Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 114 (2005) 87–102
www.elsevier.com/locate/entcs

An Adaptive High-Performance Service Architecture

Jesper Andersson[1] Morgan Ericsson[2] Welf Löwe[3]

Department of Computer Science, School of Mathematics and Systems Engineering, Växjö University, S-351 95 Växjö, Sweden

Abstract

This paper introduces an approach to dynamic software composition in the context of scientific computing where high demands performance seem to prevent such flexible solutions. In our concrete however, dynamic software composition is rather a way to high-performance than an obstacle to it. We achieve this by combining dynamic architectures and task graph scheduling.

Keywords: Dynamic software composition, dynamic architecture, task graph scheduling

1 Introduction

In the world of scientific computing a common problem is the mapping of one or more computations onto a grid of interconnected machines. The efficiency of the application or even the feasibility of running the computation is very much dependant on this mapping. Unfortunately, finding an optimal mapping is an NP-hard problem.

Another problem is that this mapping is most often done off-line and once, which means that the mapping will not be redone if the execution environment changes. This could happen, for instance, due to the application requiring increased precision, eventually leading to changes in the whole system architecture.

[1] Email: jesan@msi.vxu.se
[2] Email: mogge@msi.vxu.se
[3] Email: wlo@msi.vxu.se

doi:10.1016/j.entcs.2004.02.068

This paper presents a dynamic service infrastructure for high-performance applications solving problems like those discussed above. It is originally developed for the LOIS space antenna IT-infrastructure. Our solution is influenced by a number of different fields within computer science, for instance optimization of parallel programs, dynamic software architectures and software composition.

The paper is organized as follows. Section 2 presents the original application scenario, the LOIS and by extension the LOFAR space antenna infrastructure. Section 3 outlines the requirements posed by this and similar application. Section 4 describes the component model and the static composition of components into systems, while section 5 presents the dynamic reconfiguration. Section 6 presents a generalization of the results into a dynamic service infrastructure. Finally, section 7 concludes the paper and describes future work.

2 Concrete Application

This section describes the concrete application environment that our research is embedded in. It justifies our assumptions and constraints and serves as an example.

LOFAR (LOw Frequency ARray), [4] is a new generation, multi-purpose radio infrastructure aiming at multi-disciplinary research of astronomers, cosmologists, space and atmospheric physicists, climatologists, cosmic particle physicists, radio scientists, wireless communication developers and IT researchers. It consists of geographically distributed digital receptor units connected to computing facilities with a high-speed network. Units are distributed over distances of 400 km; any unit will produce data at a rate of 2 Gbits/s, resulting in a total system data rate of 25 Tbits/s. The Swedish initiative *LOIS (LOFAR Outrigger In Scandinavia)* [5] aims, among others, at extending and enhancing the IT infrastructure capabilities of LOFAR.

The data collected will be processed by scientists performing a variety of (virtual) experiments. In order to perform an experiment an application must be deployed to the infrastructure. These applications can be deployed and/or removed dynamically. Several experiments can be performed at any given time. Experiments can process data on-line or off-line, depending on the requirements of the experiment.

We understand the biasing between storage and processing (and alternative decisions within the two) as a global, dynamic optimization problem. An

[4] http://www.lofar.org
[5] http://www.lois-space.org

optimization engine that decides about changing the system configuration dynamically needs to attend to user (researcher) triggered reconfigurations as well as to changes triggered by applications monitoring the data. A runtime environment must allow to perform the actual reconfigurations in a robust way. We propose a dynamic service architecture to solve these problems.

3 General Requirements

This section defines some basic notions and roles, describes usage scenarios and, thereby, establishes requirements on a dynamic service infrastructure running on top of the IT infrastructure, i.e. the hardware.

The kind of basic *IT infrastructures* we are discussing consist of

- a number of sensors generating a stream of *input values*,
- a number of computation processors. These are referred to as *nodes*. Some nodes are connected to sensors receiving the input values. The latter are referred to as *input nodes*.
- an interconnection network allowing for communication between the nodes.

