Management in distributed systems: a semi-formal approach ⋆

Marco Aldinucci[1], Marco Danelutto[1], and Peter Kilpatrick[2]

[1] Department of Computer Science, University of Pisa
{aldinuc,marcod}@di.unipi.it
[2] Department of Computer Science, Queen's University Belfast
p.kilpatrick@qub.ac.uk

Abstract. The reverse engineering of a skeleton based programming environment and redesign to distribute management activities of the system and thereby remove a potential single point of failure is considered. The Orc notation is used to facilitate abstraction of the design and analysis of its properties. It is argued that Orc is particularly suited to this role as this type of management is essentially an orchestration activity. The Orc specification of the original version of the system is modified via a series of semi-formally justified derivation steps to obtain a specification of the decentralized management version which is then used as a basis for its implementation. Analysis of the two specifications allows qualitative prediction of the expected performance of the derived version with respect to the original, and this prediction is borne out in practice.

Key words: Orchestration, algorithmic skeletons, autonomic computing.

1 Introduction

The muskel system, introduced by Danelutto in [1] and further elaborated in [2], reflects two modern trends in distributed system programming: the use of program skeletons and the provision of means for marshalling resources in the presence of the dynamicity that typifies many current distributed computing environments, e.g. grids. muskel allows the user to describe an application in terms of generic skeleton compositions. The description is then translated to a macro data flow graph [3] and the graph computed by a distributed data flow interpreter [2]. Central to the muskel system is the concept of a *manager* that is responsible for recruiting the computing resources used to implement the distributed data flow interpreter, distributing the fireable data flow instructions (tasks) and monitoring the activity of the computations. The muskel manager is to a certain extent an autonomic manager [4, 5]: it adapts the run time behaviour of a muskel program to tolerate faults and maintain a user defined performance contract, much in the sense of what is advocated in [6, 7].

⋆ This research is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

While the performance results demonstrated the utility of muskel, it was noted in [2] that the centralized data flow instruction repository (taskpool) represented a bottleneck and the manager a potential single point of failure. The work reported on here addresses the latter of these issues. The planned reengineering of the muskel manager was seen as an opportunity to extend earlier related experiments [8] with the language Orc [9] to investigate if it could usefully be employed in the development of such management software. The intent was not to embark upon a full-blown formal development of a modified muskel manager (as was done earlier for the related *Lithium* system [10], or as is normally done when employing other popular formalisms, such as the π-calculus [11]), with attendant formulation and proof of its properties, but rather to discover what return might be obtained from the use of such a formal notation for *modest* effort. In this sense, the aim was in keeping with the lightweight approach to formal methods as advocated by, inter alia, Agerholm and Larsen [12].

Orc was viewed as being apt for two reasons. First, it is an orchestration language, and the job of the muskel manager is one of orchestrating computational resources and tasks; and, second, while there are many process calculi which may be used to describe and reason about distributed systems, the syntax of Orc was felt to be more appealing to the distributed system developer whose primary interest lies not in describing and proving formal properties of systems.

The approach taken was to reverse engineer the original muskel manager implementation to obtain an Orc description; attempt to derive, in semi-formal fashion, a specification of a modified manager based on decentralized management; and, use this derived specification as a basis for modifying the original code to obtain the decentralized management version of muskel. By "semi-formal" we mean that the derivation is presented as a chain of steps in which the terms are described in the (formal) notation of the specification and the steps are justified by rigorous argument of the mathematical textbook variety, but calling also upon domain knowledge and experience when appropriate.

The work described in this paper is the first part of a more complex activity aimed at both removing the single point of failure represented by the muskel manager *and* implementing a distributed data flow instruction repository, removing the current related bottleneck. While the second step is still ongoing, the first step provides a suitable vehicle to illustrate the proposed methodology.

Overall, this work is part of a set of articles that are currently being published and that build on the semi-formal framework discussed here. In particular, in [16] the semi-formal approach based on Orc is extended to encompass meta data modeling non-functional aspects related to parallel/distributed program execution, while in [17] the complete approach exploiting Orc to support distributed program development is summarized.

