Semi-formal Models to Support Program Development: Autonomic Management within Component Based Parallel and Distributed Programming

M. Aldinucci[1], M. Danelutto[2], and P. Kilpatrick[3]

[1] Dept. Computer Science, Univ. of Torino
[2] Dept. Computer Science, Univ. of Pisa
[3] Dept. Computer Science, Queen's Univ. Belfast

Abstract. Functional and non-functional concerns require different programming effort, different techniques and different methodologies when attempting to program efficient parallel/distributed applications. In this work we present a "programmer oriented" methodology based on formal tools that permits reasoning about parallel/distributed program development and refinement. The proposed methodology is *semi-formal* in that it does not require the exploitation of highly formal tools and techniques, while providing a palatable and effective support to programmers developing parallel/distributed applications, in particular when handling non-functional concerns.

Keywords: program modelling, rewriting, non-functional concerns, performance tuning, autonomic computing.

1 Introduction

Modern distributed systems including grids, clouds and, more generally, service oriented architectures, are characterized by heterogeneity and dynamism in the sense of failure, delays and the varying availability of services. They therefore pose new challenges to the programmer of parallel/distributed applications.

In particular, when developing a parallel/distributed application, a programmer has to deal with two distinct kinds of concern: functional and the non-functional (a.k.a. extra-functional) concerns. Functional concerns are those related to *what* has to be computed, i.e. to the algorithm defining the result as a function of the input data. Non-functional concerns are those related to *how* the result has to be computed, i.e. to the techniques needed to implement the algorithm in an efficient way on the parallel/distributed architecture at hand. Examples of typical non-functional concerns include performance tuning, fault tolerance, security and power management.

In fact, programming the non-functional part of a distributed application is frequently much more demanding than programming the functional part. Programming the functional part of these applications requires sound knowledge of

F.S. de Boer et al. (Eds.): FMCO 2008, LNCS 5751, pp. 204–225, 2009.

the application field and of the algorithms that can be used to solve the problem at hand. This knowledge is normally in the repertoire of the application programmer. The situation is significantly different for non-functional concerns. In this case, specific knowledge related to the target architecture is required in order to develop efficient solutions/implementations solving the problems related to non-functional concern management. For example, if load balancing is to be achieved in the computation of some embarrassingly parallel application, the overall architecture of the target machine (shared memory vs. distributed memory, high vs. low bandwidth (latency) interconnection network, etc.) must be known to tackle effectively the load balancing. Also, the techniques used to manage non-functional concerns are often significantly different from those used to address functional concerns. The "normal" application programmer, however, usually has in his background neither specific knowledge related to the target architecture nor knowledge related to the particular techniques needed to tackle non-functional concerns.

It is therefore commonly recognized that, ideally, functional concerns should be the responsibility of the application programmer, i.e. the programmer with specific knowledge on the application field, whereas the non-functional concerns should be addressed by system programmers, i.e. the programmers with specific knowledge of the target architecture and of the techniques and peculiarities of particular non-functional concerns. In the terminology of Aspect-Oriented Programming, non-functional concerns represent cross-cutting concerns w.r.t. functional ones, and thus typically require orthogonal techniques and experience.

The remainder of the paper is structured as follows: Sec.2 further discusses the functional/non-functional aspects in parallel and distributed programming and Sec. 3 introduces behavioural skeletons and GCM, the Grid Component Model by CoreGRID where these concepts were first introduced. Then Sec. 4 introduces Orc, the formal model we use in our semi-formal program development support methodology. Finally, Sections 5 to 7 discuss how the semi-formal methodology supports reasoning about alternative implementations (5), autonomic management strategy design (6) and metadata usage to evaluate again alternative implementations (7).

2 Addressing Functional and Non-functional Concerns

In this work we consider some typical non-functional concerns that have to be managed when developing parallel and distributed applications on modern architectures and we propose a methodology based on semi-formal use of formal models and tools to support design, refinement, improvement and in general reasoning about non-functional concerns in parallel and distributed applications. However, as will be seen, here we do not take a classical approach to non-functional concern management.

In both sequential and concurrent programming, coding for a specific non-functional behaviour to achieve a given QoS goal was evident three decades ago. The software engineering solution to achieve it was to introduce levels of

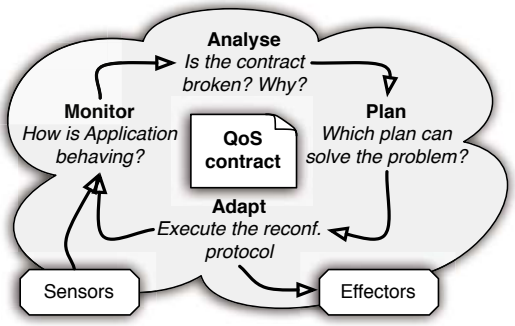

Fig. 1. Basic control loop for autonomic management

abstraction, effectively yielding a tree of refinements, from the problem specification to alternative target programs [1]. The derivation of a target program then follows a path down this tree. The transition from one node to the next can be described formally by a semantics-preserving program transformation or refinement. Conceptually, porting a program to a different execution platform configuration and/or QoS specification means backtracking to a previous node on the path and then following another path to a different target program. Typically, the goal is achieved according to the spiral model by way of a number of tuning iterations [2]. In this, the real extent of non-functional flexibility is often experienced ex-post. Commonly, the cost of some of those iterations turns out to be unacceptably high, thus reducing the potential market of the applications. Traditionally, the design backtracking happens off-line because it requires the partial re-design of the code. This makes the approach completely unsuited to capturing variation points modelling run time events or dynamic changes in the required QoS. Moreover, the design backtracking cost is directly related to the frequency of non-functional adaptations.

An alternative approach consists in moving the non-functional concern handling into an autonomic manager associated with the functional application code. This autonomic manager, implementing a control loop such as that depicted in Fig. 1, moves the choice of different design alternatives to launch or run time. These alternatives may have been either fully or partially abstracted out during the static design of the application.

In this work we assume this latter approach. Thus we consider that non-functional concerns are dealt with within autonomic managers as is the case in *behavioural skeletons* – introduced in Sec. 3 – which can be considered as code factories in the form of high-order, parametric components that can be dynamically adapted along a predefined schema that is dynamically instantiated by a previously unknown QoS contract. Having restricted the domain to autonomic management of non-functional concerns *á la behavioural skeleton*, we introduce *semi-formal* reasoning, i.e. a semi-formal way to use formal models [3,4,5] and we demonstrate how several semi-formal techniques can be used to support the design, development and refinement of autonomic managers dealing with non-functional concerns in parallel and distributed applications.

