FROM ORC MODELS TO DISTRIBUTED GRID JAVA CODE[*]

Marco Aldinucci and Marco Danelutto
Dept. of Computer Science – University of Pisa – Italy
Marco.Aldinucci@di.unipi.it, Marco.Danelutto@di.unipi.it

Peter Kilpatrick
Dept. of Computer Science – Queen's University Belfast – UK
p.kilpatrick@qub.ac.uk

Patrizio Dazzi
ISTI/CNR – Pisa, Italy
patrizio.dazzi@isti.cnr.it

Abstract We present O2J, a Java library that allows implementation of Orc programs on distributed architectures including grids and clusters/networks of workstations. With minimal programming effort the grid programmer may implement Orc programs, as he/she is not required to write any low level code relating to distributed orchestration of the computation but only that required to implement Orc expressions. Using the prototype O2J implementation, grid application developers can reason about abstract grid orchestration code described in Orc. Once the required orchestration has been determined and its properties analysed, a grid application prototype can be simply, efficiently and quickly implemented by taking the Orc code, rewriting it into corresponding Java/O2J syntax and finally providing the functional code implementing the sites and processes involved. The proposed *modus operandi* brings a Model Driven Engineering approach to grid application development.

Keywords: Grid, orchestration, model-driven engineering, distributed computing.

[*]This research is carried out under the FP6 Network of Excellence CoreGRID and the FP6 GridCOMP project funded by the European Commission (Contract IST-2002-004265 and FP6-034442).

1. Introduction

In recent years Model-Driven Engineering (MDE) [11, 6] has emerged as a means of employing abstraction to allow exploration of properties of a system free of implementation detail. MDE advocates building models of systems to be implemented, reasoning informally about these models (for example, comparing alternative designs, identifying bottlenecks, etc.) and then developing code from these models, ideally automatically or semi-automatically.

In many ways MDE is similar in intent, if not in style, to Formal Methods (such as B [4], VDM [7]). A formal method requires a developer to provide a specification of a system written in a mathematically precise (i.e. formal) notation. The developer may then prove properties of the specification before committing to implementation. The implementation may proceed through a series of refinement steps, which can be proven consistent. The requirement for such (expensive) mathematical precision may have been a contributory factor to the limited uptake of Formal Methods.

In earlier work [1–3] we presented a semi-formal approach to the development of grid software. The approach draws upon ideas from both Formal Methods and MDE: we use a formal notation (Orc [8]) to describe different designs of grid software, together with an informal style of reasoning about the properties of the designs. Orc is suitable for the description of such software [12] as it has been designed explicitly as an orchestration language for distributed services. Orc has the benefit of being a formal notation in the sense that it is a small abstract notation (like traditional process algebras) amenable to reasoning, while at the same time (unlike traditional process algebras) it has a syntax which is appealing to the programmer and thus allows the description of highly readable designs and the development of informal arguments which reference code extracts.

Here we extend our earlier work, again in the spirit of MDE, by addressing the issue of generation of implementations from models. We describe O2J, a Java library that supports the semi-automated development of grid implementations from Orc models. The idea of such a system was introduced briefly in [3] where we described a preliminary version of O2J. Here we present a detailed description of the O2J library together with sample translations and a step-by-step guide indicating how the developer may use O2J to support the development of grid software.

2. Orc specification of distributed (grid) applications

In this section we present a simple example to motivate the approach and provide an orchestration expression to illustrate the use of O2J. First we briefly summarise the Orc notation.

2.1 The Orc notation

Orc is a “programming language and system for orchestrating distributed services ” [9]. Its focus is on the *orchestration* of what might be termed “services” (web site calls, grid site calls, method calls, etc.) which provide core functionality. The notion of service is captured by the *site* primitive and the orchestration of site calls is described using three operators plus recursion. (See [8] for a very readable introduction to Orc.)

Site A site call may return a *single* value or remain silent (not respond). A site represents the simplest form of Orc expression.

Sequential composition In $E_1 > x > E_2(x)$, expression E_1 is evaluated and may produce zero or more results. For each result generated, the result is labelled x and a new instance of expression E_2 is executed with parameter x. If evaluation of E_2 is independent of x the expression may be written $E_1 \gg E_2$.