2 muskel: an overview

muskel is a skeleton based parallel programming environment written in Java. The distinguishing feature of muskel with respect to other skeleton environments

[13, 14] is the presence of an application manager. The muskel user instantiates a manager by providing the skeleton program to be computed, the input and the output streams containing the (independent) tasks to be computed and the results, respectively, and a performance contract modeling user performance expectations (currently, the only contract supported is the `ParDegree` one, requesting the manager to maintain a constant parallelism degree during application computation). The user then requests invocation of the `eval()` method of the manager and the application manager takes care of all the details relating to the parallel computation of the skeleton program.

When the user requires the computation of a skeleton program, the muskel system behaves as follows. The skeleton program is compiled to a macro data flow graph, i.e. a data flow graph of instructions modeled by significant portions of Java code corresponding to user `Sequential` skeletons [3]. A number of remote resources (sufficient to ensure the user performance contract) running an instance of the muskel run time are recruited from the network. The muskel run time on these remote resources provides an RMI object that can be used to compute arbitrary macro data flow instructions, such as those derived from the skeleton program. For each task appearing on the input stream, a copy of the macro data flow graph is instantiated in a centralized `TaskPool`, with a fresh graph id [2]. A `ControlThread` is started for each of the muskel remote resources (`RemoteWorker`s) just discovered. The `ControlThread` repeatedly looks for a fireable instruction in the task pool (the data-flow implementation model ensures that all fireable instructions are independent and can be computed in parallel) and sends it to its associated `RemoteWorker`. That `RemoteWorker` computes the instruction and returns the results. The results are either stored in the appropriate data flow instruction(s) in the task pool or delivered to the output stream, depending on whether they are intermediate results or final ones. In the event of `RemoteWorker` failure, i.e. if either the remote node or the network connecting it to the local machine fails, the `ControlThread` informs the manager and it, in turn, requests the name of another machine running the muskel run time support from a centralized *discovery service* and forks a new `ControlThread` to manage it, while the `ControlThread` managing the failed remote node terminates after reinserting in the `TaskPool` the macro data flow instruction whose computation failed [1]. Note that the failures handled by the muskel manager are *fail-stop* failures, i.e. it is assumed that an unreachable remote worker will not simply restart working again, or, if it restarts, it does so in its initial state. muskel has already been demonstrated to be effective on both clusters and more widely distributed workstation networks and grids [1, 2].

3 The Orc notation

The orchestration language Orc has been introduced by Misra and Cook [9]. Orc is targeted at the description of systems where the challenge lies in organising a set of computations, rather than in the computations themselves. Orc has, as primitive, the notion of a site call, which is intended to represent basic computations. A site, which represents the simplest form of Orc expression, either returns a *single* value or remains silent. Three operators (plus recursion) are provided for the orchestration of site calls:

1. operator $>$ (sequential composition)
   $E_1 > x > E_2(x)$ evaluates E_1, receives a result x, calls E_2 with parameter x. If E_1 produces two results, say x and y, then E_2 is evaluated twice, once with argument x and once with argument y. The abbreviation $E_1 \gg E_2$ is used for $E_1 > x > E_2$ when evaluation of E_2 is independent of x.
2. operator $|$ (parallel composition)
   $(E_1 \mid E_2)$ evaluates E_1 and E_2 in parallel. Both evaluations may produce replies. Evaluation of the expression returns the merged output streams of E_1 and E_2.
3. where (asymmetric parallel composition)
   E_1 where $x :\in E_2$ begins evaluation of both E_1 and $x :\in E_2$ in parallel. Expression E_1 may name x in some of its site calls. Evaluation of E_1 may proceed until a dependency on x is encountered; evaluation is then delayed. The first value delivered by E_2 is returned in x; evaluation of E_1 can proceed and the thread E_2 is halted.

