipped. The client socket objects are not used, although we still use their identifiers to simulate them. The situation after applying this abstraction is shown in Figure 6.3: the abstracted buffered sockets offer the

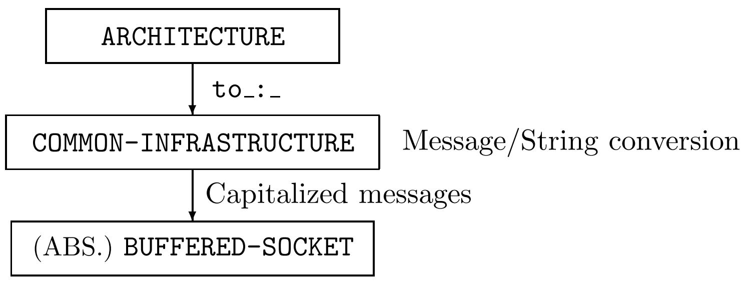

Figure 6.3: Design after the first abstraction

support for the capitalized messages, while the upper layers remain unchanged.

```
omod BUFFERED-SOCKET is
 inc STRING .

 *** Operators and classes from the algebraic sockets
 class Process | conf : Configuration .
 class Manager | count : Nat .
 class ServerSocket | address : String, port : Nat, backlog : Nat .

 op pid : Nat -> Oid [ctor] .
 op socketManager : -> Oid [ctor] .
 op server : Nat -> Oid [ctor] .
 op socket : Nat -> Oid [ctor] .

 *** Messages declarations from BUFFERED-SOCKET
 msg CreateClientTcpSocket : Oid Oid String Nat -> Msg .
 msg CreateServerTcpSocket : Oid Oid Nat Nat -> Msg .
 msg CreatedSocket : Oid Oid Oid -> Msg .
 msg AcceptClient : Oid Oid -> Msg .
 msg AcceptedClient : Oid Oid String Oid -> Msg .
 msg Send : Oid Oid String -> Msg .
 msg Sent : Oid Oid -> Msg .
 msg Receive : Oid Oid -> Msg .
 msg Received : Oid Oid String -> Msg .
 msg CloseSocket : Oid Oid -> Msg .
 msg ClosedSocket : Oid Oid String -> Msg .
```

The rules in the module describe the "merged" behavior of the sockets and the buffered sockets:

- The first one simulates the creation of a socket server, through which clients will be accepted.

```
vars SOCKET PID PID' O O' : Oid .
vars DATA ADDRESS S S' : String .
vars N M PORT BACKLOG : Nat .
vars CONF CONF' : Configuration .

rl [ServerTcpSocketCreation] :
   < PID : Process |
        conf : (CreateServerTcpSocket(socketManager, O, PORT, BACKLOG)
                CONF) >
   < socketManager : Manager | count : N >
=> < PID : Process |
         conf : (CreatedSocket(O, socketManager, server(N)) CONF) >
   < socketManager : Manager | count : s(N) >
   < server(N) : ServerSocket | address : "127.0.0.1", port : PORT,
                                backlog : BACKLOG > .
```

- The second one reproduces the connection of a client with a server. Both client and server receive the new socket identifier, through which messages can be interchanged.

```
  rl [SocketCreation] :
      < PID : Process | conf : (AcceptClient(server(N), O) CONF) >
      < PID' : Process |
        conf : (CreateClientTcpSocket(socketManager, O', ADDRESS, PORT)
               CONF') >
      < socketManager : Manager | count : M >
      < server(N) : ServerSocket | address : ADDRESS, port : PORT >
  =>  < PID : Process |
           conf : (AcceptedClient(O, server(N), "127.0.0.1", socket(M))
                   CONF) >
      < PID' : Process |
            conf : (CreatedSocket(O', socketManager, socket(M)) CONF') >
      < server(N) : ServerSocket | >
      < socketManager : Manager | count : s(M) > .
```

- The third one illustrates how the messages are sent. When we found `Send` and `Receive` messages using the same socket, we transmit the `Data` message by putting the corresponding `Sent` and `Received`.

```
  rl [Send] :
     < PID : Process | conf : (Send(SOCKET, O, DATA) CONF) >
     < PID' : Process | conf : (Receive(SOCKET, O') CONF') >
  => < PID : Process | conf : (Sent(O, SOCKET) CONF) >
     < PID' : Process | conf : (Received(O', SOCKET, DATA) CONF') > .
```

- The last one describes how sockets are closed. The `Send` and `Receive` messages are removed from the configuration.

```
  rl [CloseSocket] :
     < PID : Process | conf : (CloseSocket(SOCKET, O) CONF) >
     < PID' : Process | conf : (Receive(SOCKET, O') CONF') >
  => < PID : Process | conf : (ClosedSocket(O, socketManager, "") CONF) >
     < PID' : Process | conf : (ClosedSocket(O', socketManager, "") CONF') > .
 endom
```

6.3.2 Second abstraction

As we have seen, the messages the architecture transmits are always complete. Now we can suppose that we can transmit data of sort `Msg` through the sockets, so it is not necessary

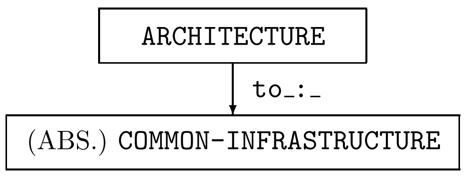

Figure 6.4: Design after the second abstraction

to translate them to and from string. This change merges the common infrastructure layer with the abstracted buffered sockets above. The situation after this abstraction is shown in Figure 6.4: the abstracted common infrastructure offers to each concrete architecture the transmission of `to_:_` messages, while the transformations from and to string are no longer needed.

```
omod COMMON-INFRASTRUCTURE{A :: ARCH-COMPLEMENT} is
 pr ARCHITECTURE-MSGS .
 inc STRING .
 pr MAP{Loc, Oid} .
 pr MAYBE{Oid} * (op maybe to null) .
 pr META-LEVEL .
```

The changes in the operators and classes defined in the abstracted buffered sockets layer are:

- Since the transmission of messages is straightforward, we do not use the `Received` message.

- Since the common infrastructure just removes the `Sent` messages, the transmission rule will not put them into the configuration and there is no need of modeling them.

- The concrete architectures use the `Send` message only once: when the `new-socket` is sent (remember that to use it the function `msg2string` must also be used). Since this abstraction must be transparent to the upper layer, we must deal with this situation. It has been solved by just deleting the `Send` messages from the configuration, and putting the `new-socket` messages directly in the configuration when a client is accepted, as we will see below.

- The `Receive` messages are not only used to represent that an object is waiting for messages, we also use them to simulate sockets between processes: two processes are connected if the socket identifiers kept in the `neighbors` and `defNeighbor` attributes in one process and the first argument of `Receive` messages (the socket through which it waits messages) in other processes are the same.