3 Components and Behavioural Skeleton

Behavioural skeletons are component abstractions that capture both the functional and non-functional behaviour of some component assemblies, each of them specialised to solve one or more management goals, such as configuration, optimisation, healing and protection. Given a component model, these paradigms can be represented as parametric schema of wiring and/or nesting. The concept of behavioural skeleton was originally introduced to bring autonomic features within the *Grid Component Model* (GCM); however, since it is more abstract than the component model itself, it can be used in any component model admitting the dynamic reconfiguration of component assemblies.

3.1 The Grid Component Model (GCM)

The *Grid Component Model* (GCM) is a hierarchical component model explicitly designed to support component-based autonomic applications in distributed contexts. GCM allows component interactions to take place with several distinct mechanisms. In addition to classical "RPC-like" use/provide ports, GCM allows streaming ports and collective interaction patterns to be used in component interaction. GCM disciplines the life-cycle of components, which can be dynamically created, destroyed, bound to and unbound from assemblies. These distinguished features makes GCM particularly suitable for modelling distributed and dynamically adaptable applications. The full specification of GCM can be found in [6].

GCM is assumed to provide several levels of autonomic managers in components; they monitor and steer the non-functional features of the component programs. GCM components thus have two kinds of interfaces: functional and non-functional ones. The functional interfaces host those ports concerned with implementation of the functional features of the component. The non-functional interfaces host those ports needed to support the component management activity in the implementation of the non-functional features, i.e. those features contributing to the efficiency of the component in obtaining the expected (functional) results but not directly involved in result computation. Each GCM component contains an *Autonomic Manager* (AM), interacting with other managers in other components via the non-functional interfaces.

In this vision, the AM can reconfigure the assembly of its managed components to pursue a QoS goal. This typically happens if one of its *plans* is foreseen to be effective in re-establishing the validity of the QoS contract. Alternatively, the AM can contact a number of the other AMs in order to set up a cooperative reconfiguration plan, which will involve the union of managed components. In both cases, the AM may induce a structural reconfiguration of the component assembly through a number of *functionally equivalent* component assemblies.

The design of those plans is clearly a critical step for the effectiveness of the whole process. Two key aspects come into play:

1. the "creative" exploration of possible equivalent design alternatives, their aggregation and variation points, and their non-functional profile;
2. the checking of their functional equivalence.

While formal tools are useful for the second aspect, they are not very effective for addressing the first. The use of behavioural skeletons also address the second point since they represent, by definition, families of functionally equivalent assemblies. In this case the issue is raised at the skeleton design time, i.e. reduced to the first aspect. In this paper we advocate the use of a semi-formal methodology to address the first aspect. The methodology uses Orc as specification tool.

3.2 Behavioural Skeletons

Behavioural skeletons represent a specialisation of the algorithmic skeleton concept for component management [7]. Algorithmic skeletons have been traditionally used as a vehicle to provide efficient implementation templates of parallel paradigms. Behavioural skeletons, as algorithmic skeletons, represent patterns of parallel computations (which are expressed in GCM as graphs of components), but in addition they exploit the inherent skeleton semantics to design sound self-management schemes of parallel components.

As shown in Fig. 2, behavioural skeletons are composed of an algorithmic skeleton together with an autonomic manager and provide the programmer with a component that can be turned into a running application by providing the code parameters needed to instantiate the algorithmic skeleton parameters (e.g. the different stages in a pipeline or the workers in a farm) and some kind of Service Level Agreement (SLA, e.g. the expected parallelism degree or the expected throughput of the application). The choice of the skeleton to be used as well as the code parameters provided to instantiate the behavioural skeleton are functional concerns, while the autonomic management itself is a non-functional concern. In turn, the implementation of both the algorithmic skeleton and the autonomic manager is in the charge of the "system" programmer, i.e. the one providing the behavioural skeleton framework to the application user, while the instantiation of the behavioural skeleton is in the charge of the application programmer.

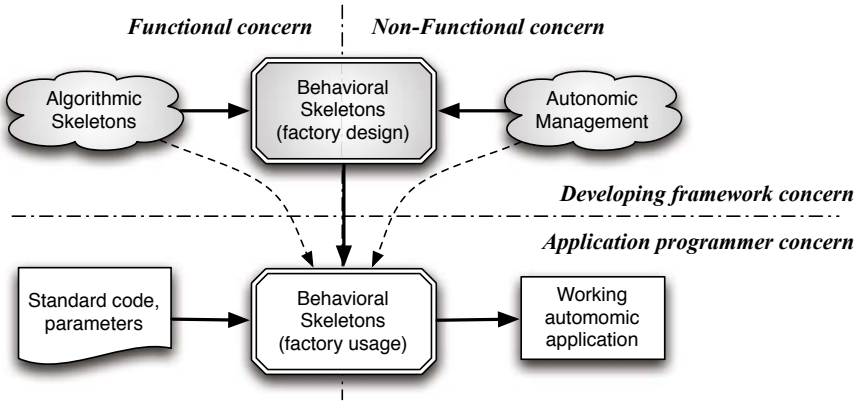

Fig. 2. Behavioural skeleton rationale

Autonomic management of non-functional concerns is based on the concurrent execution (with respect to the application "business logic") of a basic control loop such as that shown in Fig. 1. In the *monitor* phase, the application behaviour is observed, then in the *analyse* and *plan* phases the observed behaviour is examined to discover possible malfunctioning and corrective actions are planned. The corrective actions are usually taken from a library of known actions and the chosen action is determined by the result of the analysis phase. Finally, the actions planned are applied to the application during the *execute* phase [8,9,10,4].

Component technology, promotes the engineered development of distributed autonomic applications by enabling the co-design of autonomic management of non-functional concerns (performance tuning, in particular) and parallelism exploitation, which can be just-in-time derived from well-known, efficient patterns, such as behavioural skeletons [11].

In a component assembly, the autonomic management ultimately aims to induce non-functional alterations of the component assembly, which may translate into structural alterations of the component assembly. This means that an application is really described by an *evolving assembly* of components, i.e. an initial assembly and all its possible evolutions across the iterations of the adaptation phase. These reconfigurations of the assembly should be *formally* specified (at least) because they should be encoded in the manager. In addition, since in an autonomic system the management is inherently non-centralized, these reconfiguration should be *locally* specified, whereas the *global* evolution of the system is distributively realized via the cooperation of managers.

