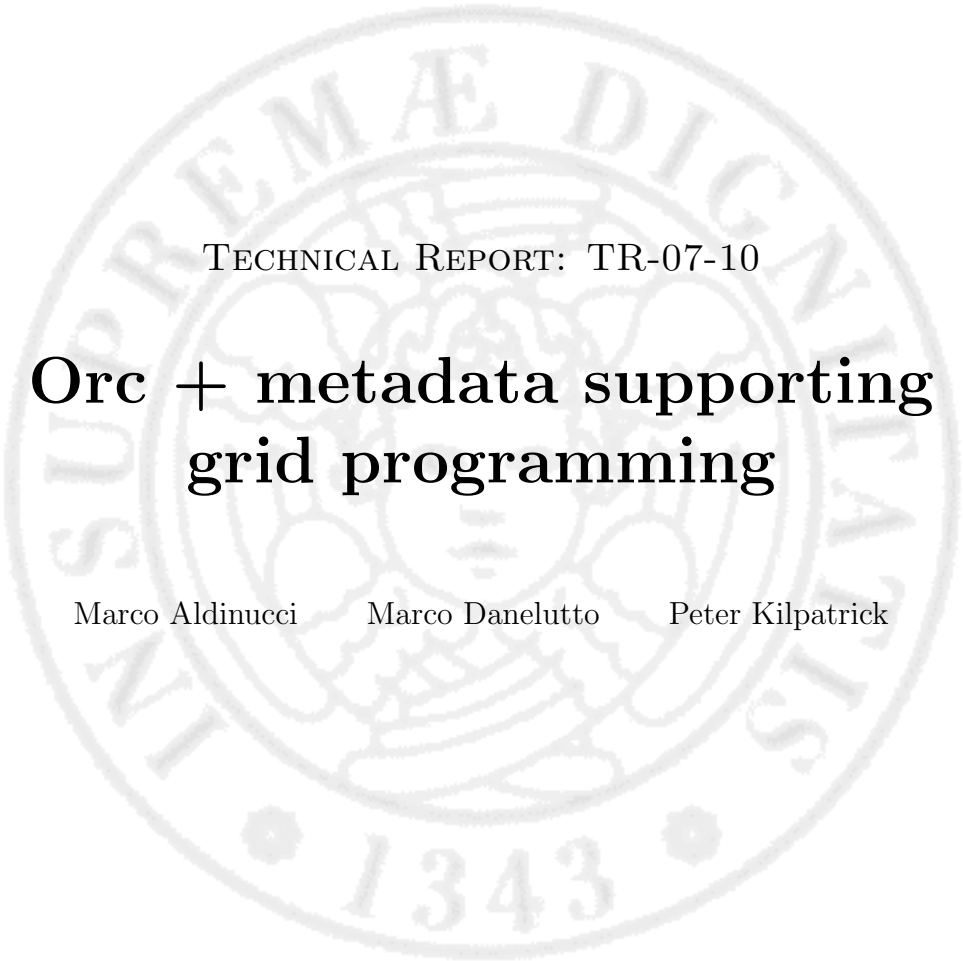
Università di Pisa
Dipartimento di Informatica



Technical Report: TR-07-10

Orc + metadata supporting grid programming

Marco Aldinucci Marco Danelutto Peter Kilpatrick

May 10, 2007

address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. tel: +39 050 2212700 fax: +39 050 2212726

Orc + metadata supporting grid programming *

Marco Aldinucci Marco Danelutto Peter Kilpatrick†

May 10, 2007

Abstract

Following earlier work demonstrating the utility of Orc as a means of specifying and reasoning about grid applications we propose the enhancement of such specifications with metadata that provide a means to extend an Orc specification with implementation oriented information. We argue that such specifications provide a useful refinement step in allowing reasoning about implementation related issues ahead of actual implementation or even prototyping. As examples, we demonstrate how such extended specifications can be used for investigating security related issues and for evaluating the cost of handling grid resource faults. The approach emphasises a semi-formal style of reasoning that makes maximum use of programmer domain knowledge and experience.

keywords: Orc, grid, metadata, fault handling, security.