- We consider that, once they are created, the sockets cannot be destroyed, so we do not provide the `CloseSocket` and `ClosedSocket` messages.

```
*** Classes Process, Manager
class Manager | count : Nat .
class Process | conf : Configuration .

op socketManager : -> Oid [ctor] .
op pid : Nat -> Oid [ctor] .
op socket : Nat -> Oid [ctor] .
op server : Nat -> Oid [ctor] .

*** Messages from BUFFERED-SOCKET
msg CreateClientTcpSocket : Oid Oid String Nat -> Msg .
msg CreateServerTcpSocket : Oid Oid Nat Nat -> Msg .
msg CreatedSocket : Oid Oid Oid -> Msg .
msg AcceptClient : Oid Oid -> Msg .
msg AcceptedClient : Oid Oid String Oid -> Msg .
msg Receive : Oid Oid -> Msg .
msg Send : Oid Oid String -> Msg .

class Router | state : RouterState, neighbors : Map{Loc,Oid},
               defNeighbor : Maybe{Oid}, port : Nat .

msg new-socket : Loc Oid -> Msg [ctor msg] .
op msg2string : Msg -> String .

vars L : Loc .
var  TC : TravelingContents .
var  LSPF : Map{Loc, Oid} .
vars SOCKET PID PID' O O' O'' : Oid .
vars DATA ADDRESS S : String .
vars N M PORT : Nat .
vars CONF CONF' : Configuration .

eq Send(O, O', S) = none .
```

The behavior of the abstracted common infrastructure is defined by the following four rules:

- The first one deals with the creation of server sockets. Although the server socket does not really exist, a new server socket name is given to the object that requested the server creation to allow the object to continue its evolution.

```
rl [ServerTcpSocketCreation] :
   < PID : Process |
      conf : (CreateServerTcpSocket(socketManager, O, PORT, BACKLOG) CONF) >
   < socketManager : Manager | count : N >
=> < PID : Process |
         conf : (CreatedSocket(O, socketManager, server(N)) CONF) >
   < socketManager : Manager | count : s(N) > .
```

- The second one simulates the connection between a server and a client. A new socket name is generated and communicated to both client and server, although no socket is put into the configuration. The messages `new-socket` are directly interchanged, and each concrete topology will deal with them.

```
rl [SocketCreation] :
   < PID : Process | conf : (AcceptClient(SOCKET, O)
                        < l(ADDRESS, N) : Router | port : PORT > CONF) >
   < PID' : Process |
      conf : (CreateClientTcpSocket(socketManager, L, ADDRESS, PORT)
              < L : Router | > CONF') >
   < socketManager : Manager | count : M >
=> < PID : Process | conf : (new-socket(L, socket(M))
            AcceptClient(SOCKET, O)
            AcceptedClient(l(ADDRESS, N), SOCKET, "127.0.0.1", socket(M))
            < l(ADDRESS, N) : Router | port : PORT > CONF) >
   < PID' : Process | conf : (new-socket(l(ADDRESS, N), socket(M))
                  CreatedSocket(L, socketManager, socket(M))
                  < L : Router | > CONF') >
   < socketManager : Manager | count : s(M) > .
```

- The other two rules redirect the messages through the corresponding socket, using the `neighbor` and `defNeighbor` attributes: we look for the process that is receiving through the appropriate socket.

```
    crl [redirectDef] :
        < PID : Process |
          conf : (CONF to O : TC
                  < L : Router | neighbors : LSPF, defNeighbor : SOCKET >) >
        < PID' : Process | conf : (Receive(SOCKET, O') CONF') >
     => < PID : Process |
          conf : (CONF < L : Router | neighbors : LSPF,
                                      defNeighbor : SOCKET >) >
        < PID' : Process | conf : ((to O : TC) Receive(SOCKET, O') CONF') >
     if getLoc(O) =/= L /\ LSPF[getLoc(O)] == undefined .

    crl [redirect] :
        < PID : Process |
          conf : (to O : TC < L : Router | neighbors : LSPF > CONF) >
        < PID' : Process | conf : (Receive(SOCKET, O') CONF') >
     => < PID : Process |
          conf : (< L : Router | neighbors : LSPF > CONF) >
        < PID' : Process | conf : ((to O : TC) Receive(SOCKET, O') CONF') >
     if getLoc(O) =/= L /\ SOCKET := LSPF[getLoc(O)] .
  endom
```

6.3.3 Third abstraction

So far, we trust in message delivery. Then, why not *trust the whole architecture*, instead of considering all the steps through different processes to deliver each message? We can suppose that when a message must get to some object, and this object resides in other process, it can be delivered immediately. Thus, we can finally define all the architectures with one single rule.

```
omod ARCHITECTURE is
 pr ARCHITECTURE-MSGS .
 pr STRING .

 class Process | conf : Configuration .

 var  TC : TravelingContents .
 vars PID PID' O : Oid .
 var  C : Cid .
```

```
  vars CONF CONF' : Configuration .

  rl [send] :
     < PID : Process |  conf : (to O : TC CONF) >
     < PID' : Process | conf : (< O : C | > CONF') >
  => < PID : Process | conf : CONF >
     < PID' : Process | conf : ((to O : TC) < O : C | >  CONF') > .
 endom
```

Notice that this abstraction requires some changes in the initial configurations: the objects related with architectures must be removed, as well as the socket manager from the algebraic sockets, and it is not parameterized.

6.4 State space reduction

Although model checking is one of the most successful automated verification techniques, there are real limitations to its applicability in practice. These limitations are mostly related to the state space explosion problem. This can make it unfeasible to model check a system except for very small state spaces, sometimes not even for those. For this reason, a host of techniques to tame the state space explosion problem, which could be collectively described as *state space reduction* techniques, have been investigated. These techniques are also useful to reduce the state space when using the `search` command.

We use a reduction technique based on the idea of *invisible transitions* [32], that generalize a similar notion in Partial Order Reduction (POR) techniques. Given the rewrite theory $\mathcal{R} = (\Sigma, E, R)$, a rewrite rule r in R is called P-invisible if in any rewrite step $[t] \to [t']$ using r the states $[t]$ and $[t']$ satisfy the *same* state predicates in P. We can identify the set R' of rules that satisfy this property and obtain a new rewrite theory $\mathcal{R}' = (\Sigma, E \cup R', R \setminus R')$. However, to obtain a correct reduction more properties must hold: the new equation set $E \cup R'$ must be *terminating* and *confluent* [1] (perhaps modulo some axioms A), and the rewrite theory must be strongly *coherent* [66].

To check the termination property, we look for a function that strictly decreases with the application of each rule in R'. Confluence is a property of term rewriting systems, describing that terms in this system can be rewritten in more than one way, to yield the same result. To check this property, we must examine that for each pair of rules (a *critical pair*) in R' that can be applied to a term, the same result is produced (it is *joinable*).

