Distributed Applications Implemented in Maude with Parameterized Skeletons*

Adrián Riesco and Alberto Verdejo

Facultad de Informática
Universidad Complutense de Madrid, Spain
ariesco@fdi.ucm.es, alberto@sip.ucm.es

Abstract. Algorithmic skeletons are a well-known approach for implementing parallel and distributed applications. Declarative versions typically use higher-order functions in functional languages. We show here a different approach based on object-oriented parameterized modules in Maude, that receive the operations needed to solve a concrete problem as a parameter. Architectures are conceived separately from the skeletons that are executed on top of them. The object-oriented methodology followed facilitates nesting of skeletons and the combination of architectures. Maude analysis tools allow to check at different abstraction levels properties of the applications built by instantiating a skeleton.

Keywords: Algorithmic skeletons, parameterization, distributed applications, Maude.

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. 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 many stand-alone computer systems.

Parallel algorithms divide the problem into subproblems, pass them to many processors and collect the results back together at the end. An *algorithmic skeleton* [3,14] is an abstraction shared by a range of applications which can be executed in a parallel way. The aim is to obtain generic schemes that allow parallel programming where the user does not have to handle low level features like communication and synchronization.

A skeleton can be executed on different architectures/topologies. However, there is often a most suitable architecture for each skeleton that takes advantage of the task distribution specified by it. In our implementation we have opted

* Research supported by MEC Spanish project *DESAFIOS* (TIN2006-15660-C02-01) and Comunidad de Madrid program *PROMESAS* (S0505/TIC/0407).

to separate the definition of the architectures from the skeletons, allowing us to combine them in several ways.

Rewriting logic [10] 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. Maude [2] 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. It has already been used to specify and analyze distributed applications and protocols [4,12]. 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.

We show here 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 solving a problem 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.

Typically, declarative implementations of skeletons are based on functional languages (like Eden [9], GpH [16], or PMLS [11]) that naturally represent skeletons as higher-order functions. These languages also allow to prove skeletons correctness [13]. Although rewriting logic is not a higher-order framework, the parameterization features provided by Maude allow to achieve similar results. From a "more practical" world, skeletons have recently been proposed for Java in the JaSkel language [8]. It uses object-oriented features like inheritance and abstract classes to present the skeletons in a hierarchical way that allows the user to instantiate them with concrete applications. We follow a very similar approach which provides an important advantage. The skeletons implemented, analyzed, and proved correct in Maude can then be translated to a language such as JaSkel with little effort.

Below we describe Maude's main features, specially the object-oriented notation used in the rest of the paper. How to implement different architectures is shown in Section 2. Parameterized skeletons are described and instantiated in Section 3. Section 4 shows how to check properties of the architectures and the skeletons. Finally, we present some conclusions and future work. For more detailed explanations of all the topics shown in this paper, see [15].

1.1 Maude

In Maude [2] the state of a system is formally specified as an algebraic data type by means of an equational specification. In this kind of specification we can define new types (by means of keyword `sort(s)`); subtype relations between types (`subsort`); operators (`op`) for building values of these types; and equations (`eq`) that identify terms built with these operators.

The *dynamic* behavior of such a distributed system is then specified by rewrite rules of the form $t \longrightarrow t'$ *if* C, that describe the local, concurrent transitions of the system. That is, when a part of a system matches the pattern t and satisfies the condition C, it can be transformed into the corresponding instance of the pattern t'.

Regarding object-oriented specifications, *classes* are declared with the syntax `class` C `|` a_1`:`S_1`,...,` 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* 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`).

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. The rewrite rules specify the behavior associated with the messages. The general form of such rules is

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

where $k, p, q \geq 0$, and the M_s are message expressions. The result of applying a rewrite rule is that the messages $M_1, \dots, M_n$ disappear; the state and possibly the class of the objects $O_{i_1}, \dots, O_{i_k}$ may change; all the other objects O_j vanish; new objects $Q_1, \dots, Q_p$ are created; and new messages $M'_1, \dots, M'_q$ are sent.

By convention, the only object attributes made explicit in a rule are those relevant for that rule. We use here the Full Maude object-oriented notation [2]. However, the actual implementation of the skeletons is in Core Maude because Full Maude does not support external objects. The complete Maude code can be found in `http://maude.sip.ucm.es/skeletons`.