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

†Department of Computer Science – Queen's University Belfast – UK.

1 Introduction

Grid computing is intended to enable the development of both industrial and scientific applications on an unprecedented scale in terms of computing power and ubiquity. However, the programming of these applications presents significant new challenges. As is widely recognized in the literature, next generation tools should enable application designers to deal transparently with dynamicity and heterogeneity of computing platforms [13]. Within the CoreGRID NoE [8], Grid application development has been envisaged as the composition of a number of coarse grained, cooperating components within a high-level programming model, which is characterized by a high-level view of compositionality, interoperability, reuse, performance and application adaptivity. This abstract view is currently directly mapped onto the *real* development of a ecosystem of components (or patterns of components), that can be deployed and connected [1, 6, 7].

Component technology focuses (by its very nature) on the decoupled development of modules implementing single features, that should then be arranged and connected to realize the application. While several frameworks for developing grid-oriented components exist or are under design [9], the models to reason about their orchestration are still inadequate. Although a model for orchestration should necessarily subsume a notion of component/module behaviour, it can be specified along a spectrum of abstraction levels: from the full implementation itself to the fully logic/algebraic description. Currently, most of the effort is concentrated on the ends of the spectum, which are far from the designer's viewpoint.

In earlier work we explored the use of Orc [12, 11] as a means of specifying and reasoning about grid computations. Orc was developed as a notation for describing the orchestration of distributed systems, rather than the core computations themselves. Orc's primitive is the *site* which may be used to abstract basic computations. A site call returns a single value or remains silent. Site calls may be combined using three composition operators (plus recursion):
Sequential : $A > x > B(x)$. For each output, x from A execute an instance of B taking x as parameter. If the output from A is not used by B this is written simply as $A \gg B(x)$
Parallel : $A \mid B$. The output is the interleaved outputs from each of A and B.
Asymmetric parallel : A *where* $x :\in B$. Execute A and B in parallel until A needs x. Take the first x delivered by B and terminate the remaining execution of B while A continues.
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.

Finally, the notation $(|i : 1 \leq i \leq 3 : worker_i)$ is used as an abbreviation for $(worker_1|worker_2|worker_3)$.

We believe that Orc lies in the middle ground of the spectrum of orchestration description: as described in previous work [4], Orc appears to be a suitable candidate to reason about some non-functional properties (e.g. fault-tolerance) of a grid-oriented *muskel* system [2]. In this paper we present a further step along the same path. We enrich Orc with *metadata* to describe non-functional properties such as deployment information. This could be used, for example, to describe the mapping of application parts (e.g. components, modules) onto a grid platform. The approach is consistent with the current trend of keeping decoupled the functional and non-functional aspects of an application. We believe that the use of metadata introduces a new dimension for reasoning about the orchestration of a distributed system by allowing a narrowing of the focus from the very general case. We expect this approach can be gracefully extended in order to allow reasoning – at design time – about several static invariants of the final implementation.

2 Orc metadata

A generic Orc program, as described in [12], is a set of Orc *definitions* followed by an Orc *goal expression*. The goal expression is the expression to be evaluated when executing the program. Assume $\mathcal{S} \equiv \{s_1, \ldots, s_s\}$ is the set of *sites* used in the program, i.e. the set of all the sites *called* during the evaluation of the top goal expression (the set does not include the pre-defined sites, such as *if* and *Rtimer*, as they are assumed to be available at any user defined site), and $\mathcal{E} \equiv \{e_0, \ldots, e_e\}$ is the set including the goal expression (e_0) and all the "head" expressions appearing in the left hand sides of Orc definitions.

The set of *metadata* associated with an Orc program may be defined as the set: $\mathcal{M} \equiv \{\mu_1, \ldots, \mu_n\}$ where $\mu_i \equiv \langle t_j, md_k \rangle$ with $t_j \in \mathcal{S} \cup \mathcal{E}$ and $md_k = f(p_1, \ldots, p_{n_k})$. f is a generic "functor" (represented by an identifier) and p_i are generic "parameters" (variables, ground values, etc.). The metadata md_k are not further defined as, in general, metadata structure depends on the kind of metadata to be represented. In the following, examples of such metadata are presented.