Intuitively, the coherence requirement means that we can identify a state $[t]$ with the canonical form $can_E(t)$ of t by the equations E, and that rewriting with equations E and with rules R commutes in an appropriate sense, so that we can safely restrict our computations with R to only rewrite E-canonical forms. To check this property, we study the critical pairs between rules in $R \setminus R'$ and equations in $E \cup R'$.

Thus, to apply partial order reduction techniques it is important to know what properties about the specified systems we want to prove. We show in the following sections the set of rules of each application that can be converted into equations, and a little sketch of the justification of its correctness is given.

Although the first implementation of the algebraic sockets was done using rules (see Section 6.2), we always use a reduced version of them when using a centralized application, where all the rules have been transformed into equations. That is because the properties we check should not depend on the sockets, because they are a built-in Maude feature which we cannot handle, so reduce them is always correct. The architectures reduced in the following sections are the ones shown in Chapter 3, not their abstracted versions from Section 6.3, although of course both ideas can be combined to reduce the state space.

6.4.1 Partial order reduction in the architectures

The properties we want to prove about the architectures in Section 6.5 have the form "a message M reaches the location L". We consider the set R' of rules that are not converted into equations composed by all the rules in the concrete architecture, while the rules from the common architecture remain unchanged. We select these rules not only because of the P-invisibility property, but also due to termination: the redirection rules can cause non-termination. We will also use this reduction when a concrete application is used on top of one architecture.

6.4.2 Partial order reduction in the skeletons

The properties we want to check in the skeletons in Section 6.6 refer to the final result obtained in the master, thus the rules modifying it cannot be converted into equations. Therefore, the set R' of rules that can be converted into equations is composed by all the rules of the skeleton (together with the rules of the architecture, as shown above) except for those that change the master's attribute where the result is kept:

- `new-work` and `no-more-work` in the farm skeleton;
- `completed` and `working` in the systolic ring skeleton;
- `new-work` and `no-more-work` in the divide and conquer skeleton; and
- `partial-results` in the branch and bound skeleton.

To prove termination we can see that the reduced rules only guide the messages between the master and the workers. These messages are delivered by the architecture (that, as we have seen, is terminating), and once all the tasks have been computed by the workers, the master cannot generate new ones with the rules in R'. To check the confluence property, we can see that the rules in R' are applied in a sequential fashion: the master sends works that the workers compute and send back. These results can arrive unordered (thus the problems can also be delivered unordered) because in the farm skeleton the theory requires it, in the systolic ring skeleton the results are numbered, in the divide and conquer skeleton they are inserted in the corresponding leaves, and in the branch and bound skeleton they are inserted as nodes in the priority queue.

Finally, the rules in R' cannot violate the coherence property, because the rules not in R' only combine the results and send new problems, that, as we have outlined before, can be done in any order.

6.4.3 Partial order reduction in Mobile Maude

In Section 6.7 we want to check properties about the left component of the state of the inner objects (remember that it has the form `C & C'`, see Section 5.3) of mobile objects that are not traveling. Therefore, the problematic rules are:

- The rule `do-something`, that makes the inner objects evolve.

- The rules `create-object` and `message-out-kill`, that create and destroy objects.
- The rules `arrive-loc` and `arrive-find-loc`, that handle the movement of objects. Notice that these rules are in charge of redirecting the `go` and `go-find` messages. Keeping them as rules is important because they could generate confluence and coherence problems. Suppose that an object `O` is looking for another one `O'` in location `L`, that want to travel to a new location `L'`. If `O` finds `O'` in `L`, and then `O'` goes to `L'`, we obtain a different configuration that if `O'` travels before `O` finds it.
- The rules `message-out-go` and `message-out-go-find`, that start the movement of objects.
- The rule `msg-in`, that pushes into the configuration a new inter-object message.

Notice that the rules `arrive-loc` and `arrive-find-loc` are in charge of redirecting the `go` and `go-find` messages. This is important because these rules could generate confluence and coherence problems. Suppose that an object `O` is looking for another one `O'` in location `L`, that want to travel to a new location `L'`. If `O` finds `O'` in `L`, and then `O'` goes to `L'`, we obtain a different configuration that if `O'` travels before `O` finds it.

All the rules not shown above, that is, the rules for redirection of inter-object messages and forwarding information and the rules to update the forwarding information, form the set R'.

It is easy to see that the equations are terminating, because the rules that define the movement of objects are not in R', so the inter-object messages always reach their destination, as well as the forwarding messages, that must find a home object, that cannot travel.

The rules in R' are in charge of the delivery of inter-objects and forwarding messages. Since the order in which messages arrive is immaterial, because the inter-object messages are pushed into by using rules not in R' and the forwarding messages have a sequence number (the number of hops), the system is confluent.

The rules not in R' are in charge of pushing (in and out) messages from the inner state, while the rules in R' redirect some of these messages and update the forwarding information. The transformation of the rules in R' into equations only "speeds up" this actions, but it does not affect the reachable states, so the new system is coherent.

6.5 Verifying architectures

Architectures have been designed independently from the concrete applications, and this allows to check properties over them in an isolated way. We show here some simple properties of the centralized ring architecture. Other properties on different architectures can be proved using the same methodology.

6.5.1 Using the model checker

We want to check in the centralized ring what happens if a node in the ring sends a message to another one also in the ring. To study it we use an initial configuration with one of the locations in the ring with an object and another one with a message for it. Some of the nodes will be traversed by the message and others will be never traversed (at least the center).

First, we must define the view to instantiate the architecture, with a constant `CONT` of sort `TravelingContents` to use in the initial configuration.

```
fmod CR-ARCHITECTURE-MESSAGES is
 pr ARCHITECTURE-MSGS .
 pr META-LEVEL .

 op CONT : -> TravelingContents .
endfm

view CR-Complement from ARCH-COMPLEMENT to CR-ARCHITECTURE-MESSAGES is
 op MOD to term upModule('CR-ARCHITECTURE-MESSAGES, false) .
endv
```

We define the property `have-message`, that checks if a given location contains a message, and we declare `Configuration` as the sort of states. Note that there is no difference with the distributed version, we must only load the "centralized" `SOCKET` module instead of the predefined `SOCKET` module.

```
omod MODEL-CHECK is
 pr STAR-CENTER{CR-Complement} .
 pr CENTRALIZED-RING-NODE{CR-Complement} .
 pr CENTRALIZED-RING-LAST{CR-Complement} .
 pr EXT-BOOL .
 inc SATISFACTION .
 inc MODEL-CHECKER .
 inc LTL-SIMPLIFIER .

 subsort Configuration < State .

 op have-message : Loc -> Prop .
```

We use as process identifier the name of the router contained there in order to find the desired process faster.

```
 vars C C' : Configuration .
 var  O : Oid .
 var  L : Loc .

 eq C < L : Process | conf : (C' to O : CONT) > |= have-message(L) = true .
 eq C |= have-message(L) = false [owise] .
```

Now we must define the LTL formulas specifying the properties. The formula `F` expresses that the location `L` receives a message exactly once, and then redirects it.

```
 ops F F' F'' : Loc -> Formula .
 eq F(L) = ~ have-message(L) U
          (have-message(L) /\ (have-message(L) U [] ~ have-message(L))) .
```