Parallel Composition In $E_1 \,|\, E_2$ both E_1 and E_2 are evaluated in parallel. The output is the interleaved outputs of the constituent expressions.

Asymmetric Parallel Composition In E_1 where $x :\in E_2$ both E_1 and E_2 are evaluated in parallel. If E_1 names x in some of its site calls, its evaluation halts at the point of dependency. Evaluation of E_2 proceeds until it delivers a value for x at which point its evaluation is terminated. E_2 may now continue its evaluation with a binding for x.

Finally, the notation $(|i : 1 \leq i \leq 3 : W_i)$ is used as an abbreviation for $(W_1|W_2|W_3)$.

2.2 Example

Consider a straightforward but common distributed application: data items in an input stream are processed independently and the results written to a shared state (for simplicity assume that the update function is associative and commutative). In [3] we presented two alternative designs for this application and it was shown there how detailed analysis of the Orc designs allowed comparison of their expected performances. Here we present the simpler design (see Fig. 1) as a vehicle to illustrate the use of O2J.

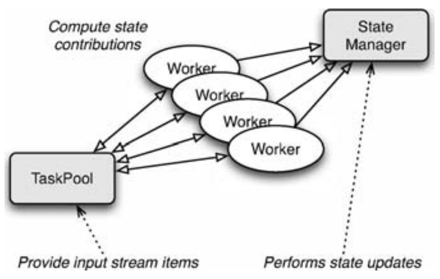


Figure 1. Master/worker implementation of state update (grey boxes represent sites, white ovals represent processes)

This design is based on the classical master/worker implementation where centralized entities provide the items of the input stream and collate the resulting computations in a single state. The *system* comprises a taskpool (modelling the input stream), *TP*, a state manager, *SM* and a set of workers, W_i. The workers repeatedly take tasks from the taskpool, process them and send the results to the state manager. The taskpool and state manager are represented by Orc sites; the workers are represented by processes (expressions). This specification corresponds to the diagram in Fig. 1 and can be formulated as follows:

$$\begin{array}{l} system(TP, SM) \triangleq\ workers(TP, SM) \\ workers(TP, SM) \triangleq \mid i : 1 \leq i \leq N : W_i(TP, SM) \\ W_i(TP, SM) \triangleq \\ \quad TP.get > tk > compute(tk) > r > SM.update(r) \gg W_i(TP, SM) \end{array}$$

3. Generating a distributed Java framework from Orc orchestration code

As discussed above, Orc fully supports distributed program orchestration design and refinement. Its conciseness and operational style allows one to construct compact implementation-oriented models of distributed system orchestrations unobscured by superfluous detail. The main focus here is to bridge the gap between the abstract (and therefore powerful) modelling and the actual (and therefore error prone and cumbersome) distributed programming practice by providing automatic translation from Orc code to Java implementation.

We present and discuss here a run time support (the O2J Java library, **O**rc **to J**ava) allowing Orc "programmers" to write simple and concise Java programs that can be run on a collection of processing elements implementing the distributed orchestration modelled by a given Orc expression/program. This is the first step in a process that will eventually be completed by a compiler taking as input Orc programs and generating in an automatic way the Java code implementing that particular Orc program. At the moment we concentrate on providing a suitable Java run time library allowing programmers to write Orc code in a "Java friendly" syntax. Table 1 presents Orc constructs and their corresponding library implementations. Thus, for example, the call `OrcSeqVar(f, x, g)` provides a Java implementation of the Orc sequential composition (with parameter passing) $f > x > g(x)$.

3.1 Library usage example

Consider again the example outlined in 2.2. Using O2J, the Orc program can be implemented by the Java code shown in Figure 2 (IW08Sample.java editor window) provided that suitable classes implementing the TP and SM sites as well as the WP process are provided by the user. Any site can be provided by subclassing `OrcSite` class and implementing an `OrcMessage` `body(OrcMessage call)` method. This method handles a single call message (the `call`) to produce a single answer to the call. For example, the TP site can be provided by programming an `OrcSite` subclass implementing the `body` method as follows:

```
public OrcMessage body(OrcMessage dummy) {
  // upon any request, send the next integer in the list, up to MAXTASK
  System.out.println("TaskPool "+getName()+" got request "+dummy);
  if(task < MAXTASK) {
    System.out.println("TaskPool "+getName()+" sending task "+task);
    return(new OrcMessage(new Integer(task++)));
  } else {
    System.out.println("TaskPool "+getName()+" returning a null");
    return null; // this implies blocking the answer ...
  }
}
```

The `Manager` object declared at the beginning of the `main` is the manager handling all the non-functional features related to execution of the Orc program. In particular, the `Manager` starts the run time processes needed to support the execution of the Orc program and manages the processing elements available according to the allocation strategies required by the programmer.