As is usual, the semantics of Orc is not affected when *meta*data is taken into account. Rather, the introduction of metadata provides a means to restrict the set of actual implementations which satisfy an Orc specification and thereby eases the burden of reasoning about properties of the specification. For example, restrictions can be placed on the relative physical placement of Orc sites in such a way that conclusions can be drawn about their interaction which would not be possible in the general case.

Suppose one wishes to reason about Orc program site "placement", i.e. about information concerning the relative positioning of Orc sites with respect to a given set of *physical resources* potentially able to host one or more Orc sites. Let $\mathcal{R} = \{r_1, \ldots, r_r\}$ be the set of available physical resources. Then,

given a program with $\mathcal{S} = \{siteA, siteB\}$ we can consider adding to the program metadata such as $\mathcal{M} = \{\langle siteA, loc(r_1)\rangle, \langle siteB, loc(r_2)\rangle\}$ modelling the situation where *siteA* and *siteB* are placed on distinct processing resources.

Define also the auxiliary function $location(x) : \mathcal{S} \times \mathcal{E} \rightarrow \mathcal{R}$ as the function returning the location of a site/expression and consider a metadata set *ground* if it contains location tuples relative to *all* the sites in the program.

loc metadata can be used to support reasoning about the "communication costs" of Orc programs. For example, the cost of a communication with respect to the placement of the sites involved can be characterized by distinguishing cases:

$$k_{Comm} = \begin{cases} k_{nonloc} & \text{if } location(s_1) \neq location(s_2) \\ k_{loc} & \text{otherwise} \end{cases}$$

where s_1 and s_2 are the source and destination sites of the communication, respectively and, typically, $k_{nonloc} \gg k_{loc}$.

Consider now a second example of metadata. Suppose "secure" and "unsecure" site locations are to be represented. Secure locations can be reached through trusted network segments and can therefore be communicated with taking no particular care; insecure locations are not trusted, and can be reached only by passing through untrusted network segments, therefore requiring some kind of explicit data encryption to guarantee security. This representation can be achieved by simply adding to the metadata tuples such as $\langle s_i, trusted()\rangle$ or $\langle s_i, untrusted()\rangle$. Then a costing model for communications that takes into account that transmission of encrypted data may cost significantly more than transmission of plain data can be devised.

$$k_{SecComm} = \begin{cases} k_{UnSecComm} & \text{if } \langle s_1, untrusted()\rangle \in \mathcal{M} \\ & \quad \vee \langle s_2, untrusted()\rangle \in \mathcal{M} \\ k_{Comm} & \text{otherwise} \end{cases}$$

2.1 Generating metadata

So far the metadata considered has been identified explicitly by the user. In some cases he/she may not wish, or indeed be able, to supply all of the metadata and so it may be appropriate to allow generation of metadata from partial metadata supplied by the user. For example, suppose the user provides only partial location metadata, e.g. metadata relative to the goal expression location and/or the metadata relative to the location of the components of the topmost parallel command found in the Orc program execution. Metadata information available can be used to infer ground location metadata (i.e. location metadata for all $s \in \mathcal{S}$) as follows. Consider two cases: in the first (completely distributed strategy) it is assumed that each time a new site in the Orc program is encountered, the site is "allocated" on a location that is distinct from the locations already used. In the second case (conservative strategy) new sites are allocated in the same location as their parent (w.r.t. the syntactic structure

of the Orc program), unless the user/programmer specifies something different in the provided metadata. More formally, in the first case:

$$\left.\begin{array}{l} E \triangleq f \mid g \\ E \triangleq f(x) \ \textit{where}\ x :\in g \\ E \triangleq f \gg g \\ E \triangleq f > x > g \end{array}\right\} \left\{\begin{array}{l} \langle f, loc(freshLoc(\mathcal{M})\rangle \\ \langle g, loc(freshLoc(\mathcal{M})\rangle \\ \textit{are both added to } \mathcal{M} \end{array}\right.$$