The formal description of evolving assemblies of processes, services and components has been the subject of active research in the global computing community [12]. Some of the results achieved in that community have also been cast to formal specification of the evolving assembly of autonomic components (see Sec. 8). However, the fully-fledged formal treatment of them requires enrichment of the model with many details that rapidly bring the complexity beyond reasonable (human) limits.

For this reason, we advocate the idea of *semi-formal* reasoning, i.e. a semi-formal way to reason about the equivalence of formal specifications [3,4,5]. Here, the main idea is to develop a, possibly partial, formal specification of a component assembly described using some formal tool such as Orc [13]. The specification provides the developer with a representation of the assembly and management overlay which allows exploration of their properties and the development of what-if scenarios while hiding the inessential detail. By studying the communication patterns present within the Orc process traces, the designer is able to derive for some paradigmatic assemblies (e.g. behavioural skeletons) an alternative structure which maintains core functionality, while allowing variation of non-functional behaviour, and thus different QoS. The derivation proceeds in a series of semi-formally justified steps, with incorporation of insight and experience as exemplified by the inclusion of expressions such as "reasonable to transfer this functionality" and "such modification makes sense".

4 Tools to Support Reasoning about Autonomic Management

As stated above, in modern parallel and, in particular, distributed systems much of the challenge lies in composing the various units of core functionality, rather than in implementation of the core functionality itself. Typically non-functional properties of an application depend on the overall "shape" of the system and this has led to an increased emphasis on orchestration: different designs of an application may be used to obtain different non-functional properties. A developer may, at design time, wish to explore the nature of different designs in terms of non-functional properties. Moreover, it is increasingly the case that *dynamic* adaption of the system design is required in response, for example, to differing resource availability, differing security considerations and so on. (Indeed, functional properties may lead to demand for architecture change: the occurrence of a hot-spot in processing data may require the addition of further resources, for example, to maintain a throughput requirement). In essence, this requirement for dynamic change is the raison d'être for autonomic management. The development of such dynamic systems requires means to describe both functional and non-functional concerns in relation to different designs; and means to support argument that the change induced by an autonomic system in response to, for example, environmental change, maintains functionality while adapting non-functional properties to the new conditions.

The need to explore different designs and their relation to differing non-functional properties motivates the search for a notation to be used as a vehicle for such investigation. We sought a notation which would

1. be oriented toward orchestration of components providing core functionality, rather than the core functionality itself;
2. allow an operational-style description of a system so that different designs could be described;
3. ideally, have a simple syntax and well-defined semantics so that properties of systems could be described and reasoned about with relative ease.

To this end we identified Orc [13] by Misra and Cook as a suitable candidate: Orc is an orchestration language which abstracts core functionality as site calls (see Sec. 4.1); it is operational in nature and provides a very small range of constructs and these are oriented toward describing the key aspects of concurrent/distributed systems. Thus it fits with our philosophy and lends itself to the level of reasoning that we wished to pursue: that is, a semi-formal style of reasoning in which one benefits from the clean, abstract, semantically well-founded description mechanism provided, but shies away from fully-formal proofs of general properties. Generally, we are content to prove properties that hold in particular situations and to draw upon insight and experience to allow conclusions to be drawn that are not fully supported by formal argument.

4.1 Orc

Orc is a language for distributed and concurrent programming that 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

$$(|i : 1 \leq i \leq 3 : worker_i)$$

is used as an abbreviation for

$$(worker_1|worker_2|worker_3).$$

In Orc 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 [13]. *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.

5 Sample "Semi-formal" Usage of Orc

As an example of the way Orc can be used to support reasoning about parallel/distributed programs we consider the reverse engineered model of the `muskel` interpreter as derived in [3].

$$
\begin{array}{l}
system(pgm, tasks, contract, G, t) \triangleq \\
\quad taskpool.add(tasks) \\
\quad \mid discovery(G, pgm, t) \\
\quad \mid manager(pgm, contract, t) \\
\\
discovery(G, pgm, t) \triangleq (\mid_{g \in G} \ (\ \mathsf{if}\ remw \neq false \ \gg rworkerpool.add(remw) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{where}\ remw :\in \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\ \ g.can\_execute(pgm) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid Rtimer(t) \gg let(false)\ ) \\
\qquad\qquad\qquad\qquad\qquad)) \\
\qquad\qquad\qquad\qquad) \gg discovery(G, pgm, t) \\
\\
manager(pgm, contract, t) \triangleq \\
\quad \mid i : 1 \leq i \leq contract : (rworkerpool.get > remw > ctrlthread_i(pgm, remw, t)) \\
\quad \mid monitor \\
\\
ctrlthread_i(pgm, remw, t) \triangleq taskpool.get > tk > \\
\quad (\ \ \mathsf{if}\ valid \ \gg \ resultpool.add(r) \ \gg \ ctrlthread_i(pgm, remw, t) \\
\quad\ \mid \mathsf{if}\ \neg valid \ \gg \ (\ taskpool.add(tk) \\
\qquad\qquad\qquad\qquad \mid alarm.put(i) \ \gg \ c_i.get > w > ctrlthread_i(pgm, w, t) \\
\qquad\qquad\qquad\quad) \\
\quad) \\
\qquad \mathsf{where}\ (valid, r) :\in \\
\qquad\quad (\ \ remw(pgm, tk) > r > let(true, r) \ \mid Rtimer(t) \ \gg \ let(false, 0) \ \ ) \\
\\
monitor \triangleq alarm.get > i > rworkerpool.get(remw) > remw > c_i.put(remw) \\
\qquad\qquad \gg \ monitor
\end{array}
$$

Fig. 3. Reverse engineering of the `muskel` prototype

`muskel` is a full Java skeleton programming environment under development at the University of Pisa[1] since the early '00s [9]. The `muskel` environment can execute in parallel stream-parallel skeleton programs on networks/clusters/grids of Java enabled workstations. A simple autonomic manager maintains "best effort"—a performance contract (parallelism degree) provided by the user—in the presence of faulty or malfunctioning processing elements. In fact, autonomic managers were first implemented in `muskel` and then moved and greatly extended in the behavioural skeleton research framework.

When a `muskel` skeleton program is run, the `muskel` framework scans the available network looking for processing nodes hosting a `muskel` runtime system and recruits a number of these resources to execute the program. The number of resources recruited is as close as possible to the parallelism degree requested by the user via a *performance contract* provided with the program code. Then, the recruited resources are used to compute tasks appearing on the program input stream. In particular, an instance of the distributed macro data flow interpreter used in `muskel` to execute skeleton programs is used on each of the resources recruited.