3.2 Deriving Java code from Orc: the formal steps

Formally, in order to implement an Orc program using O2J, the programmer must follow a precise procedure:

1 Write one class for each of the sites used in the program. Each class subclasses `OrcSite` and provides (in addition to constructors needed to store site specific parameters) the method handling a single call, i.e. an `OrcMessage body(OrcMessage call)`.

2 Write one class for each of the processes used in the program. As for sites, each class subclasses `OrcSite` and provides (in addition to constructors needed to store site specific parameters) an `OrcMessage body(OrcMessage call)` method hosting the code implementing the process body. The code may use `send`, `receive` and `call` methods provided by `OrcSite` to implement process actions.

3 Write a Java `main` which involves:

   (a) declaring a `Manager` (the Orc runtime) and possibly calling `Manager` methods to set up non-functional parameters for program execution;

   (b) declaring the sites used by calling the constructors defined in the classes extending `OrcSite` that represent the user defined sites;

(c) implementing the program expression, using the appropriate library objects implementing the Orc parallel, asymmetric parallel and sequential operators (`OrcPar` modelling the $|$ operator, `OrcSeq` modeling the $>$ operator and `OrcAsymm` modeling the **where x**:$\in$ operator); and

(d) starting the execution of the program by issuing a `startSite()` call on the main expression object.

It is worth pointing out that, at the moment, the Orc program modelled in the `main` corresponds to the *full inlining* of the actual Orc program in that all the expression calls are substituted by the corresponding expression bodies. In particular, the code presented at the beginning of section 3.1 is actually implemented as expressed in the Orc expression
$E \triangleq | \ i : 1 \leq i \leq N :$
$(TP.get > tk > compute(tk) > r > SM.update(r) \gg E)$

Also, tail recursion is to be programmed as infinite loops, and therefore the actual code implemented in the Java `main` corresponds to
$| \ i : 1 \leq i \leq N :$
`while(true){` $TP.get > tk > compute(tk) > r > SM.update(r)$ }

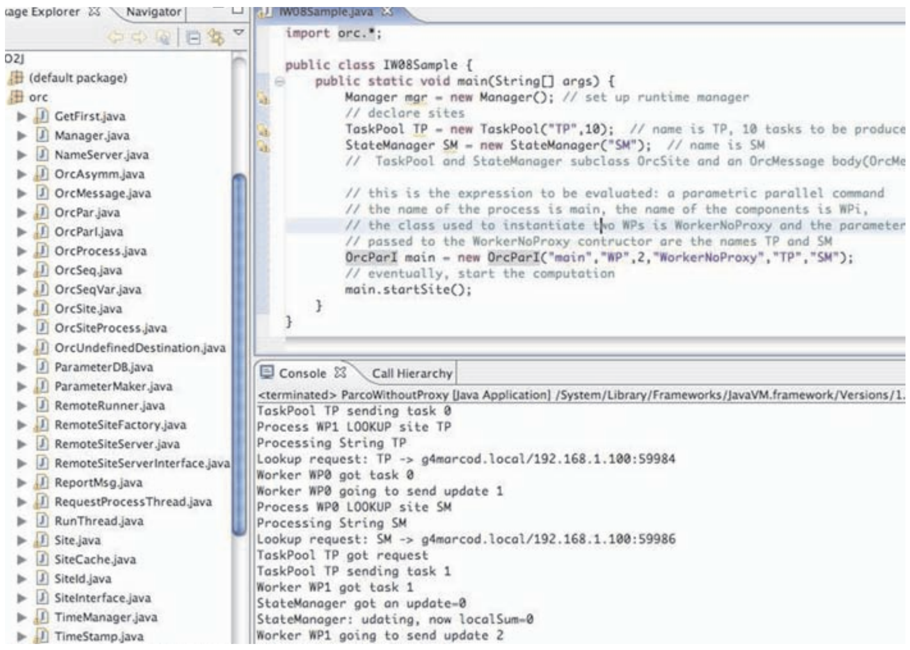
This will change in the near future, as we are currently designing a Java based compiler accepting as input Orc expressions and producing as output the skeleton of the Java code of the `main` program.