Orc has a number of special sites:

– 0 never responds (0 can be used to terminate execution of threads);
– if b returns a signal if b is true and remains silent otherwise;
– *RTimer(t)*, always responds after t time units (can be used for time-outs);
– *let* always returns (publishes) its argument.

The notation $(|i : 1 \leq i \leq 3 : w_i)$ is used as an abbreviation for $(w_1|w_2|w_3)$. Finally, while Orc does not have an explicit concept of "process", processes may be represented as expressions which, typically, name channels which are shared with other expressions. In Orc a channel is represented by a site [9]. *c.put(m)* adds m to the end of the (FIFO) channel and publishes a signal. If the channel is non-empty *c.get* publishes the value at the head and removes it; otherwise the caller of *c.get* suspends until a value is available.

4 muskel manager: an Orc description

The Orc description presented focuses on the management component of muskel, and in particular on the discovery and recruitment of new remote workers in the event of remote worker failure. The compilation of the skeleton program to a data flow graph is not considered.

The activities of the processes of the muskel system are now described, referring to the Orc specification presented in Fig. 1.

System The *system* comprises a program, *pgm*, to be executed (for simplicity a single program is considered: in reality a set of programs may be provided here); a set of *tasks* which are initially placed in a *taskpool*; a *discovery* mechanism which makes available processing engines (remote workers) recruited from a grid,

```
$system(pgm, tasks, contract, G, t) \triangleq$
        $taskpool.add(tasks) \mid discovery(G, pgm, t) \mid manager(pgm, contract, t)$

$discovery(G, pgm, t) \triangleq (|_{g \in G}$ ( if $remw \neq$ false $\gg rworkerpool.add(remw)$
                                         where $remw :\in$
                                            ( $g.can\_execute(pgm)$ $\mid Rtimer(t) \gg let(false)$ )
                                        )
                              ) $\gg discovery(G, pgm, t)$

$manager(pgm, contract, t) \triangleq$
   $|i : 1 \leq i \leq contract : (rworkerpool.get > remw > ctrlthread_i(pgm, remw, t))$
   $\mid monitor$

$ctrlthread_i(pgm, remw, t) \triangleq taskpool.get > tk >$
   (   if $valid \gg resultpool.add(r) \gg ctrlthread_i(pgm, remw, t)$
     | if $\neg valid \gg$ (  $taskpool.add(tk)$
                          $\mid alarm.put(i) \gg c_i.get > w > ctrlthread_i(pgm, w, t)$
                         )
   )
      where $(valid, r) :\in$
         (   $remw(pgm, tk) > r > let(true, r) \mid Rtimer(t) \gg let(false, 0)$ )

$monitor \triangleq alarm.get > i > rworkerpool.get > remw > c_i.put(remw)$
             $\gg monitor$
```

Fig. 1. Centralized management: Orc specification

G; and a *manager* which creates control threads and supplies them with remote workers. t is the time interval at which potential remote worker sites are polled; and, for simplicity, also the time allowed for a remote worker to perform its calculation before presumption of failure.

Discovery It is assumed that the call *g.can_execute(pgm)* to a remote worker site returns its name, g, if it is capable of (in terms of hardware and software resources) and willing to execute the program *pgm*, and remains silent otherwise. The call *rworkerpool.add(g)* adds the remote worker name *g* to the pool provided it is not already there. The *discovery* mechanism carries on indefinitely to cater for possible communication failure.

Manager The *manager* creates a number *(contract)* of control threads, supplies them with remote worker handles, monitors the control threads for failed remote workers and, where necessary, supplies a control thread with a new remote worker.

Control thread A control thread *(ctrlthread)* repeatedly takes a task from the *taskpool* and uses its remote worker to execute the program *pgm* on this task. A result is added to the *resultpool.* A time-out indicates remote worker failure which causes the control thread to execute a call on an *alarm* channel while returning the unprocessed task to the *taskpool.* The replacement remote worker is delivered to the control thread via a channel, c_i.