Maude modules can be *parameterized* with one or more parameters, each of which is expressed by means of one *theory* that defines the interface of the module, that is, the structure and properties required of an actual parameter. *Views* are used to specify how a particular module is claimed to satisfy a theory.

Maude is *reflective*, that is, it can be represented into itself in such a way that a module in Maude may be data for another Maude module. This functionality has been efficiently implemented in the predefined module `META-LEVEL`, where concepts such as reduction or rewriting are reified by means of functions.

2 Different architectures

In this section 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 skeletons we will execute on top of it. Thus, the same skeleton can be run over different architectures.

Each located configuration is executed in a Maude process, and they are connected through sockets. Maude supports (bidirectional) sockets as external objects, and offers messages for interacting with them. However, these sockets do not preserve message boundaries, so we have extended their functionality by implementing "buffered sockets" with this feature [15]. In the following sections we present how we use these sockets to define different architectures.

A first approach to really distributed architectures in Maude was shown in [5]. However, those architectures were mixed with the applications. Here, we improve our approach by implementing them in an application-independent way.

2.1 Common infrastructure

In this section we show the elements that are common to all the architectures we define below. They basically correspond to the way messages are redirected to reach their addresses. The different parts among the architectures correspond to the way the locations 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`), and have the form `l(IP, N)` with the string `IP` the IP address of the host machine and `N` a number. We assume global uniqueness of routers names in a distributed configuration. We can communicate the name of a location by using the message `new-socket`.

Objects situated in a located configuration `L` must have as identifier a value `o(L, N)` of sort `Oid`, where `N` is a number not used to name other objects in `L`. 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 `Contents`.

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

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 with the definition of all the operators used to construct messages that are going to be transmitted. But, since each application (each skeleton, in our case) needs different messages, we define a *parameterized* module, that receives as a parameter the syntax of the transferred data in a module `MOD` required by the `SYNTAX` theory.

```
 fth SYNTAX is
  inc META-LEVEL .    op MOD : -> Module .
 endfth
```

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

The Router class (that will be specialized in the different architectures) is defined as follows:

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

where the predefined parametric sort `Map{Loc,Oid}` represents partial functions from view `Loc` to view `Oid` (that identifies sockets in this case) and `Maybe{Oid}` is a sort that adds a default value `null` to `Oid`. A router may be in states `idle`, `waiting-connection`, or `active`, although other values can be added in concrete architectures. 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 general approach consisting in having a routing table in each router, that gives the socket through which a message must be sent if one wants to reach a particular location. The `neighbors` attribute maintains such a routing table as a map associating socket object identifiers to location identifiers. 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(L, N)` and the message is in a location `L'`, with $\texttt{L} \neq \texttt{L'}$, that is directly connected to `L` ($\texttt{LSPF[L]} \neq \texttt{undefined}$), then the message is sent through the socket `LSPF[L]` after converting it to a string with the function `msg2string`, that uses the `MOD` constant from the theory.

```
crl [redirect] :
    to o(L, N) : C
    < L' : Router | state : active, neighbors : LSPF >
 => < L' : Router | > Send(LSPF[L], L', msg2string(to o(L, N) : C))
 if L =/= L' /\ LSPF[L] =/= 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.

When a router sees a `Received` message that is not `new-socket`, it extracts the string (by means of the function `string2msg`) putting a new message in the configuration, and keeps listening with a new `Receive` message.

```
crl [Received] :
    Received(L, SOCKET, DATA) < L : Router | >
 => < L : Router | > string2msg(DATA) Receive(SOCKET, L)
 if not new-socket?(DATA) .
```

2.2 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.*

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's IP address. These classes define how the locations are connected by filling the neighbors and defNeighbor attributes.

The center plays the server role from the point of view of the sockets so it declares itself as a server socket, offering its services on port. Once it receives a CreatedSocket message, it becomes active and sends a message indicating that it is ready to accept clients through the server socket. In the rule below, in addition to sending messages AcceptClient (to continue accepting clients) and Receive (for receiving messages from the accepted client), the center sends to the node the message new-socket communicating its identifier.