whereas in the second case:

$$\left.\begin{array}{l} E \triangleq f \mid g \\ E \triangleq f(x) \ \textit{where}\ x :\in g \\ E \triangleq f \gg g \\ E \triangleq f > x > g \end{array}\right\} \left\{\begin{array}{ll} \textit{if } \langle f, loc(X)\rangle \in \mathcal{M} & \textit{add nothing} \\ \textit{if } \langle g, loc(X)\rangle \in \mathcal{M} & \textit{add nothing} \\ \textit{if } \langle f, \bot\rangle \in \mathcal{M} & \textit{add } \langle f, location(E)\rangle \\ \textit{if } \langle g, \bot\rangle \in \mathcal{M} & \textit{add } \langle g, location(E)\rangle \end{array}\right.$$

Example To illustrate the use of metadata, consider the following description of a classical task farm (embarrassingly parallel computation):

$farm(pgm, nw) \triangleq tasksource \mid resultsink \mid workers(pgm, nw)$
$workers(pgm, nw) \triangleq \mid i : 1 \leq i \leq nw : worker_i(pgm)$
$worker(pgm) \triangleq tasksource > t > pgm > y > resultsink(y) \gg worker(pgm)$

The typical goal expression corresponding to this program will be something like $farm(myPgm, 10)$. Suppose the user provides the metadata:

$\forall i \in [1, nw] \langle worker_i, loc(PE_i)\rangle \in \mathcal{M}$
$\langle farm(myPgm, 10), strategy(fullyDistributed)\rangle \in \mathcal{M}$

where $strategy(fullyDistributed)$ means the user explicitly requires that a "completely distributed implementation" be used. An attempt to infer metadata about the goal expression identifies $location(farm(myPgm, 10)) = \bot$ but, as the strategy requested by the user is *fullyDistributed* and as $farm(pgm, nw)$ is defined as a parallel command, the following metadata is added to $\mathcal{M}$:

$\langle tasksource,\ loc(freshLoc(\mathcal{M}))\rangle$
$\langle resultsink,\ loc(freshLoc(\mathcal{M}))\rangle$
$\langle workers(pgm, nw),\ loc(freshLoc(\mathcal{M}))\rangle$.

Next, expanding the *workers* term, gives the term

$\mid i : 1 \leq i \leq nw : worker_i(pgm)$

but in this case metadata relative to $worker_i$ has already been supplied by the user. At this point

$$\begin{array}{l} \mathcal{M} = \{\ \langle tasksource,\ loc(freshLoc(\mathcal{M}))\rangle,\ \langle resultsink,\ loc(freshLoc(\mathcal{M}))\rangle, \\ \qquad \langle workers(pgm, nw),\ loc(freshLoc(\mathcal{M}))\rangle,\ \langle worker_1,\ loc(PE_1\rangle,\ \ldots, \\ \qquad \langle worker_{nw},\ loc(PE_{nw}\rangle\} \end{array}$$

and therefore is *ground* w.r.t. the program.

Thus, in addition to the location metadata provided by the user it was possible to derive the fact that the locations of *tasksource* and *resultsink* are distinct

and, in addition, are different from the locations relating to $worker_i$.

Suppose now that the user has also inserted the metadata item $\langle PE_2, untrusted() \rangle$ in addition to those already mentioned. That is, one of the placement locations is untrusted. This raises the issue of how it can be determined whether or not a communication must be performed in a secure way. This information may be inferred from the available metadata as follows. Let functions $source(C)$ denote a site "sending" data and $sink(C)$ denote a site "receiving" data in communication C. Then C *must* be secured iff

$$source(C) = X \wedge sink(C) = Y \wedge \langle X, loc(LX\rangle \in \mathcal{M} \wedge \langle Y, loc(LY\rangle \in \mathcal{M}$$
$$\wedge\ (\langle LX, untrusted()\rangle \in \mathcal{M} \vee \langle LY, untrusted()\rangle \in \mathcal{M}).$$

Thus, for the farm example above, the metadata $\langle worker_2, PE_2 \rangle$ and $\langle PE_2, untrusted() \rangle$ and the definition

$$worker_2(pgm) \triangleq tasksource > t > pgm > y > resultsink \gg worker_2(pgm)$$

together with the metadata

$$\langle tasksource, loc(TS)\rangle,\ \langle resultsink, loc(RS)\rangle,\ \langle TS, trusted()\rangle,\ \langle RS, trusted()\rangle$$

lead to the conclusion that the communications represented in the Orc code by

$$tasksource > t > pgm.compute(t)$$

and by

$$pgm.compute(t) > y > resultsink$$

within $worker_2$ must be secured.