Monitor The *monitor* awaits a call on the *alarm* channel and, when received, recruits and supplies the appropriate control thread, i, with a new remote worker via the channel, c_i.

5 Decentralized management: derivation

In the muskel system described thus far, the *manager* is responsible for the recruitment and supply of (remote) workers to control threads, both initially and in the event of worker failure. Clearly, if the manager fails, then, depending on the time of failure, the fault recovery mechanism will cease or, at worst, the entire system of control thread recruitment will fail to initiate properly. Thus, the aim is to devolve this management activity to the control threads themselves, making each responsible for its own worker recruitment.

The strategy adopted is to examine the execution of the system in terms of traces of the site calls made by the processes and highlight management related communications. The idea is to use these communications as a means of identifying where/how functionality may be dispersed. In detail, the strategy proceeds as follows:

1. Focus on communication actions concerned with management. Look for patterns based on the following observation. Typically communication occurs when a process, A, generates a value, x, and communicates it to B. Identify occurrences of this pattern and consider if generation of the item could be shifted to B and the communication removed, with the "receive" in B being replaced by the actions leading to x's generation. For example:
   $A : \ldots a1, a2, a3, send(x), a4, a5, \ldots$
   $B : \ldots b1, b2, b3, receive(i), b4, b5, \ldots$
   Assume that $a2$, $a3$ (which, in general, may not be contiguous) are responsible for generation of x, and it is reasonable to transfer this functonality to B. Then the above can be replaced by:
   $A : \ldots a1, a4, a5, \ldots$
   $B : \ldots b1, b2, b3, a2, a3, (b4, b5, \ldots)_{[i/x]}$
2. The following trace subsequences are identified:
   - In control thread: $\mathit{alarm.put(i)} \gg c_i\mathit{.get} > w > \mathit{ctrlthread}_i(\mathit{pgm}, w, t) \ldots$
   - In monitor:
     $\mathit{alarm.get} > i > \mathit{rworkerpool.get} > \mathit{remw} > c_i\mathit{.put(remw)} \gg \ldots$
3. The subsequence $\mathit{rworkerpool.get} > \mathit{remw} > c_i\mathit{.put(remw)}$ of *monitor* actions is responsible for generation of a value (a remote worker) and its forwarding to a *ctrlthread* process. In the *ctrlthread* process the corresponding "receive" is $c_i\mathit{.get}$. So, the two trace subsequences are modified to:
   - In control thread:
     $\mathit{alarm.put(i)} \gg \mathit{rworkerpool.get} > \mathit{remw} > \mathit{ctrlthread}_i(\mathit{pgm},\ \mathit{remw}, t) \ldots$
   - In monitor: $\mathit{alarm.get} > i > \ldots$
4. The derived trace subsequences now include the communication of the control thread number, i from $\mathit{ctrlthread}_i$ to the *monitor*, but this is no longer required by *monitor*; so, this communication can be removed.

```
$systemD(pgm, tasks, contract, G, t) \triangleq$
    $taskpool.add(tasks)$
    $|i : 1 \leq i \leq contract : ctrlthread_i(pgm, t, G)$

$ctrlthread_i(pgm, t, G) \triangleq discover(G, pgm) > remw > ctrlprocess(pgm, remw, t, G)$

$discover(G, pgm) \triangleq let(remw)$ where $remw :\in |_{g \in G}\ g.can\_execute(pgm)$

$ctrlprocess(pgm, remw, t, G) \triangleq taskpool.get > tk >$
    (  if $valid \gg resultpool.add(r) \gg ctrlprocess(pgm, remw, t, G)$
     | if $\neg valid \gg taskpool.add(tk)$
                     $| \ discover(G, pgm) > w > ctrlprocess(pgm, w, t, G)$
    )
      where $(valid, r) :\in$
        (  $remw(pgm, tk) > r > let(true, r) \ | \ Rtimer(t) \gg let(false, 0)$  )
```