```
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.

```
crl [Received] :
    Received(L, SOCKET, DATA)
    < L : StarCenter | state : active, neighbors : LSPF >
 => < L : StarCenter | neighbors : insert(L', SOCKET, LSPF) >
    Receive(SOCKET, L)
 if new-socket(L') := string2msg(DATA) .
```

When a StarNode is created, it first tries to establish a connection with the center by sending a message that uses the IP address and the port of the center, reaching the state waiting-connection. The response is handled by the following rule connected, where the node also sends the new-socket message right after the socket is created. Nodes start listening with the Receive message.

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

Finally, nodes make the connection with the center the default one.

2.3 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, which is impossible for the Maude instance that is first executed. 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 asks to be their client. Once they have accepted a client, they try to be clients themselves of the next node in the ring.
- `RingLast` defines the behavior of the last node, that asks the next one to be its server, and then waits to be a server itself.

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. 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. We also declare new router states `connecting2next` and `waiting4previous`.

When a node is accepted as client, it keeps the socket in the attribute `defNeighbor`, in order to use it to redirect all the messages, and reaches the `active` state. In this architecture the `neighbors` attribute is not used; the ring nodes are just connected by `defNeighbor`, thus obtaining a unidirectional ring.

2.4 Centralized ring architecture

We show here a special ring architecture, where in addition to the ring we have a central server connected to each location, so we have a mixture of the two previous architectures. We have tried to reuse them as much as possible. We use the class `StarCenter` from the star architecture for the ring center; and we reuse the classes `RingNode` and `RingLast` described above for the nodes in the ring.

We define a new class `CRingRouter` in charge of connecting to a central server. We will combine the behavior of this new class with the classes `RingNode` and `RingLast` from the ring architecture. This new class has new attributes `centerIP` and `centerPort`, with the IP address and port of the center; new states `connecting2center` and `waiting4center`; and rules for connecting to the central node. When it is in `connecting2center` state, it tries to connect to the center and reaches `waiting4center`. Once the connection has been created, it sends a `new-socket` message and reaches the `active` state.

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 `CRNode`, which is a subclass of both `CRingRouter` and `RingNode` (and a new class `CRLast`, which is a subclass of `CRingRouter` and `RingLast`). These new classes behave

[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.

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 the `active` state, so some of the rules could not be applied. We solve it by renaming the `active` state in the ring nodes to `connecting2center`, so the rules in `CRingRouter` can be applied *after* the ring connections has been established.

```
omod CENTRALIZED-RING-NODE{M :: SYNTAX} is
 pr CENTRALIZED-RING{M} .
 pr RING-NODE{M} * (op active to connecting2center) .
 class CRNode | .   subclass CRNode < CRingRouter RingNode .
endom
```

In the following section we will show how these architectures can be used to execute skeletons on top of them. In [15], we also show how a concrete distributed application can be implemented directly in Maude (without skeletons).

3 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 accomplish this goal by means of parameterized modules whose parameter includes the characterization of the problem. We apply our methodology to three kinds of skeletons [14]:

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 show the *farm skeleton* with and without *fixed data*.

Systolic skeletons: The systolic skeletons are used in algorithms in which parallel computation and global synchronization steps alternate. We show the *ring* version of the systolic skeleton.

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. We have implemented three task-parallel skeletons: *divide and conquer* (shown here), *branch and bound*, and *pipeline*.

Indications of the most appropriate architecture for each skeleton will be given in the following sections.

3.1 Farm skeleton

We show here how to implement a skeleton with *replicated workers* and *fixed data*. There is a *master* that 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 then 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 have a direct communication between the master and the workers the star architecture is the most appropriate one, with the master located in the center and the workers in the nodes.

Each concrete application must define a module fulfilling the RW_FD-PROBLEM theory, that requires the sorts FixData (containing the data common to all the subproblems), Problem (refering to the initial problem), SubProblem (representing the smaller problems solved by the workers), Result (keeping the final result), and SubResult (corresponding to the results obtained by the workers).

The operations required by the theory are: new-work, that extracts a new subproblem from the current problem; reduce, that updates the current problem making it smaller; do-work, that given a subproblem and the fixed data solves the former; combine, that merges the current (partial) result with a new subresult, given the subproblem that was solved (this operation must be commutative[3], in the sense that the final result cannot depend on the order in which the combinations are performed, because the subresults may arrive unordered); and finished?, that checks if the problem has already been solved.

We declare the messages fixData and new-work for sending the fixed data and new tasks to the workers, and finished for communicating the subresults to the master.