The whole process achieves a clear separation of concerns between programmers and O2J library code (that is, between programmers and system designers): programmers must concentrate on the Orc code and they must of course also provide the functional code – for sites, the code handling a single call; for processes, the process body code. The O2J code then handles all the details one usually has to take into account when running distributed code, that is process/thread decomposition, mapping and scheduling, communication implementation, etc.

To support our claim, let us consider the code the programmer must supply to implement the `WP` process in the example discussed above. This code consists in class subclassing `OrcSite` and implementing[1] the method:

```
public OrcMessage body(OrcMessage dummy) {
  while(true) {  // get a task (TP.get > tk )
    OrcMessage taskMessage = call(taskPoolName,OrcMessage.nullMessage());
    Object tk = taskMessage.getValue();
    // then process it (tk > compute(tk) > r)
    Object r = compute(tk);
    // eventually send new contrib. to the SM (SM.update(r) >> )
```

[1]overwriting, actually: `OrcSite` by default implements a site that just echoes call messages



```
import orc.*;

public class IW08Sample {
    public static void main(String[] args) {
        Manager mgr = new Manager(); // set up runtime manager
        // declare sites
        TaskPool TP = new TaskPool("TP",10);  // name is TP, 10 tasks to be produce
        StateManager SM = new StateManager("SM");  // name is SM
        //  TaskPool and StateManager subclass OrcSite and an OrcMessage body(OrcMe

        // this is the expression to be evaluated: a parametric parallel command
        // the name of the process is main, the name of the components is WPi,
        // the class used to instantiate two WPs is WorkerNoProxy and the parameter
        // passed to the WorkerNoProxy contructor are the names TP and SM
        OrcParI main = new OrcParI("main","WP",2,"WorkerNoProxy","TP","SM");
        // eventually, start the computation
        main.startSite();
    }
}
```

```
<terminated> ParcoWithoutProxy [Java Application] /System/Library/Frameworks/JavaVM.framework/Versions/1.
TaskPool TP sending task 0
Process WP1 LOOKUP site TP
Processing String TP
Lookup request: TP -> g4marcod.local/192.168.1.100:59984
Worker WP0 got task 0
Worker WP0 going to send update 1
Process WP0 LOOKUP site SM
Processing String SM
Lookup request: SM -> g4marcod.local/192.168.1.100:59986
TaskPool TP got request
TaskPool TP sending task 1
Worker WP1 got task 1
StateManager got an update=0
StateManager: udating, now localSum=0
Worker WP1 going to send update 2
```

Figure 2. O2J at work in Eclipse

```
    call(stateManagerName,new OrcMessage(r));
  } // then recur (recursive call => loop) (>> Wi(TP,SM) )
}
```

where the call message received at the beginning is just a dummy call initiating the computation and the recursive structure of the worker is implemented by an infinite loop. This is more or less what the programmer writes in the (inlined) Orc code. In particular, no code is needed to take care of explicit communications synchronization, handling of the mechanisms used to implement the Orc runtime (e.g. sockets, processes), etc., that is, those things that require extensive distributed/parallel architecture knowledge in order to be used efficiently.