The formula `F'` states that `L` never contains a message. Therefore `F` ∨ `F'` states that `L` receives a message at most once, and then redirects it.

```
 eq F'(L) = [] ~ have-message(L) .
```

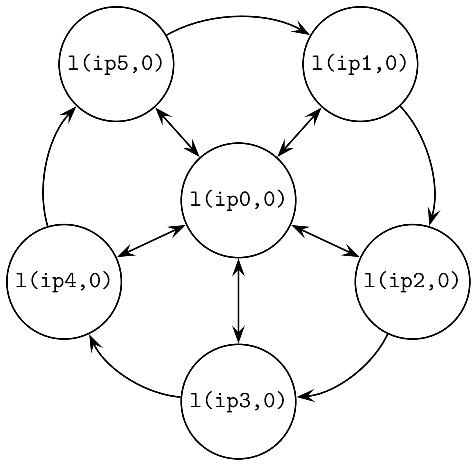

Figure 6.5: Centralized ring configuration

Finally, `F''` states that a message reaches `L` and stays there forever.

```
  eq F''(L) = ~ have-message(L) U ([] have-message(L)) .
endom
```

We check this property in the configuration shown in Figure 6.5 with five nodes in the ring (`l(ipi, 0)`, $i \in 1 \ldots 5$), and a message from `l(ip4, 0)` to an object in the location `l(ip2, 0)`, so it must traverse `l(ip5, 0)` and `l(ip1, 0)`. The center (`l(ip0, 0)`) and `l(ip3, 0)` must receive no messages. Therefore we use the following command:

```
red modelCheck(initial, F(l(ip4, 0)) /\ F(l(ip5, 0)) /\ F(l(ip1, 0)) /\
                        F'(l(ip0, 0)) /\ F'(l(ip3, 0)) /\ F''(l(ip2, 0)) .
```

We obtain that the property holds for all the locations.

```
rewrites: 6273 in 250ms cpu (268ms real) (25092 rewrites/second)
result Bool: true
```

6.5.2 Using the search command

We can check now that the connection between each node in the ring and the center is direct. To do it, we declare an initial configuration where the invariant will be checked. We place an object in the center and a message for it in one of the nodes of the ring. We consider as an invariant the property `messages-invariant`, that states that all the nodes in the ring (except the one sending the message) never contain a message in their configuration.

```
omod SEARCH is
  pr STAR-CENTER{CR-Complement} .
  pr CENTRALIZED-RING-NODE{CR-Complement} .
  pr CENTRALIZED-RING-LAST{CR-Complement} .
  pr EXT-BOOL .

  op messages-invariant : Loc Configuration -> Bool .
```

where Loc indicates the location that sends the message. We define the invariant in a similar way to the properties for model checking.

```
  vars C C' : Configuration .
  var  PID : Oid .
  vars L L' : Loc .
  var  O : Oid .

  ceq messages-invariant(L, C < L' : Process | conf : (C'
            < L' : CRingRouter | > to O : CONT) >) = false
   if L =/= L' .
  eq messages-invariant(L, C) = true [owise] .
 endom
```

The command to check the invariant is:

```
 search initial =>* C:Configuration
   such that not messages-invariant(l(ip4, 0), C:Configuration) .

 No solution.
```

where initial is a configuration with five nodes in the ring, and having the node l(ip4, 0) a message for an object in the center. We obtain that no solution is found, so the invariant is always true and the property holds.

So far we have proved that the message does not traverse the other nodes of the ring, but we must also check that the message disappears from the initial ring node and arrives to the center. We prove it by using the predicate final-conf, that checks if the message has disappeared of the ring node and appeared in the center.

```
 op final-conf : Loc Configuration -> Bool .
 eq final-conf(L, C < PID : Process | conf : C' >) = final-conf(L, C)
                                  and-then final-conf(L, C') .
 eq final-conf(L, C < L' : StarCenter | >) = have-message(C) .
 eq final-conf(L, C < L : CRingRouter | >) = not have-message(C) .
 eq final-conf(L, C) = true [owise] .

 op have-message : Configuration -> Bool .
 eq have-message(C (to O : CONT)) = true .
 eq have-message(C) = false [owise] .
```

The search command does not need to check all the states now, but only the final ones, so we use =>!.

```
 search initial =>! C:Configuration
  such that not final-conf(l(ip4, 0), C:Configuration) .

 No solution.
```

No solution is found, so we can conclude that the connection between a node in the ring and the center is direct.

6.6 Verifying skeletons

In the skeleton instantiations, we can consider the sequential versions the *specification* of the problem and the distributed versions the *implementation*. We show how to use the search command to verify that all the final solutions found in the distributed applications are the same that the solutions from the specifications.

We will define for each skeleton a getResult operation that, given a configuration, returns the result kept in the master. We use it to compare the results from the sequential and the distributed implementation. Notice that the comparison can be non trivial, as we will see in the examples.

6.6.1 Euler numbers

We must verify that each execution of the skeleton instantiated with the Euler numbers problem returns as result the same number that the sequential version. In that example, getResult returns a natural number, and we have to compare it exactly with the result from the specification.

```
 op getResult : Configuration -> P$Result .
 eq getResult(C < O : Process | conf : (
             C' < M : RW-Master | result : R >) >) = R .
```

The search command that must be used is:

```
search initial(7) =>! C:Configuration
  such that getResult(C:Configuration) =/= sumEuler(7) .
```

where initial is a configuration that receives as parameter the Euler number we are looking for.

As expected, no different results are found:

```
No solution.
states: 766  rewrites: 465030 in 12780ms cpu (13257ms real)
```

6.6.2 Ray tracing

In this example the comparison is harder than the last one. In the distributed version we keep the screen rows in a map (that will be returned by getResult), because the rows could arrive unordered, while in the sequential version the rows are returned with the juxtaposition operator. The results cannot be compared directly, so we define an operator map2screen that transforms the map in a screen, and then the results can be checked.

```
 fmod MAP2SCREEN is
  vars CL CL' : ColorList .
  var  S : Screen .
  var  M : Map{Float, ColorList} .
  vars F F' : Float .

  op map2screen : Map{Float, ColorList} -> Screen .
  op map2screen : Map{Float, ColorList} Screen -> Screen .
  op max : Map{Float, ColorList} -> Float .
  op delete : Float Map{Float, ColorList} -> Map{Float, ColorList} .

  eq max(F |-> CL) = F .
  ceq max((F |-> CL, M)) = max(F, max(M))
   if M =/= empty .

  eq delete(F, (F |-> CL, M)) = M .
  eq delete(F, M) = M [owise] .

  eq map2screen(F |-> CL) = [CL] .
  eq map2screen(M) = map2screen(delete(max(M), M), [M [max(M)]]) [owise] .
  eq map2screen(F |-> CL, S) = S [CL] .
  eq map2screen(M, S) =
          map2screen(delete(max(M), M), S [M [max(M)]]) [owise] .
  op initial : Nat Nat Nat Nat Nat Nat FigureList -> Configuration .
  eq initial = ...
 endfm
```