On this IT infrastructure, we execute the "experiments", i.e. applications processing data from the sensors. These *applications* are data parallel programs. Their input are either sensor input values or the output of other applications. If the input of an application a is the output of an application a', a is called *data-dependent* on a'. It is denoted by $a' \rightarrow a$. Applications are stateless, data driven, functions.

In addition to the above applications there is a database application and attached daemons for storing and retrieving data streams. This application is needed to allow applications to process off-line data.

The *systems* to be composed are a set of applications and their connections according to the data dependencies between them. The *configuration* of such a system is defined by:

- its set of applications,
- the data dependencies, and
- quality of service parameters for the different applications, for instance processing time and ratio.

The configuration may change over time. These changes are triggered by different sources.

User triggered: The *user* in our scenario is in charge of a certain experiment, i.e. controls a certain set of applications. User interactions are adding and

removing applications. Adding new applications requires (i) connecting its input to the input values or to the output of running applications and, (ii) setting the quality of service parameters. In order to remvove an application from the system, the identity of the application in question must be specified.

A typical scenario in this category is that a user adds a new experiment and necessary intermediate computations to the system and removes it after gaining the results.

Application triggered: Some applications are detectors, recognizing certain patterns in the processed data that require reconfiguration, for instance adding or removing applications and/or changing quality of service parameters.

A typical scenario is the detection of an interesting sensor activity requiring an increased sampling rate. This leads to changed quality of service requirements and might lead to the situation where some on-line applications must be postponed and computed off-line. Their required data is then stored in the database, and their input is connected to the database daemon.

System triggered: The complexity of applications might be input data dependent. Certain input might lead to load peaks in these applications. In order to guarantee the required quality of service in some applications certain others are removed or postponed to off-line computations.

A typical scenario is load balancing on sub-application level, i.e. redistributing some tasks in the data-parallel applications.

These changes are controlled by a dynamic service infrastructure, which is described in Sections 5 and 6.

4 Systems and their Static Composition

This section outlines the programming and composition model. More specifically section 4.1 defines a programming and execution model for the data-parallel, and section 4.2 describes their composition to systems.

4.1 Applications – Programming and Execution Model

For components we assume a High Performance Fortran (HPF)-like programming model, with data parallel synchronous program but without any data distribution. For simplicity, we further assume that the programs operate on a single composite data structure which is an array, a. The size of an input a, denoted by $|a|$, is the length of the input array a.

Array a is either the array of input values or the output of another component. Note that in our scenarios, the number of input values is fixed. We may assume that the output size of an application is a function of the input size. By induction, it follows that the input size is fixed for all applications in our systems.

We can model the execution of an application component on an input x by a family of task-graphs $G_x = (V_x, E_x, \tau_x)$. The tasks $v \in V_x$ model local computations without access to the shared memory, $\tau(v)$ is the execution time of task v on the target machine, and there is a directed edge from v to w iff v writes a value into the shared memory that is read later by task w. Therefore, task-graphs are always acyclic. G_x does not always depend on the actual input x. In many cases of practical relevance it only depends on the problem size n. We call these program *oblivious* and denote its task graphs by G_n. We write G instead of G_n if n is arbitrary but fixed. The *height* of a task v, denoted by $h(v)$, is the length of the longest path from a task with in-degree 0 to v.

Scientific applications can automatically be compiled to such a family of task-graphs. Many of them are oblivious or iterations over oblivious loop bodies, e.g. Matrix Multiplications, Fast Fourier Transformations (FFT), CG-Methods, Finite-Element-Methods etc. [13].

Machines are modelled by LogP [4]: in addition to the computation costs τ, it models communication costs with parameters Latency, $o$verhead, and $g$ap (the inverse of the bandwidth per processor). In addition to L, o, and g, the parameter P describes the number of processors. Moreover, there is a capacity constraint: at most $\lceil L/g \rceil$ messages are in transmission in the network from any processor to any other processor at any time. A send operation that exceeds this constraint stalls.