It is worth pointing out that the metadata considered here is typical of the information needed when running grid applications. For example, constraints such as the *loc* ones can be generated to force code (that is, sites) to be executed on processing elements having particular features, and information such as that modelled by *untrusted* metadata can be used to denote those cluster nodes that happen to be outside a given network administrative domain and therefore may be more easily subject to "man in the middle" attacks or to some other kind of security related leaks.

3 Metadata exploitation: a case study

In this section we consider two alternative versions of a tool and use their Orc specifications together with metadata to analyse their performance and security properties. in Java.

`muskel` [10] is a skeleton based parallel programming environment written in Java. The `muskel` system converts a user program to a data flow graph which is represented by a *taskpool* in which the tasks (each a substantial piece of code) are the nodes of the graph. The graph is constructed in such a way that the tasks can be computed independently with the results being placed

in a *resultpool*. The actual (core) task computations are performed by a set of *remote worker* processors that are recruited for the job. Each remote worker is under the supervision of a *control thread* that accesses the *taskpool*, sends a task to its worker and places the result in the *resultpool*.

(Note: here we consider farm-style computation: more generally, `muskel` can handle a wide range of distributed applications, involving return of (intermediate) results to the *taskpool* for further computation.)

Two versions of `muskel` are presented. The first (centralized) includes a *manager* that is responsible for recruitment of remote workers, their allocation to control threads and the handling of remote worker failure. This represents the original version of `muskel`, but the presence of such a manager was seen as a potential single point of failure. [5] describes how the original specification was analysed and modified to obtain a revised version in which this single point of failure was removed by making each control thread responsible for its own remote worker recruitment (decentralized version). Here, using metadata, we examine the efficiency implications of such a policy change. Figure 1 presents the Orc specifications of the two versions for comparison.

3.1 Comparison of communication costs

In comparing the two versions of `muskel`, as is typical in such studies, the focus will be on the "steady state" performance, that is, the typical activity of a control thread when it is processing tasks. There are two possibilities: the task is processed normally and the result placed in the *resultpool* or the remote worker fails and the control thread requires a new worker. In analysing the specifications a conservative placement strategy will be assumed; that is, the sub-parts of an entity are assumed to be co-located with their parent unless otherwise stated.

Given the following metadata supplied by the developer:

$\forall rw_i \in G.\langle rw_i, loc(PE_i)\rangle \in \mathcal{M}$
$\langle system, loc(C)\rangle \in \mathcal{M}$
$\langle system(myPgm, tasks, 10, G, 50), strategy(conservative)\rangle \in \mathcal{M}$

the rules for propagation and the strategy adopted ensure that the following metadata are present for both versions:

$\langle rw_i, loc(PE_i)\rangle$, $\langle ctrlthread_i, loc(C)\rangle$, $\langle taskpool, loc(C)\rangle$, $\langle resultpool, loc(C)\rangle$, $\langle rworkerpool, loc(C)\rangle$.

In addition, for the decentralized version, $\langle cntrlprocess, loc(C)\rangle$ is present.

Normal processing For the centralized version, examination of the definition of *cntrlthread* shows that in the case of a normal calculation the following sequence of actions will occur:

$taskpool.get > tk > remw(pgm, tk) > r > let(true, r) \gg resultpool.add(r)$.

Using the metadata, and reasoning in the same way as in the farm example, it can be seen that the communication of the task tk to the remote worker and the subsequent return of the result r to the control thread represent non-local communications; all other communications in this sequence are local.