The command to check the skeleton is:

```
search initial(-10, 10, 3, -3, 10, 1000000000, figListN(10)) =>! C:Configuration
  such that map2screen(getResult(C:Configuration)) =/=
          rayTracing(-10, 10, 3, -3, 10, 1000000000, figListN(10)) .
```

As expected, no results are found and the search ends successfully:

```
No solution.
states: 127  rewrites: 80774 in 880ms cpu (913ms real)
```

6.6.3 Force interactions

In this example we have a new problem when comparing the results from the sequential and the distributed version. Although both of they are floats, there are a lot of operations involved with numbers really small (about 10^{-24}), so the results may vary because of floats precision. We must check that the difference between the results are not greater than ε, a given error constant. The `getResult` function for this skeleton is defined as follows:

```
  op getResult : Configuration -> P$Result .
  eq getResult(C < O : Process | conf : (
             C' < M : SMaster | result : R >) >) = R .
```

We use the command

```
 search initial(atomGenerator(12)) =>! C:Configuration
  such that abs(getResult(C:Configuration) - attraction(atomGenerator(12)))
  > 1.0e-20 .
```

The implementation is correct and the search does not find solutions:

```
 No solution.
 states: 23  rewrites: 26487 in 60ms cpu (223ms real)
```

6.6.4 Mergesort

We use the same method to prove properties over task parallel programs. In the mergesort instantiation, we keep a `List` as result, that is a container for lists of natural numbers. The function `getResult` extracts the result from the tree:

```
op getResult : Configuration -> P$Result .
eq getResult(C < O : Process | conf : (
             C' < M : DCMaster | resultTree : tree(R, F) >) >) = R .
```

We only need to transform the `NatList` from the sequential version in a `List` using the operator `l`:

```
search initial(gen(1000)) =>! C:Configuration
 such that getResult(C:Configuration) =/= l(mergesort(gen(1000))) .
```

where `initial` is the initial configuration that receives as parameter the list generated by `gen`. The result confirms that the distributed version was properly implemented:

```
No solution.
states: 64  rewrites: 2239374 in 11200ms cpu (15315ms real)
```

In this case, the postcondition of the problem is simple enough to avoid the use of the sequential implementation. We can define an `ordered` function that checks if a list is sorted and it has the same components that other list.

```
op ordered : List NatList -> Bool .
eq ordered(l(N), N) = true .
ceq ordered(l(N N' NL), NL' N NL'') = ordered(l(N' NL), NL' NL'')
 if N <= N' .
eq ordered(L, NL) = false [owise] .
```

We can use it now in the `search` command:

```
search initial(gen(1000)) =>! C:Configuration
  such that not ordered(getResult(C:Configuration), gen(1000)) .
```

obtaining the same result with less rewrites:

```
No solution.
states: 64  rewrites: 733875 in 1140ms cpu (1308ms real)
```

6.6.5 Traveling salesman problem

In the sequential version of this problem, we keep a tuple with the path and its cost, the same that in the distributed version. We define an `equal` function that compares the two tuples

```
op equal : Node TravelResult -> Bool .
eq equal(node(P, N), result(P', N')) = P == P' and N == N' .
```

and use it in the condition of the search:

```
search initial(city(0), 6, generateCostMatrix(6)) =>! C:Configuration
 such that not equal(getResult(C:Configuration),
                  travel(city(0), 6, generateCostMatrix(6))) .
```

where `initial` is the initial configuration, that receives the initial city, the number of cities, and the cost matrix as parameter. We obtain a positive answer:

```
No solution.
states: 4 rewrites: 1539479 in 11180ms cpu (11643ms real)
```

6.6.6 Model checking skeletons

The properties P we have proved so far are defined over final configurations, so we can express them in Linear Temporal Logic as $\Diamond\ \Box\ P$ and check them with Maude model checker. However, the model checker needs more work to verify the property (and define the property is harder too), so it is better to check that kind of properties with `search`. Just as illustration, we show how to verify the Euler numbers example with the model checker.

Euler numbers

First, we create a module defining the sort of states as `Configuration` and the property we want to check.

```
mod EULER-CHECK is
 pr EULER-EXAMPLE .
 inc SATISFACTION .
 inc LTL-SIMPLIFIER .
 inc MODEL-CHECKER .

 subsort Configuration < State .
 op result : Nat -> Prop .
```

Then we define when the property holds by means of equations.

```
 vars C C' : Configuration .
 vars PID O : Oid .
 var  N : Nat .

 eq (C < PID : Process | conf : C' >) |= result(N) =
     if (C =/= none) then
       ((C |= result(N)) or (C' |= result(N)))
     else
       (C' |= result(N))
     fi .
 eq (C < O : RW-Master | result : N >) |= result(N) = true .
 eq C |= result(N) = false [owise] .

 op initial : Nat -> Configuration .
 eq initial(N) = ...
endm
```

where `initial` is a configuration that receives as parameter the Euler number we are looking for.

Once we have that module, we can use the model checker to verify the property:

```
red modelCheck(initial(7), <> [] result(sumEuler(7))) .
```

The model checker finishes with more rewrites than the search command, as expected:

```
rewrites: 483993 in 12760ms cpu (13710ms real) (37930 rewrites/second)
result Bool: true
```

6.7 Verifying Mobile Maude

Mobile Maude applications present a new problem: we would like to check properties on the application code, which is metarepresented in the belly of the mobile objects. That problem has been solved by considering two-level properties, stating different properties on each of the reflection levels:

- In the mobile objects level, we look for objects that could fulfill the properties.
- In the application level, we check the properties.

We show in the following sections two examples of formal analysis of properties about Mobile Maude applications. The first one uses the `search` command, while the second one uses the model checker.

6.7.1 Two-level atomic propositions for the buying printers example

Let us see an example about the buying printers case study shown in Section 5.6. Suppose we want to prove that the buyer always finds the best price, and that, when he has visited all sellers, he finishes in the process of the seller who has such a best price. If *bestPrice&Seller* represents the predicate asserting that the buyer is in the process of the seller with the best offer, then we can use the `search` command to check that there is no final state that fulfills the negation of this predicate.