A LogP-schedule is a schedule that obeys the precedence constraints given by the task-graph and the constraints imposed by the LogP-machine, i.e., sending and receiving a message takes time o, the time between two consecutive send or receive operations must be at least g, the time between the end of a send task and the beginning of the corresponding receive task must be a least time L, and the capacity constraint must be obeyed. For simplicity, we only consider LogP-schedules that use all processors and where no processor sends a message to itself. A *LogP-schedule* is a set of sequences of computations, send, and receive operations and their starting times corresponding to the tasks and edges of the task-graph. For each task, its predecessors must be computed either on the same processor or their outputs must be received from other processors. The schedules must guarantee the following constraints: (i) sending and receiving a message of size k takes time $o(k)$, (ii) between two sends or two receives on one processor, there must be at least time $g(k)$, (iii)

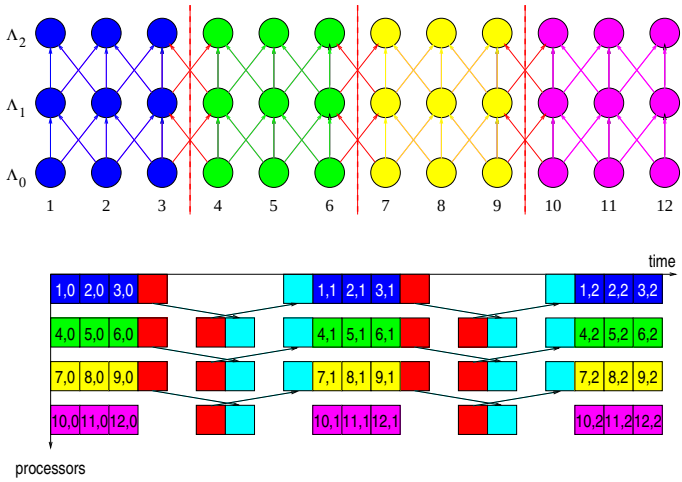

Figure 1. Partitioning the task graph according to block-wise data distribution (left) and the corresponding LogP-schedule (right) with parameters $L = 2$, $o = 1$, $g = 2$.

a receive must correspond to a send at least $L(k) + o(k)$ time units earlier in order to avoid waiting times, (iv) computing a task v takes time $\tau(v)$, and (v) a correct LogP-schedule of a task-graph G must compute all tasks at least once. $TIME(s)$ denotes the execution time of schedule s, i.e., the time when the last task finishes. Figure 1 shows a task graph, sketches a scheduling algorithms according to block-wise distribution of the underlying data array and gives the resulting schedule.

The computation of a optimal LogP-schedule is known to be NP-hard. However, good approximations and heuristics, c.f. our own contributions in task scheduling, e.g. [9,8,7], guarantee a small constant factor in delay. In practice, the results are even closer to the optimum. The techniques can be applied to the class of scientific problems and parallel machines required in the proposed project's context [14]. Moreover, an upper time bound for $TIME(s)$, the execution time of a schedule s, the can be determined statically.

4.2 Static Composition of Applications to Systems

So far, we defined the components as data-parallel applications, translated to task graphs and scheduled to the IT infrastructure. For each component, we can determine an upper time bound for its execution. Each component implements a function mapping an input array a_i to an output array a_o.

To this end, composition of components can be done by defining a program using these components and assigning the output array of one component to the input of another. This system is also a data-parallel program and can

therefore be compiled and scheduled just like the individual components.

Adding/removing a component requires a complete new translation of the system to a new task graph and a rescheduling of the new task-graph. This is no problem since adding/removing a component corresponds to setting up a new experiment or terminating an experiment. These activities can be planned and prepared off-line. After the complete schedule is computed for the new system computation can switch from the running to the new system.

Preparation time can be reduced by two means: (i) performing composition on task graph level instead of application level and (ii) using predefined schedules for the task graphs. Both will be discussed below.