Fig. 2. Decentralized management: Orc specification

5. Thus the two trace subsequences become:
   - In control thread:
     $\ldots \gg rworkerpool.get > remw > ctrlthread_i(pgm, remw,\ t) \ \ldots$
   - In monitor: $\ldots \gg \ldots$
6. Now the specifications of the processes $ctrlthread_i$ and *monitor* are examined to see how their definition can be changed to achieve the above trace modification, and consideration is given as to whether such modification makes sense and achieves the overall goal.
   (a) In monitor the entire body apart from the recursive call is eliminated thus prompting the removal of the *monitor* process entirely. This is as would be expected: if management is successfuly distributed then there is no need for centralized monitoring of control threads with respect to remote worker failure.
   (b) In control thread the clause:
   $| \ alarm.put(i) \gg c_i.get > w > ctrlthread_i(pgm,\ w,\ t)$
   becomes
   $| \ rworkerpool.get > remw > ctrlthread_i(pgm, remw, t)$
   This now suggests that $ctrlthread_i$ requires access to the *rworkerpool*. But the *rworkerpool* is an artefact of the (centralized) manager and the overall intent is to eliminate this manager. Thus, the action *rworkerpool.get* must be replaced by some action(s), local to $ctrlthread_i$, which has the effect of supplying a new remote worker. Since there is no longer a remote worker pool, on-the-fly recruitment of an remote worker is required. This can be achieved by using a discovery mechanism similar to that of the centralized manager and replacing *rworkerpool.get* by *discover(G, pgm)*:
   $discover(G,\ pgm) \triangleq let(rw)$ where $rw :\in |_{g \in G} \ g.can\_execute(pgm)$
   (c) Finally, as there is no longer centralized recruitment of remote workers, the control thread processes are no longer instantiated with their initial remote worker but must recruit it themselves. This requires that

i. the control thread process be further amended to allow initial recruitment of a remote worker, with the (formerly) recursive body of the process now defined within a subsidiary process, *ctrlprocess*, as shown below.
ii. the parameter *remw* in *ctrlthread* be replaced by G as the control thread is no longer supplied with an (initial) remote worker, but must handle its own remote worker recruitment by reference to the grid, G.

The result of these modifications is shown in the decentralized manager specification in Fig. 2. Here each control thread is responsible for recruiting its own remote worker (using a discovery mechanism similar to that of the centralized manager specification) and replacing it in the event of failure.

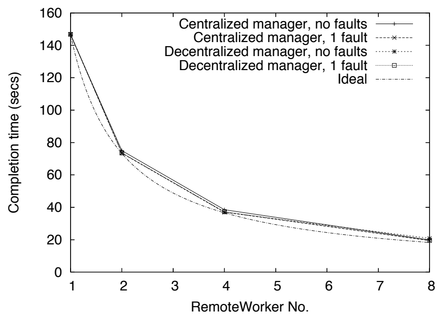
5.1 Analysis

Having derived a decentralized manager specification, the "equivalence" of the two versions must be established. In this context, equivalent means that the same input/output relationship holds, as clearly the two systems are designed to exhibit different non-functional behaviour.

The input/output relationship (i.e. functional semantics) is driven almost entirely by the *taskpool*, whose contents change dynamically to represent the data-flow execution. This execution primarily consists in establishing an on-line partial order among the execution of fireable tasks. All execution traces compliant to this partial order exhibit the same functional semantics by definition of the underlying data-flow execution model. This can be formally proved by showing that all possible execution traces respecting data-dependencies among tasks are functionally confluent (see [10] for the full proof), even if they do not exhibit the same performance.

Informally, one can observe that a global order among the execution of tasks can not be established *ex ante*, since it depends on the program and the execution environment (e.g. task duration, remote workers' availability and their relative speed, network connection speed, etc.). So, different runs of the centralized version will typically generate different orders of task execution. The separation of management issues from core functionality, which is a central plank of the muskel philosophy, allows the functional semantics of the centralized system to carry over intact to the decentralized version as this semantics is clearly independent of the means of recruiting remote workers.