Similar analysis of the decentralized version reveals an identical series of actions for normal processing and an identical pattern of communications. Naturally then, similar results from the two versions for normal processing would be expected, and indeed this is borne out by experiment - see section 4.

Fault processing Now consider the situation where a remote worker fails during the processing of a task. In both versions the *Rtimer* timeout occurs, the task being processed is returned to the *taskpool* and a new worker is recruited.

In the centralized version the following sequence of events occurs:

$$taskpool.get \gg Rtimer(t) \gg let(false, 0) \gg taskpool.add(tk) \gg rworkerpool.get(remw)$$

while in the decentralized version the events are effectively:

$$taskpool.get \gg Rtimer(t) \gg let(false, 0) \gg taskpool.add(tk) \gg rw.can\_execute(pgm) > rw > let(g)$$

where rw is the first site in G to respond.

Analysis of these sequences together with the metadata reveals that the comparison reduces to the local communication to the *rworkerpool* in the centralized version versus the non-local call to the remote site rw in the decentralized version. This comparison would suggest that, in the case of fault handling, the centralized version would be faster than the decentralized version and, again, this is borne out by experiment.

3.2 Comparison of security costs

Consider now the issue of security. Suppose that one of the remote workers, say rw_2, is in a non-trusted location (that is $\langle PE_2, untrusted() \rangle \in \mathcal{M}$).The implications of this can be determined by analysing the specification together with the metadata. In this case, as $\langle rw_2, loc(PE_2) \rangle \in \mathcal{M}$ we can conclude that $cntrlthread_2$ will be affected (while it is operating with its initially allocated remote worker) to the extent that the communications to and from its remote-worker must be secured. This prompts reworking of the specification to split the control threads into two parallel sets: those requiring secure communications and those operating exclusively in trusted environments. In this way the effect, and hence cost, of securing communications can be minimised. Experimental results in section 4 illustrate the cost of securing the communications with differing numbers of control threads.

4 Experimental results

We ran a number of experiments, on a distributed configuration of Linux machines, aimed at verifying that the kind of results derived by working on Orc specifications of `muskel` together with metadata can be considered realistic.

We first verified that centralized and decentralized manager versions of `muskel` perform the same (up to a reasonable percentage difference) when no faults occur in the resources used for remote program execution. The following table presents completion times (in seconds) for runs of the same program with the two `muskel` versions, on a variable amount of remote resources. The average difference between the centralized and decentralized version was 1% ($\sigma = 1.2$), demonstrating the two versions are substantially equivalent in the case of no faults.

`muskel` version	*1 PE*	*2 PEs*	*3 PEs*	*4 PEs*
Centralized manager	146.46	74.02	38.53	19.86
Decentralized manager	146.93	73.64	36.93	20.87
Difference	0.4%	-1.2%	-1.6%	1.0%

Then we considered what happens in the case where remote resources fail. The table below shows the time spent in handling a single fault (in msecs) in 5 different runs. The decentralized manager `muskel` version takes longer to handle a single fault, as expected. Moreover, the standard deviation of the time spent handling the single fault is $\sigma = 1.1$ in the centralized case but it is $\sigma = 5.6$ in the decentralized case, reflecting the fact that in this case we need to interact with remote resources to recruit a new worker and the time spent depends on the location of the remote resource recruited.

`muskel` version	Run_1	Run_2	Run_3	Run_4	Run_5	*Average*
Centralized manager	114	113	116	115	114	114.4
Decentralized manager	128	134	127	133	129	128.4

We also measured the effect of selectively deciding which `muskel` remote workers have to be handled by means of secure (SSL based, in this case) communications exploiting the metadata provided by the user/programmer. The plot in Figure 2 shows the completion time of a `muskel` program whose remote worker sites are running on a variable mix of *trusted* and *untrusted* locations. The more *untrusted* locations are considered, the poorer the scalability that is achieved. This demonstrates that metadata exploitation to identify the minimal set of remote workers that actually *need* to be handled with secure communications can be very effective.