Instead of composing data-parallel applications to a data-parallel system that is translated to task graphs and scheduled to the IT infrastructure, we bookkeep the task graphs of the individual applications and just compose these task graphs. Adding a new application e' is still specified by calling its corresponding function with the result of an existing application e. However, only the new application e' is translated into a new task graph. The input tasks of this task graph are connected with the output nodes of the task graph of e and then handed over to the scheduling unit. Inversely, removing e' leads to disconnecting the corresponding task graphs and deleting transitively depending tasks.

While reusing task graphs is straight forward, reusing schedules is not since a optimum schedule (or its approximation) does not necessarily keep the schedules for the different task graphs distinct. Instead, it might merge task of different task graphs into one process. Moreover, optimum schedules of individual task graphs (or their approximations) are, in general, not part of the optimal schedule for a composed system (or its approximation).

The problem discussed above is approached by modelling task graphs as *malleable tasks* and systems with *malleable task graphs*. A malleable task is a task that can be executed on $p = 1 \ldots P$ processors. Its execution time is described by a non-increasing function, τ, in the number of processors p actually used. For each task graph the schedules s_p can be pre-computed for $p = 1 \ldots P$ and $\tau(p) = TIME(s_p)$. A malleable task graph is recursively defined. It is a task graph over malleable tasks, i.e. nodes are ordinary task graphs or malleable task graphs and edges are the data-dependencies between them. Scheduling algorithms for malleable task graphs and their performances is a relatively new research area; first results are discussed in [12].

So far our components are modelled as malleable tasks. These tasks are composed to systems modelled by malleable task graphs. The systems are mapped to the IT infrastructure by reusing the pre-computed mapping of the individual malleable task graphs. It remains to show how the schedules

are started and/or replaced when adding/removing components and how the individual tasks exchange data. This is described in the next section.

5 Dynamic System Reconfiguration

In our dynamic service architecture, we distinguish the architecture of the *processing system* described in the previous section from the architecture of the *control system* managing the changes in the processing system. The architecture of the processing system is defined by a static composition of data-parallel applications.

Moreover, we distinguish between the set of *architectures*, A, of the processing system and the actual *implementations*, I, executed at runtime. There is a 1-to-1 mapping, M, between the conceptual processing architectures A, and the actual physically running systems $M : A \rightarrow I$. This mapping is established by the compilation of systems of data-parallel applications to malleable task graphs and the scheduling of these graphs to processing nodes, as described in the previous section. The mapping M transforms the processing system architecture into a physical process schedule and distribution.

In short the whole dynamic service architecture contains two architecture levels, described here and depicted in Figure 2.

The *conceptual level* includes the processing and control system architectures,
used for system and application (re-)configuration;

The *physical level* includes implementation architecture, described below.

The conceptual level is used to manage new system architectures, either advised by a user who adds/removes applications and/or quality of service requirements or as a reaction of predefined runtime events. Each new system architecture $a \in A$ triggers the computation of a corresponding implementation $i = M(a)$ including the computation of a new schedule that is distributed to the physical level. Inversely, the physical level affects the conceptual level by forwarding events initiating changes in the system architecture, i.e. reconfiguration.

5.1 Implementation Architecture

At run-time certain activities mainly concerned with creation, disposal, connection, and activation require support in a run-time system. These activities are coordinated in a specific coordinator component. Additionally, this *coordinator* component attaches probe connections to detector applications generating system events for reconfiguration. These events triggers, for instance

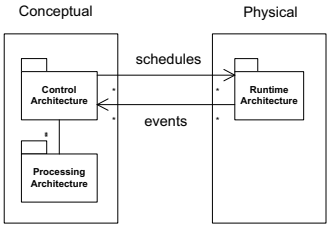

Figure 2. Our Dynamic Service Architecture

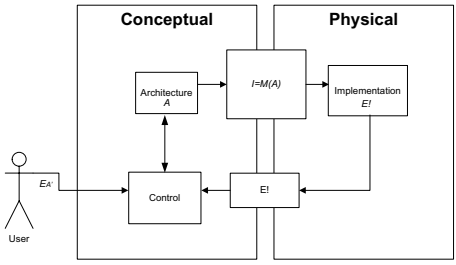

Figure 3. Reconfiguration and Mapping principle

reconfigurations of the implementation architecture. The run-time system receives new implementations, from the off-line conceptual architecture system, that should be deployed immediately to the IT-infrastructure or bookkept for event triggered reconfigurations (see discussion below).