[1] See `http://cotognata.di.unipi.it/~marcodanelutto/wiki/doku.php?id=muskel`

The `muskel` prototype is written in Java and uses RMI to interact with remote interpreter instances and UDP multicast to discover available resources in the network. The full `muskel` environment amounts to some 5K lines of code.

In Fig. 3 we show the "reverse engineering" of the `muskel` prototype in Orc. The Orc code here presents all the significant features of the actual prototype. This code has been manually derived from the actual Java code of the `muskel` prototype. A first version of the Orc code was written, which was much more complex than that of Fig. 3. This version was then refined to produce that of Fig. 3. No specific tools were used in this process, but most of the techniques outlined in this work relating to transformation and manipulation of Orc programs were used.

The discovery process, performed in parallel with the complete execution of the skeleton program, is modelled by the process behind the $discovery(G, pgm, t)$ expression. G represents the *grid* environment on which the program executes, *pgm* is the skeleton program itself, and t is the timeout delay before initiating another discovery action.

The autonomic manager action is modelled by the *manager(pgm,contract,t)* term. The manager starts a pool of *contract* control threads. Each of the control threads is in charge of fetching fireable macro data flow instructions[2] from the task pool and executing them on the remote interpreter instance (*remw* in the control thread) associated with the control thread. The manager also starts a *monitor* process in charge of getting a new remote resource from the discovery process and launching a new control thread when a previously running control thread terminates upon discovery of failure of the associated remote interpreter.

This Orc code can be understood much more readily than the actual Java `muskel` implementation and can be used to investigate properties of the implementation. In fact, in [3] it has been used to derive a new version of `muskel` where the potential bottleneck represented by the centralized discovery service has been removed. The new version was derived in three steps:

- First, the Orc code was analysed looking for possible modifications that may be used to remove the bottleneck. In fact we first analysed process traces to aid understanding of the interactions involved and, using insight gleaned from this, identified functionality that could be shifted between processes to achieve the desired non-functional goal—removal of the bottleneck—while retaining the functional behaviour.
- Then a new Orc model was written with the bottleneck removed—with a discovery service distributed among the control threads.
- Finally, the actual `muskel` code was modified in accordance with the model redesign to produce a new decentralized discovery version.

The whole process allowed us to postpone all Java related coding until a feasible solution had been identified and modelled in Orc. The modified version of the Java `muskel` prototype fulfilled the expectations of its Orc model.

[2] That in turn derive from the compilation of the `muskel` skeleton program.

The technique used to derive the new Orc model of autonomic management and discovery in `muskel` uses *traces* derived from Orc computations. In particular, the approach followed to derive the new manager/discovery structure in the `muskel` interpreter is the following:

- we take the Orc description of the `muskel` interpreter and expand terms so as to obtain *traces* modelling the evolution of the different parallel/distributed computations involved;
- we match traces by identifying matching pairs of send receive statements;
- we try to merge these traces into a single trace by collapsing send/receive pairs and moving item generation accordingly;
- we finally reverse-engineer an Orc expression that generates the resulting trace.

This process is effectively the application of a rule such as:

$$\frac{a > x > ch.put(x) > R) \mid (\ldots \gg ch.get() > y > S)}{R \mid \ldots \gg a > y > S} \tag{1}$$

where R should be a term with no occurrence of x. Rule 1 states that part of process A leading to the generation of a value x eventually sent to process B can be moved to process B in place of the actions receiving the x value from A, provided x is not needed in the continuation of A.

The same procedure will be used in Sec. 6 to validate skeleton transformation rules used within behavioural skeleton autonomic managers in a completely different context. It is worth pointing out the kind of usage made of Orc here: we use a formal notation to develop an abstract version of the code needed to implement the application at hand. The programmer can then reason on the abstract version in terms of mechanisms and tools close to his background: computations, traces, pairing of communication primitives, etc. Eventually, when something satisfactory from the viewpoint of the goal he had in mind has been achieved in the abstract code, this solution can be programmed with the actual programming tools at hand, that are much more difficult to manage properly and require a significantly more substantial effort than "programming" with Orc.

6 Demonstrating the Validity of Autonomic Management Policy with Semi-formal Reasoning in Orc

In this section we illustrate in more detail the kind of reasoning we have found useful with models expressed in Orc. We first introduce a model of the autonomic managers used in GCM behavioural skeletons (as discussed in Sec. 3). Then we introduce the skeleton structured programming model defined through behavioural skeletons and we provide an Orc modelling of the skeletons used. Finally, we show how we can justify the source-to-source transformations applied by autonomic managers of behavioural skeletons taking care of the performance tuning of an application.

6.1 Modelling Autonomic Management

We introduce an Orc model of the autonomic management activities in behavioural skeletons. Any autonomic manager in a GCM behavioural skeleton can be modelled by the following Orc code:

$$\begin{aligned} Mgr(Sk, SLA) = {} & distribute(Sk, SLA) > s > \\ & monitor(s) > m > analyse(s, m) > (b, p, v) > \\ & ((if(b) \gg adapt(s, p) > s1 > Mgr(s1, SLA)) \\ & \mid (if(\sim b) \gg raise(v) > Mgr(s, SLA)) \end{aligned}$$

where Sk is the skeleton program derived from the behavioural skeleton nesting used by programmers to model their application and SLA is the contract the user specifies/requires to be ensured. The manager structure clearly reflects the control cycle outlined in Fig.1. During the *adapt* phase, a new version of the original Sk program may be produced to adapt the program to the dynamic change in either the target architecture or in the computation, as perceived from the *monitor* phase. This new version may differ from the pervious one, either by some non-functional feature (e.g. a varied number of workers in the implementation of a task farm skeleton) or by some functional feature (e.g. a varied parallelism exploitation pattern). In this latter case, the varied pattern will be one among the possible rewritings of the original skeleton program Sk that preserve the functionality of the application while (possibly) improving some non-functional feature. The new version of the program—$s1$—is eventually used to call recursively the Mgr. If the *analyse* phase does not succeed in finding a corrective plan for a malfunction perceived through the *monitor* phase, a violation is raised to the upper levels of management (upper level autonomic managers in the case of a hierarchy of behavioural skeletons, or to the user if this is the top level manager).

For example, if in the *analyse* phase the manager discovers that the user defined SLA cannot be guaranteed due to the too fine grain of two consecutive pipeline stages in Sk it may consequently decide to apply a stage merging rule (i.e. a rule merging the computation of two consecutive pipeline stages at the same computing element)[3] in the *adapt* phase, and therefore restart with deployment (*distribute*) of the (possibly new) SLA related to the new program version $s1$ with the collapsed stages.