The module specifying the properties at the mobile objects level imports the modules `MOBILE-MAUDE-SYNTAX` and `SOCKET`, that provides the syntax for mobile objects and processes. First, we define when a top configuration of processes fulfills such a property. For it, we use an auxiliary predicate *bestPrice&Seller* with another argument, (the metarepresentation of) the best price, obtained by means of the auxiliary function `minPrice`.

```
 mod PRINTERS-PREDS is
  pr MOBILE-MAUDE-SYNTAX .
  pr SOCKET .

  op bestPrice&Seller : Configuration Term -> Bool .
  op bestPrice&Seller : Configuration -> Bool .

  op minPrice : Configuration -> Term .
  op minimum : Term Term -> Term .
  vars PID O : Oid .
  var  MODULE : Module .
  vars N TERM TERM' A T T' : Term .
  vars C C' : Configuration .

  eq bestPrice&Seller(C) = bestPrice&Seller(C, minPrice(C)) .
```

`minPrice` is defined as follows, where `price` is applied by using the `reduce` function, that uses the metalevel function `metaReduce` to reduce it in the module `PRINTERS-INNER-PREDS` described below.

```
  eq minPrice(C < PID : Process | conf : C' >) =
    if (C =/= none) then
      minimum(minPrice(C), minPrice(C'))
    else
      minPrice(C')
    fi .

  op reduce : Term -> Term .
  eq reduce(T) = getTerm(metaReduce(upModule('PRINTERS-INNER-PREDS, false), T)) .

  eq minPrice(< O : MobileObject | s : ('_&_[TERM, TERM']) > C) =
    if (C =/= none) then
      minimum(minPrice(C), reduce('price[TERM]))
    else
      reduce('price[TERM])
    fi .

  eq minPrice(C) = upTerm(999999) [owise] .
```

The definition of `bestPrice&Seller(C, N)` recursively traverses all the processes in `C` going inside each configuration looking for a seller with the given price and a buyer who has it as the best price.

```
  op existsSeller : Configuration Term -> Bool .
  op existsBuyer : Configuration Term -> Bool .

  eq bestPrice&Seller(C < PID : Process | conf : C' >, N) =
      bestPrice&Seller(C, N) or
      (existsSeller(C', N) and existsBuyer(C', N)) .
  eq bestPrice&Seller(C, N) = false [owise] .

  eq existsSeller(< O : MobileObject | s : ('_&_[TERM, TERM']) > C, N) =
       (reduce('exSeller[TERM, N]) == 'true.Bool) or
       existsSeller(C, N) .
  eq existsSeller(C, N) = false [owise] .

  eq existsBuyer(< O : MobileObject | s : ('_&_[TERM, TERM']), AtS > C, N) =
       (reduce('exBuyer[TERM, N]) == 'true.Bool) or
       existsBuyer(C, N) .
  eq existsBuyer(C, N) = false [owise] .
 endm
```

The definition of `existsSeller(C, N)` uses the function `exSeller` defined at the inner objects level. The predicate `existsBuyer(C, N)` is defined in the same way. The module

PRINTERS-INNER-PREDS includes the definition of the predicates exSeller and exBuyer. Note that the buyer's state is relevant because the property must hold when the buyer has visited all the sellers, that is, it is in buying state.

```
mod PRINTERS-INNER-PREDS is
 pr BUYER .
 pr SELLER .

 op exSeller : Configuration Nat -> Bool .
 op exBuyer : Configuration Nat -> Bool .
 op price : Configuration -> Nat .

 vars B S : Oid .
 var  C : Configuration .
 var  N : Nat .

 eq exBuyer(< B : Buyer | price : N, status : buying > C, N) = true .
 eq exBuyer(C, N) = false [owise] .

 eq exSeller(< S : Seller | description : N > C, N) = true .
 eq exSeller(C, N) = false [owise] .
```

Notice that these atomic propositions are defined at the level of the application code. It also defines the function price, used to calculate the minimum price kept in the objects of the configuration.

```
 eq price(< S : Seller | description : N > C) = N .

 eq price(C) = 999999 [owise] .
endm
```

After having defined these predicates, the Maude search command is as follows:

```
mod PRINTERS-CHECK is
 inc PRINTERS-PREDS .
 pr BUYER .
 pr SELLER .
 pr STAR-CENTER{MM-Complement} .
 pr STAR-NODE{MM-Complement} .
 pr MOBILE-MAUDE-SEMANTICS .

 op initial : -> Configuration .
 eq initial = ...
endm

search initial =>! C:Configuration s.t. not bestPrice&Seller(C:Configuration) .
No solution.
```

where initial is the initial configuration shown in Section 5.6.

6.7.2 Model checking the auctions

In the auctions example, there is a lot of non-determinism, and properties like "the client C eventually buys something" may be false because LTL implicitly involves *all the paths*. To prove properties like that, we must use the model checker and look for a counterexample of the negation of the property. The intended sort of states that we will use when model checking Mobile Maude applications is `Configuration`.

For example, suppose we have a system with two clients, `C1` and `C2`, each with money `10` and `12`, and one auction center with one item `I1`. The client `C1` can get `I1` if his agent offers more money than `C2`'s agent. But if `C2`'s agent offers more money, he will obtain the item. Consider $P(x)$ the property "x buys something", the property $\Diamond P(x)$ is false for both `C1` and `C2`, because some paths do not fulfill it. Let us see the problem with the model checker, first we define the property at the application level:

```
mod AUCTION-INNER-PREDS is
  pr AUCTION .
  inc SATISFACTION .

  subsort Configuration < State .
  op getsSomething : -> Prop .

  eq < B : Buyer | bought : IS > C |= getsSomething  = IS =/= noItem .
endm
```

Now we traverse the configuration at the processes and mobile objects level. The difference with the printers example is that now the name of the object is known, so we do not need to check all the objects. First we look inside the processes.

```
 mod AUCTION-PREDS is
  pr MOBILE-MAUDE-SEMANTICS .
  inc LTL-SIMPLIFIER .
  pr AUCTION-INNER-PREDS .
  pr MODEL-CHECKER .

  vars PID O O' : Oid .
  var  MODULE : Module .
  vars TERM TERM' T : Term .
  vars C C' : Configuration .

  op getsSomething : Oid -> Prop .

  eq (C < PID : Process | conf : C' >) |= getsSomething(O) =
      if (C =/= none) then
        (C |= getsSomething(O)) or (C' |= getsSomething(O))
      else (C' |= getsSomething(O))
      fi .
```

In the multiset of objects in the configuration of the processes, we select the concrete object we are looking for, if possible.

```
  op reduce : Term -> Term .
  eq reduce(T) = getTerm(metaReduce(upModule('AUCTION-INNER-PREDS, false), T)) .

  eq (< O : MobileObject | s : ('_&_[TERM, TERM']) > C) |=
    getsSomething(O) = reduce('_|=_[TERM, 'getsSomething.Prop]) == 'true.Bool .

  eq C |= getsSomething(O) = false [owise] .
 endm
```

If now we try to prove $\Diamond$ `getsSomething(C1)`, it returns a counterexample where `C1` does not obtain the item, which informs us that in some paths `C1` obtains nothing, but we do not know if there is a path where it buys something. We can try now to prove $\neg \Diamond$ `getsSomething(C1)` (or $\Box \neg$ `getsSomething(C1)`), and we get a counterexample with a final configuration where `C1` has bought the item.

Some other similar properties can be defined in this example, like "x buys all the items he wants", that must be defined in the same way.