3.3 Implementation

The Orc runtime implemented by O2J is completely OO and quite straightforward. Sites and processes are implemented by objects distributed on the available processing nodes by using features of ProActive [10] active objects. In turn, site and process objects communicate with other sites and processes using plain TCP/IP sockets and `OrcMessages`. We could have used any other mechanism suitable for instantiating and running an object on a remote processing element. In fact, we are considering using plain RMI (possibly wrapped through `ssh`) instead of the ProActive `migrateTo` mechanism that requires the

Orc term	O2J *implementation*
O2J RTS start	mgr=new Manager()
$f \gg g$	OrcSeq(f,g)
$f > x > g(x)$	OrcSeqVar(f,"x",g)
$f \mid g$	OrcPar(f,g)
$\mid i : 1 \leq i \leq k : W_i$	OrcParI("W",k,W)
$f(x)$ where $x :\in g$	OrcAsymm(f,g,"x")

Orc mechanism	O2J *implementation*
site call	call(sitename,message)
channel send	send(dest,message)
channel receive	m = receive(source)
process start	process.startSite()
process/site naming	setName(name)
formal param access	x=getParam("x")

Table 1. Orc expressions and corresponding O2J constructors (left), Orc mechanisms and corresponding O2J implementations (right; procedure calls are calls to methods of the `OrcSite` class).

full ProActive library to be installed on the employed nodes. For the same reason (clean OO structure of the library and plain usage of TCP/IP sockets to implement communication and synchronization), we have not considered using more elaborate distributed programming environments such as the ones developed in Pisa (muskel [5] and ASSIST) or even more complex grid middleware.

A centralized name server, started by the `Manager` constructor, takes care of all the port numbers and IP addresses needed to communicate with sites and processes. When a site/process object is created, it publishes a `ServerSocket` to accept calls/messages and communicates the IP address, port number and site/process name to the centralized name server. The first time a `call` or a `send` is performed at a given site/process, the runtime communicates with the centralized name server to get the IP address and port number of the destination site/process. These values are stored in a local cache and all future communications involving that site/process are handled by looking up the addresses in the local cache.

When a new site or a new process is declared in the user defined `main` it is allocated on one of the available machines according to an allocation strategy established with the manager. To this end, users can call a `Manager setAllocationStrategy(OrcRTSAllocStrategy s)` method to establish an `OrcManager.SPECULATIVE` or an `OrcManager.CONSERVATIVE` allocation strategy such as the ones discussed in [2]. In the former case, sites and processes are always placed on new processing elements, if available. In the latter, sites and processes are preferably placed on the processing element where the parent site or process expression has already been placed. The list of available processing elements is provided by an external XML file which is consulted when the `Manager` is started. We are considering also the possibility of having a distinct and autonomous "discovery thread" in the `Manager` that constantly keeps the list up to date by querying the networked processing elements and discovering their capabilities to run O2J processes[2]. In both cases (in the conservative and in the speculative strategy) the `OrcSite` invokes `Manager` methods (services) to determine which processing element is to be used to run the site (process).

O2J also provides a very basic logging facility. Every site or process can issue a `log(String eventName)` call to register an event in a trace that can be consulted offline after (or during) program execution. The trace is maintained by a centralized process and each event is registered in the trace with its name, the local timestamp (i.e. the time value obtained on the machine where the event was generated) and a global timestamp (i.e. the time value taken at the centralized server when the logging message was received). Despite the fact that local timestamps may be generated on processing elements with non-synchronized timers and the global timestamp counts also the communication overhead incurred in the logging process, the logging mechanism has proven effective at measuring coarse grain computations. With such computations the overhead involved in logging message transmission to the centralized log manager is negligible w.r.t. the times we wish to measure and therefore global timestamps may be used to determine the actual behaviour of distributed computations.

4. Experiments

The feasibility of the O2J approach was first tested with O2J 1.0, an implementation realized by Antonio Palladino, for his graduation thesis at the Dept. of Computer Science in Pisa, under the supervision of the authors. Once the feasibility was demonstrated we carried out a complete re-engineering of the library to obtain O2J 2.0. This required substantially new code, although the structure of sites, processes and manager did not change significantly. Here we present results achieved with O2J 2.0.

With O2J 2.0, the programmer may directly translate to O2J objects any part of a given Orc program. Table 1 summarizes the main correspondences between Orc constructs and O2J code. A *Manager* object is used to handle those aspects concerned with distributed implementation of the Orc code. After establishing the *Manager* object and the sites of the Orc program, the programmer must set up the O2J objects modelling the Orc expression to be evaluated in the program (as depicted in Figure 2). Then the program computation is started by issuing a `startSite()` method call on the main object representing the Orc program.