Another notable case of *adapt*ation is represented by the variation of non-functional features of the skeleton program in execution—typically, variations of the parallelism degree used when implementing task farms. If in the *analyse* phase the manager discovers that there is a farm with a small inter arrival time for input tasks and a longer service time, its parallelism degree can be increased—new workers can be added—to improve the overall program efficiency [14].

Once more, the Orc model allows system designers to reason about the logical behaviour of the system at hand without needing to resort to analyzing the actual implementation code. In the following sections, we will show how Orc

[3] This rule will be better explained and demonstrated in Sec. 6.2.

based reasoning can be used in the application of one of the transformation rules used within the manager.

6.2 Reasoning about Program Transformation Rule Correctness with Orc

We assume the availability of behavioural skeletons modelling the more common patterns of stream parallel computations, namely pipeline and task farm computations (i.e. computations organised in stages, and embarrassingly parallel computations over streams of input tasks). We also assume the availability of a skeleton modelling sequential composition of other skeletons onto the same processing resources (aka "in place" pipeline, henceforth named *comp*).

An application parallel program will thus be structured as a hierarchical tree of skeletons with pipeline, farm and comp skeletons in the nodes of the tree, and sequential components in the leaves providing the sequential code to be computed in the lowest level pipeline stages or task farm workers. Here we will assume that the structure of the parallel application, in terms of the skeleton used, can be represented with terms derived using the following grammar:

$$Sk ::= farm(Sk) \mid pipeline(Sk, Sk) \mid comp(Sk, Sk) \mid seq(f)$$

where **seq** models a sequential component implementing some function f[4]. The task farm and pipeline skeletons can be modelled in Orc as follows:

$$\begin{array}{lcl}
\mathsf{pipeline}(A, B, ch_{in}, ch_{out}) & = & \mathsf{stage}(A, ch_{in}, ch_{new}) \;\mid\; \mathsf{stage}(B, ch_{new}, ch_{out}) \\
\mathsf{farm}(W, nw, ch_{in}, ch_{out}) & = & \mid i = 1, nw \;:\; \mathsf{stage}_i(W, ch_{in}, ch_{out}) \\
\mathsf{seq}(A, ch_{in}, ch_{out}) & = & \mathsf{stage}_i(A, ch_{in}, ch_{out}) \\
\mathsf{comp}(A, B, ch_{in}, ch_{out}) & = & \mathsf{cBody}(A, B, ch_{in}, ch_{out}) \gg \\
 & & \mathsf{comp}(A, B, ch_{in}, ch_{out}) \\
\mathsf{cBody}(A, B, ch_{in}, ch_{out}) & = & ch_{in}.get() > task > A(task) > y > \\
 & & B(y) > result > ch_{out}.put(result) \\
\mathsf{stage}(A, ch_{in}, ch_{out}) & = & \mathsf{body}(A, ch_{in}, ch_{out}) \gg \mathsf{stage}(A, ch_{in}, ch_{out}) \\
\mathsf{body}(A, ch_{in}, ch_{out}) & = & ch_{in}.get() > task > A(task) > \\
 & & result > ch_{out}.put(result)
\end{array}$$

In the algorithmic skeleton framework it has been demonstrated that suitable rewriting can be performed at the skeleton tree level to obtain differently performing applications.

For example, pipeline computations with sequential stages can be collapsed to sequential computations to provide higher grain stages/workers and therefore to improve efficiency of the parallel computation:

$$pipeline(seq(f), seq(g)) \equiv comp(seq(f); seq(g))$$

[4] That is represents the skeleton wrapping of sequential code modelling a function (i.e. code with no side effects).

This result can be easily demonstrated using the Orc modelling of the skeletons presented above, and we will use this example to illustrate the Orc-based semi-formal reasoning that underpins our methodology.

The approach followed to demonstrate the equivalence above is the same as that used to derive the new version of the `muskel` manager in Sec. 5: we generate traces relative to the execution of Orc code, we look for matching *put* and *get* pairs, and we try to collapse traces using rule 1 of Sec. 5.

Applying this rule to our sample equation gives the following transformation:

$$
\begin{aligned}
&\mathsf{pipe}(A, B, c_1, c_3) = \\
&\qquad \mathsf{stage}(A, c_1, c_2) \mid \mathsf{stage}(B, c_2, c_3) \\
=\ &\qquad \mathsf{body}(A, c_1, c_2) \gg \mathsf{stage}(A, c_1, c_2) \mid \mathsf{body}(B, c_2, c_3) \gg \mathsf{stage}(B, c_2, c_3) \\
=\ &\qquad c_1.get() > t > A(t) > y > \mathbf{c_2.put(y)} \gg \mathsf{stage}(A, c_1, c_2) \mid \\
&\qquad\quad \mathbf{c_2.get()} > t > B(t) > y > c_3.put(y) \gg \mathsf{stage}(B, c_2, c_3) \\
\equiv\ &\qquad \mathsf{stage}(A, c_1, c_2) \mid \\
&\qquad\quad c_1.get() > t > A(t) > y > B(y) > z > c_3.put(z) \gg \mathsf{stage}(B, c_2, c_3) \\
=\ &\qquad \mathsf{stage}(A, c_1, c_2) \mid \\
&\qquad \mathsf{comp}(A, B, c_1, c_3) \gg \mathsf{stage}(B, c_2, c_3)
\end{aligned}
$$

and unfolding another iteration we get:

$$
\begin{aligned}
&= c_1.get() > t > A(t) > y > \mathbf{c_2.put(y)} \gg \mathsf{stage}(A, c_1, c_2) \mid \\
&\quad\ \mathsf{comp}(A, B, c_1, c_3) \gg \mathbf{c_2.get()} > t > B(t) > y > c_3.put(y) \gg \mathsf{stage}(B, c_2, c_3) \\
&\equiv \mathsf{stage}(A, c_1, c_2) \mid \mathsf{comp}(A, B, c_1, c_3) \gg \\
&\quad\ c_1.get() > t > A(t) > y > B(y) > z > c_3.put(z) \gg \mathsf{stage}(B, c_2, c_3) \\
&= \mathsf{stage}(A, c_1, c_2) \mid \mathsf{comp}(A, B, c_1, c_3) \gg \mathsf{comp}(A, B, c_1, c_3) \gg \mathsf{stage}(B, c_2, c_3)
\end{aligned}
$$

It is clear that $\mathsf{pipe}(A, B, c_1, c_3)$ unfolds to an iterated sequence of $\mathsf{comp}(A, B, c_1, c_3)$ when rule 1 is applied. The parallelism degree of the original program schema ($\mathsf{pipe}(A, B, c_1, c_3)$) is clearly higher than that of the derived schema ($\mathsf{comp}(A, B, c_1, c_3)$). The original schema allows A and B to be computed in parallel on two consecutive tasks appearing on the pipeline input stream. The derived schema allows only computation of one item at a time, but this computation has clearly a higher computation grain[5] and therefore is more suitable for use in conditions where communication overheads are not negligible. In other words, the two schemas can be considered functionally equivalent but they differ non-functionally in that they offer different grains of computation and thus are suitable for differing execution platforms.