Chapter 7

Conclusions

We consider this work part of an ongoing project where several distributed applications will be implemented in Maude and then formally analyzed, trying to work in each step in a more complex task, as well as improving the previous stages. The next objective in this project is to analyze an IP routing protocol, the Enhanced Interior Gateway Routing Protocol (EIGRP) [15].

In Chapter 3, we have shown how to implement diverse architectures (that is, a star, a ring, and a centralized ring network) by using the object oriented features of Maude, that allow us to share most of the components by means of inheritance, and the sockets that Maude supports as external objects, in such a way that they are transparent to the concrete applications executed on top of them. This is possible because they are parameterized, and receive as parameter the information needed to deliver the messages. This is the first really distributed application implemented in Maude, and supports the distribution of the rest of applications.

However, the shown architectures are static, in the sense that exactly all the nodes specified in the initial configurations must form the architecture, except for the star one, where star nodes can join at any time. We are going to develop dynamic, reconfigurable topologies, where nodes can join and leave, by using the EIGRP. Since this kind of protocols requires timeouts, we are going to use sockets in a new way: to connect Maude with an external Java "clock".

We have presented in Chapter 4 the implementation of several skeletons (namely data-parallel such as the farm skeleton; systolic such as the ring skeleton; and task-parallel such as the divide and conquer, the branch and bound, and the pipeline skeletons) as parameterized modules that receive as parameter the operations solving each concrete problem. This allows us to instantiate the same skeleton for a concrete problem in different ways, for example varying its granularity.

Thus we have described a methodology to specify, prototype, and check skeletons that can be later implemented in other languages such as Java in order to obtain a better speed-up and more "commercial" implementations. In the same way, we can translate Java skeletons to check properties over them (we plan to study in the future which is the best way to achieve both translations).

We have tested the skeletons with several examples, using three 2 GHz PowerPC G5 and two 1.25 GHz PowerPC G4, obtaining an average speed-up of 2.5. Although this speed-up is not remarkable, we observed in the executions that all the processors were always busy, so most of the time was used in manipulating the transmitted data. We have to study how to improve the efficiency; the profiling feature in Maude allows a detailed analysis of which rules are most expensive to execute in a given application. We are now also studying how our skeletons can be nested and combined by using the object-oriented inheritance features provided by Maude.

We have presented in Chapter 5 a distributed implementation of Mobile Maude where mobile objects, carrying its own code and internal state, can travel from one machine to another one. We have used the language to implement several case studies. One of them is the implementation of algorithmic skeletons by using generic mobile objects; although the same generality that in the parameterized case was obtained, the main drawback was lack of efficiency due to the reflection levels introduced. The conference reviewing system, the example described by Cardelli as a challenge for any wide area language to demonstrate its usability and whose Mobile Maude implementation was presented in [30], has also been migrated to this new version of the language. This distributed implementation of Mobile Maude has advanced one more step in the path started in [28], where a simulator of Mobile Maude in Maude 1.0.5 was presented. We plan to continue this work with an extension of Mobile Maude with security features.

Finally, we have shown in Chapter 6 how to formally analyze the applications shown throughout the work. We show how the Maude sockets can be simulated in order to represent a "centralized" configuration. Once the sockets have been simulated, we explain how the architectures can be abstracted by incrementally removing the "layers" that compose them. We also describe how partial order reduction can be applied to the specifications shown in the previous chapters, in order to reduce the state space explosion problem usually found when model checking systems. The Maude tools that we use once we have applied the techniques above are the `search` command, that explore (following a breadth-first strategy) the reachable state space, allowing us to check invariants, and the model checker, that examine if some properties, expressed in Linear Temporal Logic, are fulfilled by the systems.

Although we have proved relevant properties over these specifications, such as invariants and correspondence between specification and implementation, an inconvenience of these tools is that they prove properties starting in an initial state, that is, we prove properties about concrete configurations. We are going to study how to apply rule induction on Maude specifications in order to prove general properties about them.

Some problems were found while developing these applications. Since TCP sockets are the first (and the unique, so far) external objects supported by Maude, their behavior is so general that they are not so indicated for applications where the efficiency is critical, because all the data must be converted to string before being transmitted, and translated again when received. A new type of socket able to transmit messages (of sort `Msg`) instead of strings could increase the speed-up obtained by the skeletons substantially. Moreover, some problems were also detected when extremely large strings were transmitted through sockets, resulting in the break of the connection. We hope that the results obtained in this work will help also to improve the Maude system.

Bibliography

[1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[2] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A structured high-level parallel language, and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[3] G. H. Botorog and H. Kuchen. Efficient parallel programming with algorithmic skeletons. In Bouge et al. [4], pages 718–731.

[4] L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors. *Euro-Par '96 Parallel Processing: Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings*, volume 1123 of *Lecture Notes in Computer Science*. Springer, 1996.

[5] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[6] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Eden – The paradise of functional concurrent programming. In Bouge et al. [4], pages 710–713.

[7] R. Bruni and J. Meseguer. Generalized rewrite theories. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Automata, Languages and Programming. 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003, Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2003.

[8] S. R. Buss. *3-D Computer Graphics*. Cambridge University Press, 2003.

[9] L. Caires and L. Cardelli. Spatial logic for concurrency (Part I). In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 2001, Proceedings*, volume 2215 of *Lecture Notes in Computer Science*, pages 1–37. Springer, 2001.

[10] L. Caires and L. Cardelli. Spatial logic for concurrency (Part II). In P. J. L. Brim, M. Ketínský, and A. Kuera, editors, *CONCUR 2002 – Concurrency Theory 13th International Conference, Brno, Czech Republic, August 20-23, 2002. Proceedings*, volume 2421 of *Lecture Notes in Computer Science*, pages 209–225. Springer, 2002.

[11] L. Cardelli. Obliq - A language with distributed scope. Technical report, Systems Research Center, 1994.

[12] L. Cardelli. Abstractions for mobile computations. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 51–94. Springer, 1999.

[13] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *Foundations of Software Science and Computation Structures, First International Conference, FoSSaCS'98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98 Lisbon, Portugal, March 28–April 4, 1998 Proceedings*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 1998.

[14] G. Castagna, J. Vitek, and F. Zappa Nardelli. The Seal calculus. *Information and Computation*, 201:1–54, 2005.

[15] Cisco. *White papers - Enhanced Interior Gateway Routing Protocol*. http://www.cisco.com/warp/public/103/eigrp-toc.html.

[16] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[17] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[18] 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.

[19] 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/maude2-manual.

[20] M. Cole. *Algorithmic Skeletons: Structure Management of Parallel Computations*. MIT Press, 1989.

[21] M. Danelutto. QoS in parallel programming through application managers. In *PDP'05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05)*, pages 282–289. IEEE Computer Society, 2005.

[22] M. Danelutto. "Second generation" skeleton systems. In Joubert et al. [40], pages 803–811.