Using O2J 2.0, we implemented several Orc programs and successfully ran these programs with the expected modest programming effort: the programmer was required to write only the functional code (the routines to represent site calls and the bodies of the processes in the program) and all the rest was handled by the library. The messages output by running the sample code dis-

[2]A similar mechanism is successfully used in muskel [5].

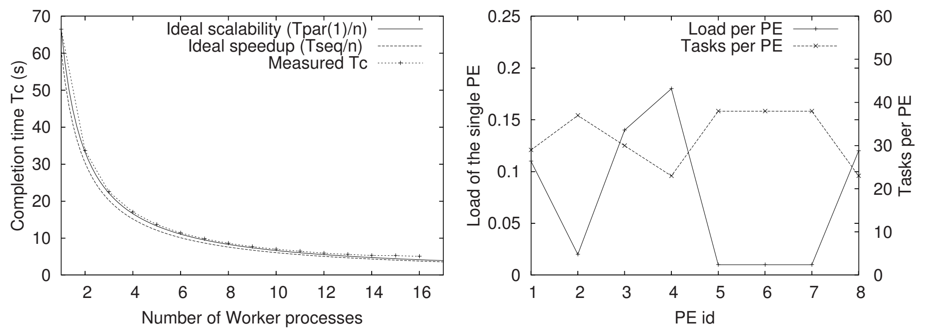


Figure 3. Sample experimental results: scalability (left) and load balancing (right)

cussed in Section 3 are shown in the lower window in Figure 2 (the Console window here shows only an initial portion of the messages printed by the sample code). It is worth pointing out that, in this case, both worker WP0 and WP1 were placed on the same machine. As a result, the first time a worker tried to `call` site SM, a lookup call was issued to the centralized manager which replied with the IP/port of site SM (this is the `Lookup request: SM -> g4marcod.local/192.168.1.100:54882` line in the dump). After some time, process WP1 attempted to send an update to SM but this time the address was resolved using the local cache (line `Process WP1 LOOKUP site SM`).

Figure 3 shows examples of results achieved with O2J 2.0. All the experiments have been run on a set of Linux workstations running kernel 2.4, Java 1.5 and interconnected via a Fast Ethernet network. The left half of the figure plots completion time for a run of the O2J implementation of the code described in Section 2 and whose process structure is shown in Figure 1. Using the program, we computed 256 tasks using a number of worker processes ranging from 1 to 16. The figure shows that good scalability is achieved up to 16 workers, despite the bottlenecks represented by the taskpool and by the state manager sites. The right part of the figure, plots average load (taken with the Linux command uptime) and number of tasks executed relative to a single processing element hosting a worker process. As the code shown in Section 2 clearly implements a self-scheduling policy (worker i calls the taskpool site to have a new task to compute as soon it completes the previous one), we expect that a sound implementation will achieve load balancing for either variable size tasks or heterogeneous processing elements. The right half of Figure 3 corresponds to a single run of the farm program of Section 2 with 8 workers. Nodes with a higher average load executed fewer tasks, as expected. It is worth noting that the variance in load (due to the concurrent run of a full compilation of *mpich*) is very small, but the auto scheduling reacted appropriately by running more tasks on the machines without the additional load (PEs 2, 5, 6 and 7, in this case).

We ran the program several times, using different loads to make the processing elements heterogeneous and each time we obtained results comparable to the ones in Figure 3.

The completion times used to compile Figure 3 (left) have been taken without considering the deployment time, i.e. from the moment the objects representing Orc sites and processes have been deployed onto their "home" remote nodes and initialized. As the code eventually run uses plain TCP/IP socket wrapped into Object streams, we did not measure any significant overhead with respect to hand written code modelling the same Orc program. In particular, the grain needed to achieve good scalability on the 16 nodes in the experiment of Figure 3 is of the same order of magnitude as that needed to make a `muskel` task farm scale (`muskel` uses only RMI and `ssh` to run applications).

Finally, we compared the amount of code needed to implement a simple site with O2J, with the amount of code needed to implement it in the library, representing a rough measure of the code needed to implement the same site from scratch. The Worker process of the program of Section 2 represents about 65 lines of Java code. The library classes used to implement the Site (of which Process is a sub-class) account for 10 times as many lines of code, without taking into account the `Manager` and deployment code.