Although the result emerging here from transformation of the Orc model is well-known (pipeline stage collapsing to coarsen granularity) the intent here is to illustrate the way in which we use Orc descriptions supported by semi-formal reasoning to investigate design alternatives for non-functional properties. In the

[5] Ratio between the time spent to compute and the time spent to communicate, i.e. the time spent to receive the input task and to deliver the result.

case of the earlier `muskel` example, no such well-known pattern underpinned the design, but reasoning at a similar level allowed redesign to achieve the desired non-functional property—bottleneck removal.

7 Extending Orc with Metadata

In the previous sections we showed how an Orc based framework can be used to described parallel/distributed programs, to analyze their features and possibly to compare different versions of the same parallel/distributed applications with respect to some well defined features (e.g. number of actual parallel activities, kind of synchronizations involved, etc.).

The next step in the methodology is aimed at extending the amount and the kind of information within the Orc based framework, in such a way that further applications of the methodology presented so far can be investigated.

The kind of enrichment of the Orc framework we consider is *adding metadata* to the Orc expressions and terms used to model the parallel application [15]. By metadata we mean any data associated with Orc terms and expressions to represent non-functional concerns of the computation. Metadata are therefore *annotations* associated with Orc terms.

We will demonstrate how metadata can be used by considering a simple case: metadata representing *locations* of the computation where the associated Orc terms are actually computed. Other typical kinds of metadata modelling information on the non-functional concerns include those related to security (e.g. whether a given computation described by an Orc term has to be considered confidential or not), to performance (e.g. actual and predicted performance values relative to computations performed by the Orc term/expression) or to fault tolerance (e.g. MTBF of a node). Using *location* metadata we will eventually be able to evaluate the best implementation among a set of functionally equivalent implementations differing only with respect to their non-functional features.

7.1 Introducing *location* Metadata

According to our methodology, metadata is associated to Orc terms in a formal way. We assume that each Orc expression has one or more metadata associated. We also assume that metadata are represented by using names (functors) and parameters (parameters of the functors). As an example, the term $location(E, a)$ represents the fact that $location(a)$ is associated with the Orc expression E.

Location metadata can be formally associated to complex Orc expressions in a completely formal way. For example, consider Orc expressions using the farm and pipeline skeletons presented in Sec. 6. Location metadata can be associated as follows:

- explicit association of user supplied metadata with expressions/terms in the Orc code;
- a rule rewrite method is defined to derive location metadata from the user supplied metadata in such a way that location information is propagated along the entire skeleton tree.

Several policies can be defined to propagate *location* metadata along the skeleton tree. We consider, at the extremes:

conservative placement policy the location of the root skeleton nodes are propagated unchanged to *all* the immediate descendant nodes, unless differently specified by the user/programmer. The process is applied recursively.

speculative placement policy independent of the location of the root node, a fresh location is assigned to each of the immediate descendant nodes, unless differently specified by the user/programmer. The process is applied recursively.

What usually will happen is that the user supplies location metadata for a few, notable expressions, and then the other metadata location can be derived with one of the available policies, possibly the one identified by appropriate metadata provided by the user/programmer. For example, consider the code:

$$prog(f, g, h) \equiv pipeline(seq(f), pipeline(seq(g), seq(h))$$

In our example, the programmer may be interested in expressing the maximum parallelism degree possible, and to keep the root of the tree on his own workstation. Therefore the program sketched above can be *user* annotated as follows: `location(prog,my_workstation), locPropagPolicy(speculative)`. This in turn, will lead to the following annotation of the skeleton tree:

$$\begin{array}{l}\langle\ location(prog, my\_workstation), location(seq(f), fresh\_loc()),\\ \ \ location(pipe(seq(f), pipe(seq(g), seq(h))), fresh\_loc()),\\ \ \ location(pipe(seq(g), seq(h)), fresh\_loc()),\\ \ \ location(seq(g), fresh\_loc()), location(seq(h), fresh\_loc())\ \ \ \ \ \ \rangle\end{array}$$

where the $fresh\_loc()$ function will query a resource manager and return the name of a fresh location.

7.2 Exploiting *location* Metadata

The annotation of a skeleton tree with location metadata can be used for different purposes. First (and obviously) it can be used to drive the deployment of the skeleton program on the distributed architecture at hand (the one represented by the resource manager answering the $fresh\_loc()$ calls. Then, it can be used to analyse those non-functional concerns that depend on (relative) location of computations: communication cost analysis, for example.

If we wish to evaluate the communication cost of our sample computation, we can keep expanding the relevant Orc terms and adding/deriving location metadata in such a way that we eventually get the locations of the sites involved in sends and receives. In turn, this information can be used to derive the cost of all the communications involved, assuming we know some constant T_{lc} and T_{rc} for communications having partners on the same node (T_{lc} *l*ocal

*c*ommunication) and those having the involved partners on different nodes (T_{rc} *r*emote *c*ommunication), respectively[6].

Traces may also be considered, associated to *location* metadata. In this case, the cost derived using metadata represents the overall amount of time spent communicating in the parallel/distributed application generating the trace.

These results, however, are not so interesting of themselves. The ability to take a program model and come up with a figure stating that the communication cost is $k \times T_{lc} + h \times T_{rc}$ is not so meaningful, independent of the ks and hs involved.

A much more interesting result stems from the ability to compare two *alternative* implementations. Let us assume that the parallel/distributed computation at hand can be implemented with two different algorithms/applications, modelled by Orc terms $OrcAppl_a$ and $OrcAppl_b$

In this case, we can proceed with the same user supplied initial metadata and location propagation policies and evaluate the final *ground* location labelling of our program (or, better, of the corresponding traces). Once this is done, we can compute the communication costs in terms of T_{lc} and T_{rc}. This time, however, by getting the two resulting terms giving the communication costs of traces relative to the same computation in $OrcAppl_a$ and $OrcAppl_b$, we can *compare* them and therefore determine which is the better of the computations with respect to communication costs.