The skeleton receives as another parameter the SYNTAX theory, that will be used by the architecture. First, classes for the master and the workers 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 : Maybe{FixData}, master : Oid .
```

The master stores the fixed data (fixData, that cannot be null), the current unsolved problem, the partial result, the list of idle workers, and the number of initial tasks assigned to each worker (numWorks).[4]

```
  class RW_FD-Master | fixData : P$FixData, problem : P$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:

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

where sendTasks and update are equationally defined operations that generate the messages with the initial tasks and reduce the problem accordingly. While the list of unfinished tasks of a worker is not empty, it must do the following one and send the subresult to the master.

[3] This requirement is represented in the theory by means of an equation [15].

[4] P$Sort means that Sort comes from the parameter P.

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

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

Ray tracing instantiation We can implement this well-known case study by starting from part of the sequential implementation included in module ROWTRACER [15] solving the problem for one row by means of function traceRow, and extending it in such a way that it fulfills the requirements from theory RW_FD-PROBLEM. The sort Pair is declared to define the initial problem (the highest and the lowest y), while World defines the fixed data (the width of the screen, the camera, and the list of figures). A partial function from floats (identifying rows) to colored rows is used to represent the final result. To instantiate the module we create a view [2] and define the mapping between sorts and operators with different names 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,ColorRow} .    sort SubResult to ColorRow .
  sort FixData to World .
  op do-work to trace-row .               op new-work to sub-problem .
  op combine(R:Result, SP:SubProblem, SR:SubResult) to
       term insert(SP:Float, SR:ColorRow, R:Map{Float,ColorRow}) .
endv
```

Finally, we instantiate the module RW_FD-SKELETON and use the star architecture. RT-Syntax is a view that encapsulates the syntax of transmitted messages.

```
 mod RAYTRACING-SKELETON is
  pr RW_FD-SKELETON{RayTracer, RT-Syntax} .
  pr STAR-ARCH-STAR-CENTER{RT-Syntax} .
  pr STAR-ARCH-STAR-NODE{RT-Syntax} .
 endm
```

Euler instantiation In some problems the fixed data is not needed; we have implemented a slightly modified skeleton to deal with this situation.

The Euler number $\varphi(x)$ is the number of natural numbers smaller than x that are relatively prime to x. We are interested in computing $\sum_{i=1}^{n} \varphi(i)$. We distribute the problem by considering 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. The operations are very simple too: a new work of the problem N is just N; we reduce the problem by subtracting 1; the work that must be done is the function euler from module EULER; combining two results is just adding them; and we have finished when the number reaches 0.

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

Calculating $\varphi(x)$ may be quite faster than communicating it, 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 Euler 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 the sum of 20 Euler numbers in each step with a new function `euler20`.