Overall, the experimental data obtained has demonstrated reasonable efficiency in the O2J implementation of Orc programs, and the small amount of code needed to implement sites, processes and the Orc expressions to be evaluated illustrates the expressive power of using O2J.

5. Conclusions

We have discussed O2J, a Java library supporting distributed application development based on the Orc formal orchestration language. O2J allows Orc programmers to write Java code implementing Orc programs and to run these programs on distributed architectures hosting Java and ProActive enabled processing elements. Grid targeting comes as a consequence of the usage of ProActive as the "distributed/grid middleware". We presented a simple example that notably requires considerable programming effort if implemented directly using standard middleware mechanisms and we showed that the amount of Java/O2J code needed is small and mostly a direct translation from the high level specification of the problem in Orc. We also discussed some preliminary experimental results demonstrating the feasibility of the approach. Currently, to the best of our knowledge, there are no other "distributed" implementations of Orc (the Orc system available at [9] is not a distributed implementation). Our approach allows application programmers to reason about their distributed application structure in terms of an abstract Orc model and then obtain support from the O2J tools to produce the actual distributed implementation. Thus it brings grid application development under the umbrella of Model Driven Engineering techniques, and, as such, represents a significant step toward freeing the developer from the burden of detailed middleware knowledge.

Acknowledgments The authors wish to thank Antonio Palladino who implemented O2J 1.0 demonstrating the feasibility of the approach. They thank also the anonymous referees for their careful reading of the original submission and their constructive suggestions.

References

[1] M. Aldinucci, M. Danelutto, P. Kilpatrick, Management in distributed systems: a semi-formal approach. in: A.-M. Kermarrec, L. Bougé, and T. Priol, eds., *Proc. of 13th Intl. Euro-Par 2007 Parallel Processing*, vol. 4641 of LNCS, pp. 651–661, Rennes (F), Aug. 2007. Springer.

[2] M. Aldinucci, M. Danelutto, P. Kilpatrick, Adding metadata to Orc to support reasoning about grid programs, in: T. Priol and M. Vanneschi, eds., *Proc. of the CoreGRID Symposium 2007*, pp. 205–214, Rennes (F), Aug. 2007, Springer.

[3] M. Aldinucci, M. Danelutto, P. Kilpatrick, A framework for prototyping and reasoning about grid systems, in: C. Bischof, M. Bücker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, and F. J. Peters, eds., *Parallel Computing: Architectures, Algorithms and Applications (Proc. of PARCO 2007, Jülich (G))*, vol. 38 of NIC, pp. 355–362, Dec. 2007.

[4] B home page (2007). `http://www-lsr.imag.fr/B/`

[5] M. Danelutto, P. Dazzi, Joint structured/non structured parallelism exploitation through data flow, in: V. Alexandrov, D. van Albada, P. M. A. Sloot, J. Dongarra (Eds.), *Proc. of ICCS: Intl. Conference on Computational Science, Workshop on Practical Aspects of High-level Parallel Programming*, vol. 3992 of LNCS, Springer, Reading (UK), 2006.

[6] OMG Model Driven Architecture (2007). `http://www.omg.org/mda/`

[7] J. S. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, M. Verhoef, Validated Designs for Object-oriented Systems, Springer, 2005. ISBN: 1-85233-881-4.

[8] J. Misra, W.R. Cook, Computation Orchestration: A basis for a wide-area computing, *Software and Systems Modeling*, 6(1):83–110, Mar. 2006. DOI 10.1007/s10270-006-0012-1.

[9] Orc home page (2007). `http://www.cs.utexas.edu/users/wcook/projects/orc/`

[10] ProActive home page (2007). `http://www-sop.inria.fr/oasis/proactive/`

[11] D.C. Schmidt, Model-Driven Engineering, *Computer*, 39(2):25–31, Feb. 2006.

[12] A. Stewart, J. Gabarrò, M. Clint, T. Harmer, P. Kilpatrick, R. Perrott, Managing Grid Computations: An ORC-Based Approach, in: M. Guo et al, eds., *Parallel and Distributed Processing and Applications (ISPA)*, vol. 4330 of LNCS, Springer, pp. 278-291, 2006.