One can also make an observation on how the overall performance of the system might be affected by these changes. In the centralized management system, the discovery activity is composed with the "real work" of the remote workers by the parallel composition operator: discovery can unfold in parallel with computation. In the revised system, the discovery process is composed with core computation using the sequence operator, $\gg$. This suggests a possible price to pay for fault recovery.



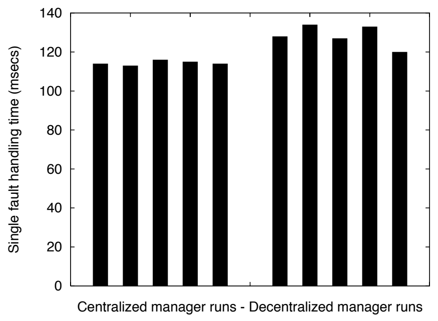


Fig. 3. Scalability (left) and fault handling cost (right) of modified vs. original muskel

6 Decentralized management: implementation

Following the derivation of the decentralized manager version outlined above, the existing muskel prototype was modified to introduce distributed fault management and to evaluate the cost, if any, in terms of performance. As shown above, in the decentralized manager, the *discovery(G, pgm, t)* parallel component of the *system(. . .)* expression become part (the *discover(G, pgm)* expression) of the *ctrlprocess(. . .)* expression. The *discovery* and *discover* definitions are not exactly the same, but *discover* is easily derived from *discovery*. Thus, the code implementing *discovery(G, pgm, t)* was moved and transformed appropriately to give an implementation of *discover(G, pgm)*. This required the modification of just one of the files in the muskel package (194 lines of code out of a total of 2575, less than 8%), the one implementing the control thread.

Experiments were run using the original and the modified versions to test the functionality and cost of the new implementation. The experiments were run on a Fast Ethernet network of Pentium III machines running Linux and Java 1.5. First the scalability of the decentralized manager version was verified: the scalabilities of the original muskel and of the one with the decentralized version of the manager were measured, with the same skeleton program and input data, to check that no overhead was introduced by the decentralized management, at least in the case where no faults were detected. Figure 3 left shows almost perfect scalability of the decentralized manager version up to 8 nodes, comparable to that achieved when using the original muskel, both in the case of no faults and in the case of a single fault per computation. Then the times spent in managing a node fault in the centralized and decentralized versions were compared. The same skeleton program with the same input data was run using the centralized and decentralized versions of muskel and a number of faults were artificially introduced into the system while the computations were running. In particular, up to 4 faults were introduced per run and the average time taken to handle a fault was measured for each of the two versions. Figure 3 right plots the average time spent in handling a single fault in each run. The centralized version performs slightly better than the decentralized one, as expected: in the centralized version the discovery of the name of the remote machines hosting the muskel RTS is performed *concurrently* with the computation, whereas it is performed *serially* to the main computation in the decentralized version. The rest of the activities performed to handle the fault (lookup of the remote worker RMI object and delivery of the macro data flow) is the same in the two cases.

7 Conclusions

The manager component of the muskel system has been re-engineered to provide distributed remote worker discovery and fault recovery. A formal specification of the component, described in Orc, was developed. The specification provided the developer with a representation of the manager that allowed exploration of its properties and the development of what-if scenarios while hiding the inessential detail. By studying the communication patterns present within the process traces, the developers were able to derive a system exhibiting equivalent core functionality, while having the desired decentralized management properties. The derivation proceeded in a series of semi-formally justified steps, with incorporation of insight and experience as exemplified by the use of expressions such as "reasonable to transfer this functionality" and "such modification makes sense".

The claim is that the creation of such a derivation facilitates exploration (and documentation) of ideas and delivers much return for small investment. Lightweight reasoning about the derived specification gave the developers some insight into the expected performance of the derived implementation relative to its parent. In addition, the authors suggest that Orc is an appropriate vehicle for the description of management systems of the sort described here. Its syntax is small and readable; its constructs allow for easy description of the sorts of activities that typify these systems (in particular the asymmetric parallel composition operator facilitates easy expression of concepts such as time-out and parallel searching); and the *site* abstraction allows clear separation of management activity from core functionality.