```
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
```

3.2 Systolic skeleton

In this skeleton, a master divides 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 (the first shared data is produced by the worker itself), they do their work, combine the partial result, and give the new shared data to the next worker. When a worker finishes all its tasks, it sends its subresult to the master, that will combine them in order.

We define a theory that requires the following sorts: `Problem` and `Result` represent the initial and final data; `SharedData` corresponds to the data that is passed by all the workers; and `Pair` is a wrapper of `Result` and `SharedData`.

The theory defines the following operations: `divide` splits the initial problem into a list of problems; `initialSharedData` extracts from the initial data the shared one; `do-work` computes, given the initial and the shared data, a partial result and the shared data to be communicated to the next worker; `combine`, used by the workers, merges the current partial result with a new one; `combine-all`, used by the master, merges all the partial results from the workers; and `finished?` checks if the worker has finished all its tasks.

We need the following messages: `initial-work` communicates the initial data to the workers; `shared-data` delivers the shared data to the next worker; and `finished` sends a result to the master, once the worker has finished.

This skeleton uses the classes `SWorker` and `SMaster` with attributes that allow the workers to keep the partial results and send and receive the shared data in order, and the master to collect and combine the results in order.

The first thing that must be done by the master is to divide the initial problem into a list of problems, that are delivered to all the workers, which first store each problem and extract the initial shared data. Once the worker has shared data it can do a new work and send the updated shared data to the next worker, forgetting its own. When the next shared data arrives, it is checked if the work is not finished, in which case the worker keeps the shared data. Finally, when the master has received all the results, it merges them.

In this case, the centralized ring architecture is the most appropriate one: the workers are located in the ring nodes and the master in the center. Examples can be found in [15].

3.3 Divide and Conquer

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 an implementation based on the replicated workers scheme, that allows a balanced distribution of the leaves of the problem tree. This implementation is suitable when 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.

We define 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. The function `divide` splits a problem into a list of subproblems, finishing when the problem `isTrivial`. Each trivial task is computed with `solve`. The function `combine` merges a list of subresults into a new subresult.

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

This skeleton defines the classes `DCMaster` and `DCWorker`, with attributes that allow the master to keep the tree of results and the workers to transmit the results with their corresponding identifier. First, the master must transform the initial problem into a list of subproblems, and create the initial result tree, that initially has all its nodes without data. Once the list of problems has been calculated, the master must transmit the initial tasks to the workers. Eventually, a task is finished and sent to the server, that inserts it in the result tree, merging the subresults if possible [15].

Since this skeleton is based on replicated workers, the most suitable architecture for the applications that instantiate it (examples are shown in [15]) is the star architecture. When the cost of the composition of the subresults is relevant, a hierarchical architecture with more levels could be more convenient.

4 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, we must be able to represent the whole system in one single term. We have provided an algebraic specification of sockets [15] and represented the processes (hosts in the distributed version) as objects of a class `Process` identified by the name of the location it represents, and with a single attribute `conf` keeping the configuration in that host separated from the others. The implementation of the distributed applications can be executed using these "simulated" sockets without changes. By doing this, we can check the properties of a system that is almost equal to the distributed one. However, we can trust some of the components of the whole system, and then abstract them, representing only the "suspicious" elements. These different *abstraction levels* speed up the checking process.

Model checking [1] is used to formally verify finite-state concurrent systems. 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 make it unfeasible to model check a system except for very simple cases. For this reason, several state space reduction techniques have been investigated. We use one based on the idea of *invisible transitions* [7], that generalizes a similar notion in partial order reduction techniques. By using this technique we can select a set of rewriting rules that fulfill some properties (such as termination, confluence, and coherence) and convert them into equations, thus reducing the number of states.

Maude's model checker [6] allows us to prove properties on Maude specifications. The properties to be checked are described by using Linear Temporal Logic (LTL) [1]. Then, the model checker can be used to check whether a given initial state, represented by a Maude term, fulfills a given property. To use the model checker we just need to make explicit two things: the intended sort of states (`Configuration` in our case), and the relevant *state predicates*, that is, the relevant atomic propositions. The latter are defined by means of equations that specify when a configuration C satisfies a property P, C `|=` P.

Sometimes all the power of model checking is not needed. Another Maude analysis tool is the `search` command, that allows exploration (following a breadth-first search strategy) of the reachable states in different ways. By using the `search` command we can check *invariants* [2]. 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 such that not I(C)` has no solution, then I holds.

4.1 Analyzing architectures

Architectures have been designed independently from the skeletons, and this allows to check properties over them. We show here some simple properties of the centralized ring architecture. Other properties on different architectures can be proved using the same methodology.

Using the model checker We want to check the behavior of the centralized ring when a node in the ring sends a message to another ring node. To study it we use an initial configuration with one of the nodes in the ring with an object and another with a message for it. Some of the nodes will be traversed by the message and others never will be traversed (at least the center). We define the property `has-msg`, that checks if a given location contains messages.

```
  op has-msg : Loc -> Prop .
  eq C < L : Process | conf : C' (to O : CNT) > |= has-msg(L) = true .
  eq C |= has-msg(L) = false [owise] .
```

We define the LTL formulas specifying the properties. The formula `F(L)` below expresses that `L` receives a message exactly once, and then redirects it, where `U` represents the *until*, `~` the negation, and `[]` the *henceforth* LTL operators.