Part of the experimental results presented here have been generated *prior* to the development of the Orc based techniques discussed in this work. They were aimed at verifying exactly the differences in the centralized and distributed manager versions of `muskel` and the impact of adopting secure communications with remote `muskel` workers. This required substantial programming effort, debugging and fine tuning of the new `muskel` versions and, last but not least, extensive

grid experimental sessions. The approach discussed in this work allowed us to achieve the same qualitative results by just developing Orc specifications of the `muskel` prototypes, enhancing these specifications with metadata and then reasoning with these enhanced specifications, without actually writing a single line of code and without the need for running experiments on a real grid.

5 Conclusions

We have shown how, by associating metadata with an Orc specification, we can reason about the specification and that this reasoning carries through to the actual grid code which implements the specification. In particular, we considered how user provided metadata can be associated with the Orc model of a real structured grid programming environment (`muskel`) and showed how this could be used to perform qualitative performance comparison between two different versions of the programming environment, as well as to determine how the overhead introduced by security techniques can be minimized. We compared these theoretical results with actual experimental results and we verified they qualitatively match. Thus, the availability of an Orc model on which to "hang" the metadata allows metadata to be exploited *before* the actual implementation is available.

We are currently working to formalize and automate the techniques discussed in this work. In particular, we are aiming to implement tools to support the reasoning procedures adopted. The whole approach, based on Orc, as described here and in the cited companion papers [4, 3] encourages the usage of semi-formal reasoning to support program development (both program design and refinement) and has the potential to substantially reduce experimentation by allowing the exploration of alternatives prior to costly implementation.

References

[1] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for high performance grid programming in grid.it. In V. Getov and T. Kielmann, editors, *Proc. of the Intl. Workshop on Component Models and Systems for Grid Applications*, CoreGRID series, pages 19–38, Saint-Malo, France, Jan. 2005. Springer.

[2] M. Aldinucci and M. Danelutto. Algorithmic skeletons meeting grids. *Parallel Computing*, 32(7):449–462, 2006. DOI:10.1016/j.parco.2006.04.001.

[3] M. Aldinucci, M. Danelutto, and P. Kilpatrick. A framework for prototyping and reasoning about grid systems. In *Proc. of PARCO 2007: Parallel Computing*, Jülich, Germany, Sept. 2007. To appear.

[4] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Management in distributed systems: a semi-formal approach. In *Proc. of 13th Intl. Euro-Par 2007*

Parallel Processing, LNCS, Rennes, France, Aug. 2007. Springer. To appear.

[5] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Management in distributed systems: a semi-formal approach. Technical Report TR-07-05, Università di Pisa, Dipartimento di Informatica, Feb. 2007.

[6] M. Alt, J. Dünnweber, J. Müller, and S. Gorlatch. HOCs: Higher-order components for grids. In *Component Models and Systems for Grid Applications*, CoreGRID, pages 157–166. Springer, Jan. 2005.

[7] F. Baude, D. Caromel, and M. Morel. On hierarchical, parallel and distributed components for grid programming. In V. Getov and T. Kielmann, editors, *Proc. of the Intl. Workshop on Component Models and Systems for Grid Applications*, CoreGRID series, pages 97–108, Saint-Malo, France, Jan. 2005. Springer.

[8] *The EU FP6 CoreGRID Network of Excellence*, 2007. `http://www.coregrid.net/`.

[9] CoreGRID NoE deliverable series, Institute on Programming Model. *Deliverable D.PM.04 – Basic Features of the Grid Component Model (assessed)*, Feb. 2007.

[10] M. Danelutto and P. Dazzi. Joint structured/non structured parallelism exploitation through data flow. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra, editors, *Proc. of ICCS: Intl. Conference on Computational Science, Workshop on Practical Aspects of High-level Parallel Programming*, LNCS, Reading, UK, May 2006. Springer.

[11] D. Kitchin, W. R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In C. Baier and H. Hermanns, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 477–491. Springer, 2006.

[12] J. Misra and W. R. Cook. Computation orchestration: A basis for a wide-area computing. *Software and Systems Modeling*, 2006. DOI 10.1007/s10270-006-0012-1.