[23] M. Danelutto, R. D. Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. In D. Skillicorn and D. Talia, editors, *Programming Languages for Parallel Processing*. IEEE Computer Society Press, 1994.

[24] M. Danelutto and M. Stigliani. SKElib: Parallel programming with skeletons in C. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par 2000 - Parallel Processing: 6th International Euro-Par Conference, Munich, Germany, August/September 2000. Proceedings*, volume 1900 of *Lecture Notes for Computer Science*, pages 1175–1184. Springer, 2000.

[25] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *Proceedings of PARLE'93 – Parallel Architectures and Languages Europe*, volume 694 of *Lecture Notes for Computer Science*, pages 146–160. Springer, 1993.

[26] J. Darlington, Y. Guo, H. W. To, Q. Wu, J. Yang, and M. Kohler. Fortran-S: A Uniform Functional Interface to Parallel Imperative Languages. In *Proceedings of the Third Parallel Computing Workshop*, 1994.

[27] G. Denker, J. Meseguer, and C. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In *Proc. DARPA Information Survivability Conference and Exposition DICEX 2000, Vol. 1, Hilton Head, South Carolina, January 2000*, pages 251–265. IEEE, 2000.

[28] F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of 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 on Mobile Agents, ASA/MA 2000, Zurich, Switzerland, September 13–15, 2000, Proceedings*, volume 1882 of *Lecture Notes in Computer Science*, pages 73–85. Springer, 2000.

[29] F. Durán, A. Riesco, and A. Verdejo. A distributed implementation of Mobile Maude. In G. Denker and C. Talcott, editors, *Proceedings Sixth International Workshop on Rewriting Logic and its Applications, WRLA 2006*, Electronic Notes in Theoretical Computer Science, pages 35–55. Elsevier, 2006.

[30] F. Durán and A. Verdejo. A conference reviewing system in Mobile Maude. In Gadducci and Montanari [35], pages 79–95.

[31] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In Gadducci and Montanari [35], pages 115–141.

[32] A. Farzan and J. Meseguer. State space reduction of rewrite theories using invisible transitions. In M. Johnson and V. Vene, editors, *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 5-8, 2006, Proceedings*, volume 4019 of *Lecture Notes for Computer Science*, pages 142–157. Springer, 2006.

[33] J. F. Ferreira, J. L. Sobral, and A. J. Proença. JaSkel: A Java skeleton-based framework for structured cluster and grid computing. In *CCGRID'06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 301–304. IEEE Computer Society, 2006.

[34] C. Fournet and G. Gonthier. The join calculus: A language for distributed mobile programming. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics: Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer, 2002.

[35] F. Gadducci and U. Montanari, editors. *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[36] R. Gray. *Agent Tcl: A flexible and secure mobile-agent system*. PhD thesis, Department of Computer Science, Dartmouth College, June 1997.

[37] K. Hammond and A. J. Rebón Portillo. HaskSkel: Algorithmic skeletons in haskell. In P. Koopman and C. Clack, editors, *Implementation of Functional Languages, 11th International Workshop, IFL'99, Lochem, The Netherlands, September 7-10, 1999. Selected*

Papers, volume 1868 of *Lecture Notes in Computer Science*, pages 181–198. Springer, 1999.

[38] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[39] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms*. Computer Science Press. W. H. Freeman and Company, 1997.

[40] G. Joubert, W. Nagel, F. Peters, O. Plata, P. Tirado, and E. Zapata, editors. *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, volume 33 of *NIC Series*. John von Neumann Institute for Computing, 2005.

[41] D. Kotz and R. Gray. Mobile agents and the future of the internet. *ACM Operating Systems Review*, 33(3):7–13, 1999.

[42] D. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42:88–89, 1999.

[43] R. Loogen, Y. Ortega-Mallén, and R. Peña. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(1):431–475, 2005.

[44] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism abstractions in Eden. In Rabhi and Gorlatch [58], chapter 4, pages 95–129.

[45] B. B. Mandelbrot. *Fractals and Chaos: The Mandelbrot Set and Beyond*. Springer, 2004.

[46] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Specifications*. Springer, 1992.

[47] N. Martí-Oliet, I. Pita, J. L. Fiadeiro, J. Meseguer, and T. Maibaum. A verification logic for rewriting logic. *Journal of Logic and Computation*, 15(3):317–352, 2005.

[48] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[49] 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. The MIT Press, 1993.

[50] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[51] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[52] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[53] J. Misra. A logic for concurrent programming. Technical report, University of Texas at Austin, 1994.

[54] P. Ölveczky, J. Meseguer, and C. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design*, 29:253–293, 2006.

[55] R. Peña and C. Segura. Reasoning about skeletons in Eden. In Joubert et al. [40], pages 851–858.

[56] S. Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.

[57] I. Pita. *Técnicas de especificación formal de sistemas orientados a objetos basadas en lógica de reescritura*. PhD thesis, Facultad de Matemáticas, Universidad Complutense de Madrid, 2003.

[58] F. A. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2002.

[59] A. Riesco and A. Verdejo. Distributed applications implemented in Maude with parameterized skeletons. In M. Bonsangue and E. Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems: 9th IFIP WG 6.1 International Conference, FMOODS 2007, Paphos, Cyprus, June 5-8, 2007, Proceedings*, volume 4468 of *Lecture Notes for Computer Science*, pages 91–106. Springer, 2007.

[60] A. Riesco and A. Verdejo. Parameterized skeletons in Maude. Technical Report TR 1/07, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2007.

[61] G.-C. Roman and P. J. McCann. An introduction to mobile UNITY. In J. Rolim, editor, *Parallel and Distributed Processing, 10 IPPS/SPDP'98 Workshops Held in Conjunction with the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing Orlando, Florida, USA, March 30 – April 3, 1998 Proceedings*, volume 1388 of *Lecture Notes in Computer Science*, pages 871–880. Springer, 1998.

[62] G.-C. Roman and J. Payton. Mobile unity schemas for agent coordination. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003. Advances in Theory and Practice: 10th International Workshop, ASM 2003, Taormina, Italy, March 3-7, 2003. Proceedings*, volume 2589 of *Lecture Notes in Computer Science*, pages 126–150. Springer, 2003.

[63] J. Tardo and L. Valenta. Mobile agent security and Telescript. In *Proceedings of the 41st IEEE International Computer Conference*, pages 58–63. IEEE Computer Society, 1996.

[64] A. R. Tripathi, N. M. Karnik, M. K. Vora, T. Ahmed, and R. D. Singh. Mobile agents programming in Ajanta. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS '99)*, pages 190–197. IEEE Computer Society, 1999.

[65] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, 2002.

[66] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.

[67] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In H. Bal, B. Belkhouche, and L. Cardelli, editors, *Internet Programming Languages: ICCL'98 Workshop, Chicago, IL, USA, May 1998. Proceedings*, volume 1686 of *Lecture Notes for Computer Science*, pages 47–77. Springer, 1999.