```
eq F(L) = ~ has-msg(L) U (has-msg(L) /\ (has-msg(L) U [] ~ has-msg(L))) .
```

The formula `F'(L)` states that `L` never contains a message and `F''(L)` states that a message reaches `L` and stays there. They are defined in a similar way as above. We check this property in an example with five nodes in the ring (`l(ipi, 0)`, $i \in 1..5$), 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 message. Therefore we use the following command:

```
 red modelCheck(init, F(l(ip4, 0)) /\ F(l(ip5, 0)) /\ F(l(ip1, 0)) /\
                      F'(l(ip0, 0)) /\ F'(l(ip3, 0)) /\ F''(l(ip2, 0)) .
 result Bool: true
```

Using the `search` command We can check now that the connection between each node in the ring and the center is direct. In the configuration above, we place an object in the center and a message for it in the ring node `l(ip4, 0)`. We consider as an invariant (equationally defined) 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. The command to check the invariant is:

```
 search init2 =>* C such that not messages-invariant(l(ip4, 0), C) .
```

4.2 Analyzing skeletons

In order to check properties of the skeleton instantiations, we can consider the sequential version of the concrete application as the *specification* of the problem and the distributed, skeleton version as the *implementation*. We use the `search` command to analyze that in all possible executions of an instantiated skeleton (which introduces nondeterminism) the final result obtained coincides with the result of the deterministic sequential version. We define for each skeleton a `getResult` operation that, given a final configuration, returns the result kept in the master. We use it to compare the results from the sequential and the distributed implementation, although the comparison can be non trivial [15].

In the Euler example, `getResult` returns a natural number, that we have to compare with the result from the specification. The search command used is:

```
search initial(7) =>! C such that getResult(C) =/= sumEuler(7) .
```

In the mergesort application, used to instantiate the divide and conquer skeleton, the postcondition is simple enough to avoid the use of the sequential version to prove the correctness of the skeleton. We can define an `ordered` predicate that checks if a list is sorted and has the same components as another and use it in the `search` command:

```
search init(gen(1000)) =>! C s.t. not ordered(getResult(C), gen(1000)) .
```

5 Conclusions

We have presented the implementation of some static architectures using sockets, that Maude supports as external objects. We are currently developing more complex, fault-tolerant architectures, where nodes can join and leave.

We have implemented several skeletons as parameterized modules that receive as parameters 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.

From the Maude side, we show that truly distributed applications can be implemented and that the recently incorporated support for parameterization in Core Maude can be applied to more complex applications. From the point of view of skeleton development, we describe a methodology to specify, prototype, and check skeletons that can be later implemented in other languages such as Java (we plan to study in the near future which is the best way to achieve this).

We have tested the skeletons with some examples, using three 2 GHz PowerPC G5 and two 1.25 GHz PowerPC G4, obtaining a 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 wasted 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.

Mobile Maude [5], an extension of Maude allowing mobile computation, has also been used to implement skeletons, where the master and the workers were implemented as mobile objects that travelled through the architecture [15]. They had an attribute with the concrete code of the application. Although the same generality as in the work presented in this paper was obtained, the main drawback was lack of efficiency due to the reflection levels introduced.

Finally, we have started to study how our skeletons can be nested by using the object-oriented inheritance features provided by Maude. We are also investigating how to prove properties of the skeletons independently of the instantiations, by means of rule induction.

References

1. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.
2. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.2)*, December 2005. `http://maude.cs.uiuc.edu/maude2-manual`.
3. M. Cole. *Algorithmic Skeletons: Structure Management of Parallel Computations.* MIT Press, 1989.
4. 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.
5. F. Durán, A. Riesco, and A. Verdejo. A distributed implementation of Mobile Maude. In G. Denker and C. Talcott, editors, *Proc. Sixth Int. Workshop on Rewriting Logic and its Applications, WRLA 2006*, ENTCS, pages 35–55. Elsevier, 2006.
6. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Proc. Fourth Int. Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002*, ENTCS 71, pages 115–141. Elsevier, 2002.
7. 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*, LNCS 4019, pages 142–157. Springer, 2006.
8. 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.
9. R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism abstractions in Eden. In [14], chapter 4, pages 95–129.
10. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
11. G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested algorithmic skeletons from higher order functions. *Parallel Algorithms and Applications*, 16(2-3):181–206, 2001.
12. 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.
13. R. Peña and C. Segura. Reasoning about skeletons in Eden. In *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, NIC Series 33, pages 851–858, 2006.
14. F. A. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing.* Springer, 2002.
15. A. Riesco and A. Verdejo. Parameterized skeletons in Maude. TR 1/07, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2007. `http://maude.sip.ucm.es/skeletons/psm.pdf`
16. P. W. Trinder, H. W. Loidl, and R. F. Pointon. Parallel and distributed Haskells. *Journal of Functional Programming*, 12(4-5):469–510, 2002.