The approach has been applied in the context of skeletal systems where the complexity of the orchestration is constrained by the use of skeletons. However, efficient, large distributed systems often have regular structures that can be described using concise parametric definitions. Thus, one may be optimistic that the approach will be feasible for systems of significant size, although, of course, further experimentation is required to confirm this.

Future work will involve tackling the more difficult task of removing the centralized task pool bottleneck, which should provide a stiffer test of the proposed approach. And, the availability of an Orc description makes possible the analysis of system variants with respect to cost and reliability using techniques described in [15].

References

1. Danelutto, M.: QoS in parallel programming through application managers. In: Proc. of Intl. Euromicro PDP: Parallel Distributed and network-based Processing, Lugano, Switzerland, IEEE (2005) 282–289
2. Danelutto, M., Dazzi, P.: Joint structured/non structured parallelism exploitation through data flow. In: Proc. of ICCS: Intl. Conference on Computational Science, Workshop on Practical Aspects of High-level Parallel Programming. LNCS, Reading, UK, Springer (2006)
3. Danelutto, M.: Dynamic run time support for skeletons. In: Proc. of Intl. PARCO 99: Parallel Computing. Parallel Computing Fundamentals & Applications. Imperial College Press (1999) 460–467
4. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer **36** (2003) 41–50
5. White, S., Hanson, J., Whalley, I., Chess, D., Kephart, J.: An architectural approach to autonomic computing. In: Proc. of the Intl. Conference on Autonomic Computing. (2004) 2–9
6. Parashar, M., Liu, H., Li, Z., Matossian, V., Schmidt, C., Zhang, G., Hariri, S.: AutoMate: Enabling autonomic applications on the Grid. Cluster Computing **9** (2006) 161–174
7. Kennedy, K., al: Toward a framework for preparing and executing adaptive Grid programs. In: Proc. of NSF Next Generation Systems Program Workshop (IPDPS 2002). (2002)
8. Stewart, A., Gabarró, J., Clint, M., Harmer, T.J., Kilpatrick, P., Perrott, R.: Managing grid computations: An orc-based approach. In: ISPA'06. Volume 4330 of LNCS., Springer (2006) 278–291
9. Misra, J., Cook, W.R.: Computation orchestration: A basis for a wide-area computing. Software and Systems Modeling (2006) DOI 10.1007/s10270-006-0012-1.
10. Aldinucci, M., Danelutto, M.: Skeleton based parallel programming: functional and parallel semantic in a single shot. Computer Languages, Systems and Structures **33** (2007) 179–192
11. Milner, R.: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press, Cambridge (1999)
12. Agerholm, S., Larsen, P.G.: A lightweight approach to formal methods. In: FM-Trends. Volume 1641 of LNCS., Springer (1998) 168–183
13. Cole, M.: Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. Parallel Computing **30** (2004) 389–406
14. Kuchen, H.: The Muesli home page (2006) `http://www.wi.uni-muenster.de/PI/forschung/Skeletons/`.
15. Stewart, A., Gabarró, J., Clint, M., Harmer, T.J., Kilpatrick, P., Perrott, R.: Estimating the reliability of web and grid orchestrations. In: Integrated Reserach in Grid Computing, Kraków, Poland, CoreGRID, Academic Computer Centre CYFRONET AGH (2006) 141–152
16. Aldinucci, M., Danelutto, M., Kilpatrick, P.: Adding metadata to Orc to support reasoning about grid programs. In Priol, T., Vanneschi, M., eds.: Grid and Peer-To-Peer Technologies (Proc. of the CoreGRID Symposium 2007), Springer (2007)
17. Kilpatrick, P., Danelutto, M., Aldinucci, M.: Deriving Grid Applications from Abstract Models. Technical Report TR-85, CoreGRID (2007) available at `http://www.coregrid.net`.