More formally, this example of exploitation of Orc associated metadata can be expressed by:

- a grammar of terms over Orc expressions and metadata values is defined. For example:

$$\begin{aligned} E &::= \ldots Orc\ expressions \ldots \\ LocationMetadata &::= location(E, M) \mid locPropPolicy(M) \\ M &::= fresh\_loc() \mid loc(\langle literal \rangle) \mid \ldots \end{aligned}$$

  The grammar is used to denote all the "admissible" metadata for our Orc code.
- a set of rewriting rules are defined that provide a rewriting system propagating metadata along the Orc expressions modelling the computations: As an example, the following rule will belong to the set, denoting propagation of location in case of a conservative policy within a pipeline program:

$$\frac{location(pipe(A, B), L)}{location(A, L), location(B, L)} \quad \textsc{Cons.1}$$

- an abstract interpreter that computes the Orc expressions with respect to the associated metadata only and exploiting the rewriting rule set mentioned above.

[6] More realistically, we may consider functions of the sizes d of transmitted data $T_{lc}(d) = d/memory\_bandwidth$ and $T_{rc}(d) = latency + d/bandwidth$, with the same kind of results.

7.3 Exploiting *location* Metadata within Autonomic Managers

In previous sections, we have shown how *location* metadata can be used to evaluate which is the best implementation—with respect to a particular aspect, e.g. communication cost—among a set of equivalent, alternative implementations.

Such a result can be exploited in the manager described in Sec. 6.1. In particular, the result can drive choices made during the $analyse(s, m) > (b, p, v)$ phase, i.e. when analyzing a particular skeleton implementation s and the corresponding monitored behaviour m to determine whether some corrective action can be planned (b:boolean), which is the relative actuation plan p and, if necessary, which violation v has to be reported to the upper level manager. If alternative, feasible plans p' and p'' exist the result of $analyze(s, m)$ will be (b, p^x, v) with $p^x \in \{p', p''\}$ being the plan that in the subsequent $adapt(s, p) > s1$ phase will generate the improved new skeleton configuration, $s1$.

8 Related Work

In this work we concentrated on various issues relating to autonomic management of non-functional features in parallel/distributed computations. Although there is extensive work demonstrating how various aspects of parallel and distributed programming can be modelled using formal tools, there is much less work on exploitation of *semi*-formal techniques to support reasoning about non-functional concerns in parallel and distributed programs. We mention here a few research areas where reasoning schemas similar to that discussed in this work can be adopted.

The Service Component Architecture (SCA) [16] focuses on policies and implementation aspects of services but does not natively support dynamic reconfiguration of service assembly. However, the model can be extended to support dynamic reconfiguration. For example, the *Spatio-Temporal sKeleton Model* (STKM) [17,18], which can be defined in term of SCA, supports model reconfiguration by way of behavioural skeletons [4]. The STKM does not provide any specific methodology to reason about functionally equivalent assemblies or workflows of components.

An alternative approach is based on UML models. The work of [19] proposes the use of modes to address dynamic reconfiguration of service-oriented architectures and extends the UML to visualise such reconfiguration. The UML extension sticks to the mode terminology and does not include a visualisation of the transformation rules. The OMG is also working to standardize a UML profile and metamodel for services (UPMS) [20]. The current version does not support reconfigurations. Those approaches also propose a non-formalised approach (i.e. neither formal nor semi-formal). The only exception is the UML extension for service-oriented architectures that can be found in [21], which proposes refinement issues based on architectural styles formalised by graph transformation systems.

Architectural styles are the basis of the *Architectural Design Rewriting* (ADR) approach, which has been inspired mainly by graph-based approaches [22,23].

The use of graphs and graph transformations to model architectural styles has been proposed by several authors (e.g [24]) who based their approaches on the concept of shapes in programming languages. ADR shares also concepts with approaches based on process calculi with reconfigurable components (e.g. [25]). iADR is also related to approaches that deal with reconfigurations in software architectures defined by an ADL [26], and by graph transformation such as the *Synchronised Hyperedge Replacement* (SHR)[27]. Models in this family typically support the fully-fledged formal reasoning on assembly reconfiguration and equivalence; they have been proved effective in proving the correctness of single adaptations and simple sequences of them [28]. However, to be checked, these abstract models should be mapped down into concrete models describing a specific implementation enormously increasing the complexity of the description. As a matter of a fact, this complexity often prevents the designer from reasoning about the expected long-term evolution of the distributed system.

Model Driven Architecture [29] concepts look close to the idea of using Orc as the modelling language for actual application code. In this perspective, Orc can be intended as the PIM (platform independent model) to be used to derive, with some kind of automatic or semi-automatic tools, the PSM (platform specific model) and eventually an actual implementation.

Aspect Oriented Programming techniques have been taken into account in different frameworks to model and handle non-functional concerns (see, for example, [30,31]). We believe this approach is complementary to the behavioural skeleton idea adopted here. However, AOP techniques and mechanisms could probably be exploited in the autonomic manager implementation to further relieve system programmers of non-functional concern handling details, providing a finer grain of "separation of concerns" within the non-functional ones.

Finally, we chose Orc as our modelling language for two reasons. First, our interest was in *management* of functionality, and Orc's emphasis on orchestration of computations made it thus a perfect fit; second we wished to have a very compact language that allowed us to develop constructive representations of different designs, and reason about them at a high (but not too high) level of abstraction: this caused us to steer away from, on the one hand, very abstract notations such as π-calculus [32] which support a more abstract level of reasoning than we desired; and, on the other hand, parallel programming languages such as Erlang [33] and Oz [34] which are suitable for implementation rather than design.

9 Conclusions

In this paper we have discussed the challenge of non-functional concern management in parallel/distributed systems and emphasized the desirability of separation of functional and non-functional concerns. We have presented behavioural skeletons as a means of extending component-based parallel/distributed skeletons with autonomic managers taking care of non-functional concerns. The suitability and use of Orc (and its extension with metadata) to specify such autonomic management has then been argued and, to this end, we have emphasized a semi-formal

style in which the specifications are treated as designs and an informal style of reasoning, drawing heavily upon insight and experience, is used to compare non-functional properties of alternative designs. While the experience has suggested the efficacy of the approach, much more experimentation is needed to determine the extent to which aspects of the methodology such as, for example, the rule identified in section 5 are transferable across different applications within a domain and even across domains of application. Ideally, rules of thumb of this sort would be identified at a level consistent with the approach, that is, an approach in which one gains benefit from the curt, well-founded definitions without resort to onerous formal reasoning.