[13] Next Generation GRIDs Expert Group. *NGG3, Future for European Grids: GRIDs and Service Oriented Knowledge Utilities. Vision and Research Directions 2010 and Beyond*, Jan. 2006.

$$
\begin{array}{l}
systemCentrManager(pgm, tasks, contract, G, t) \triangleq \\
\quad taskpool.add(tasks) \mid discovery(G, pgm, t) \mid manager(pgm, contract, t)
\end{array}
$$

$$
\begin{array}{l}
discovery(G, pgm, t) \triangleq (|_{g \in G} \ (\ \mathsf{if}\ remw \gg rworkerpool.add(remw) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{where}\ remw :\in \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\ \ g.can\_execute(pgm) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid Rtimer(t) \gg let(false)\ )\ \ ) \\
\qquad\qquad\qquad\qquad\quad) \gg discovery(G, pgm, t)
\end{array}
$$

$$
\begin{array}{l}
manager(pgm, contract, t) \triangleq \\
\quad |i : 1 \leq i \leq contract : (rworkerpool.get > remw_i > ctrlthread_i(pgm, remw_i, t))
\end{array}
$$

$$
\begin{array}{l}
ctrlthread_i(pgm, remw, t) \triangleq taskpool.get > tk > \\
\quad (\ \ \mathsf{if}\ valid \gg resultpool.add(r) \gg ctrlthread_i(pgm, remw, t) \\
\qquad \mid \mathsf{if}\ \neg valid \gg ((taskpool.add(tk) \mid rworkerpool.get) > w > ctrlthread_i(pgm, w, t)) \\
\quad) \\
\qquad \mathsf{where}\ (valid, r) :\in \\
\qquad\quad (\ \ remoteworker(pgm, tk) > r > let(true, r) \\
\qquad\qquad \mid Rtimer(t) \gg let(false, 0) \\
\qquad\quad)
\end{array}
$$

$$
\begin{array}{l}
systemDistribManager(pgm, tasks, contract, G, t) \triangleq \\
\quad taskpool.add(tasks) \mid i : 1 \leq i \leq contract : ctrlthread_i(pgm, t, G)
\end{array}
$$

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

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

$$
\begin{array}{l}
ctrlprocess(pgm, rw, t, G) \triangleq taskpool.get > tk > \\
\quad (\ \ \mathsf{if}\ valid \gg resultpool.add(r) \gg ctrlprocess(pgm, rw, t, G) \\
\qquad \mid \mathsf{if}\ \neg valid \gg taskpool.add(tk) \\
\qquad\qquad\qquad \mid discover(G, pgm) > w > \\
\qquad\qquad\qquad\quad ctrlprocess(pgm, w, t, G) \\
\quad) \\
\qquad \mathsf{where}\ (valid, r) :\in \\
\qquad\quad (\ \ remoteworker(pgm, tk) > r > let(true, r) \\
\qquad\qquad \mid Rtimer(t) \gg let(false, 0)\ \ )
\end{array}
$$

Figure 1: `muskel` centralized and decentralized manager specifications in Orc

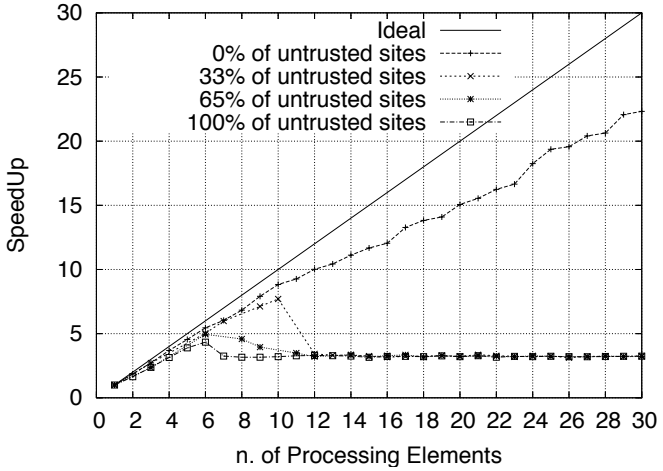


Figure 2: Comparison of runs involving different percentages of *untrusted* locations.