References

1. Parnas, D.L.: On the design and development of program families. IEEE Trans. on Software Engineering SE-2(1), 1–9 (1976)
2. Boehm, B.W.: A spiral model of software development and enhancement. Computer 21(5), 61–72 (1988)
3. Aldinucci, M., Danelutto, M., Kilpatrick, P.: Management in distributed systems: A semi-formal approach. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 651–661. Springer, Heidelberg (2007)
4. Aldinucci, M., Campa, S., Danelutto, M., Vanneschi, M., Dazzi, P., Laforenza, D., Tonellotto, N., Kilpatrick, P.: Behavioural skeletons in GCM: autonomic management of grid components. In: Baz, D.E., Bourgeois, J., Spies, F. (eds.) Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing, Toulouse, France, pp. 54–63. IEEE, Los Alamitos (2008)
5. Aldinucci, M., Danelutto, M., Kilpatrick, P., Dazzi, P.: From Orc models to distributed grid Java code. In: Gorlatch, S., Fragopoulou, P., Priol, T. (eds.) Grid Computing: Achievements and Prospects. CoreGRID, pp. 13–24. Springer, Heidelberg (2008)
6. CoreGRID NoE deliverable series, Institute on Programming Model: Deliverable D.PM.04 – Basic Features of the Grid Component Model (assessed) (2007), `http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf`
7. Cole, M.: Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. Parallel Computing 30(3), 389–406 (2004)
8. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
9. Danelutto, M.: QoS in parallel programming through application managers. In: Proc. of Intl. Euromicro PDP: Parallel Distributed and network-based Processing, Lugano, Switzerland, pp. 282–289. IEEE, Los Alamitos (2005)
10. Aldinucci, M., Danelutto, M.: Algorithmic skeletons meeting grids. Parallel Computing 32(7), 449–462 (2006)
11. Aldinucci, M., Danelutto, M., Kilpatrick, P.: Co-design of distributed systems using skeletons and autonomic management abstractions. In: César, E., et al. (eds.) Euro-Par 2008 Workshops. LNCS, vol. 5415, pp. 403–414. Springer, Heidelberg (2009)
12. Sensoria Project: Software Engineering for Service-Oriented Overlay Computers (2008), `http://sensoria.fast.de/`
13. 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

14. Aldinucci, M., Danelutto, M., Kilpatrick, P.: Autonomic management of non-functional concerns in distributed and parallel application programming. In: Proc. of Intl. Parallel & Distributed Processing Symposium (IPDPS), Rome, Italy. IEEE, Los Alamitos (2009)
15. Aldinucci, M., Danelutto, M., Kilpatrick, P.: Adding metadata to orc to support reasoning about grid programming. In: Priol, T., Vanneschi, M. (eds.) Towards Next Generation Grids (Proc. of the CoreGRID Symposium 2007). CoreGRID, Rennes, France, pp. 205–214. Springer, Heidelberg (2007)
16. IBM: Service Component Architecture (SCA), `http://www.ibm.com/developerworks/library/specification/ws-sca/` (last accessed 2008)
17. Aldinucci, M., Danelutto, M., Bouziane, H.L., Pérez, C.: Towards software component assembly language enhanced with workflows and skeletons. In: Proc. of the ACM SIGPLAN Component-Based High Performance Computing (CBHPC), pp. 1–11. ACM, New York (2008)
18. Bouziane, H.L., Pérez, C., Priol, T.: A software component model with spatial and temporal compositions for grid infrastructures. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 698–708. Springer, Heidelberg (2008)
19. Foster, H., Uchitel, S., Kramer, J., Magee, J.: Leveraging modes and UML2 for service brokering specifications. In: CEUR 2008. LNCS, vol. 389, pp. 76–90. Springer, Heidelberg (2008)
20. Object Management Group (OMG): UML Profile and Metamodel for Services (2008)
21. Baresi, L., Heckel, R., Thöne, S., Varró, D.: Style-based modeling and refinement of service-oriented architectures. SOSYM 5(2), 187–207 (2006)
22. Hirsch, D., Montanari, U.: Shaped hierarchical architectural design. In: ENTCS, vol. 109 (2004)
23. Bruni, R., Bucchiarone, A., Gnesi, S., Hirsch, D., Lluch Lafuente, A.: Graph-based design and analysis of dynamic software architectures. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 37–56. Springer, Heidelberg (2008)
24. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, New Jersey (1996)
25. Aguirre, N., Maibaum, T.S.E.: Hierarchical temporal specifications of dynamically reconfigurable component based systems. In: ENTCS, vol. 108, pp. 69–81 (2004)
26. Bruni, R., Lluch-Lafuente, A., Montanari, U., Tuosto, E.: Architectural design rewriting as an architecture description language (position paper). Technical Report MSR-TR-2008-61, Microsoft Research Cambridge, Proceedings of R2D2, Workshop on the Rise and Rise of Declarative Datacentre (2008)
27. Ferrari, G.-L., Hirsch, D., Lanese, I., Montanari, U., Tuosto, E.: Synchronised hyperedge replacement as a model for service oriented computing. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 22–43. Springer, Heidelberg (2006)
28. Aldinucci, M., Tuosto, E.: Towards a formal semantics for autonomic components. In: Priol, T., Vanneschi, M. (eds.) From Grids To Service and Pervasive Computing (Proc. of the CoreGRID Symposium 2008). CoreGRID, Las Palmas, Spain, pp. 31–45. Springer, Heidelberg (2008)
29. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture–Practice and Promise. Addison-Wesley Professional, Reading (2003)

30. Jingjun, Z., Furong, L., Yang, Z., Liguo, W.: Non-functional attributes modeling in software architecture. In: SNPD 2007: Proceedings of the Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, Washington, DC, USA, pp. 149–153. IEEE Computer Society, Los Alamitos (2007)
31. Lohmann, D., Spinczyk, O., Schröder-Preikschat, W.: On the configuration of non-functional properties in operating system product lines. In: Proceedings of the 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS 2005), Chicago, IL, USA, Northeastern University, Boston (NU-CCIS-05-03), 19–25 (2005)
32. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press, Cambridge (1999)
33. Cesarini, F., Thompson, S.J.: Erlang Programming, A Concurrent Approach to Software Development. O'Reilly, Sebastopol (2009)
34. Van Roy, P. (ed.): MOZ 2004. LNCS, vol. 3389. Springer, Heidelberg (2005)