5.2 Reconfiguration Process

The implementation architecture uses off-line processes generating implementations (including schedules). The fact that different events may occur in the system or external to the system requires that new architecture-implementation mappings are continuously generated and deployed.

In Figure 3 we depict this schematically. On the left-hand side we have the conceptual system architecture and a control architecture, while the right-hand-side represent the physical implementation. Both the events $E_{A'}$ from the user and the runtime events $E!$ trigger changes of the conceptual architecture and thereby changes of the physical architecture, as well. Events $E_{A'}$ are inputs to the conceptual architecture from the user, while the runtime events $E!$ are forwarded from the physical architecture.

These two event classes initiates the generation of new implementations using the translation and scheduling $M : A \rightarrow I$.

The problem with this approach is that the generation of new mappings is a time consumptive process. As discussed earlier, NP-hard optimization problems needs to be approximated. However, if an $e! \in E!$ occurs, the

reconfiguration must be finished in milliseconds. Call back the application scenario to see that: a radio antenna listens to sun eruptions with a moderate sample rate. If an application discovers that an eruption is about to start, the sample rate needs to be increased considerably. This leads to system reconfigurations making it necessary to take some applications off-line and just store the data instead.

We exploit the assumption that the expectation on the rate of such events $E!$ is rather low. The fundamental principle for the optimization of our dynamic reconfiguration is to employ a continuous implementation mapping with a *lookahead schema*. For each system architecture A, we pre-compute possible changes Δ w.r.t. possible system events $E!$. More specifically: given a current (baseline) architecture $a \in A$ and each possible system event $e! \in E!$, we compute the evolved architecture $a' = \Delta(e!, a)$. After having determined possible deltas, we have a set of *lookahead(1)* architectures. They are mapped in the same way as the base-line-architecture into *lookahead(1)*- implementations: $i' = M(a')$. Together with the current baseline implementation $i = M(a)$, these possible *lookahead(1)*-implementations are deployed. If this process is not interrupted by system events $E!$, we can react on events to come very shortly. Below we give a more detailed account for and formal definition of the two different mapping generation activities. The complexity of calculating the Δ is polynomial, since we do not differentiate between combinations of events and the same set of events occuring in a sequence, and since we use a lookahead of one.

5.2.1 User Triggered Reconfiguration

As described above and in Section 3, users trigger reconfigurations. This will be performed on the conceptual level. The system gets the new baseline architecture, simulates system events and stores the resulting *lookahead(1)* architectures. Both the baseline as well as the *lookahead(1)* architectures are translated and optimized. While the baseline implementation is deployed, the *lookahead(1)* implementations are stored in the coordinator of the runtime system.

(i) changes to the current baseline architecture a are conducted leading to a new baseline architecture a'

(ii) a new base-line implementation $i' = M(a')$ is generated,

(iii) the base-line implementation i' is handed over to the runtime coordinator for immediate deployment on the IT-infrastructure.

(iv) the *lookahead(1)* architectures $A'' = \{a'' \mid a'' = \Delta(e!, a'), e! \in E!\}$ are created.

(v) the *lookahead(1)* implementations $I'' = \{i'' \mid i'' = M(a''), a'' \in A''\}$ are created. Moreover, we bookkeep the mapping $M\Delta : E! \rightarrow I''$ between possible system events $E!$ and the *lookahead(1)* implementations I''. Both I'' and $M\Delta$ are handed over to the runtime coordinator for rapid deployment of evolved implementation in case of system event occurrence.

5.2.2 System Event Triggered Reconfiguration

The system event triggered generation is managed slightly different compared to off-line system configurations. We describe this activity more formally below.

(i) the new base-line implementation i' for the system event $e!$is selected as $i' = M\Delta(e!)$ and deployed immediately on the IT-infrastructure,

(ii) the run-time coordinator initiates an off-line schedule by forwarding $e!$ to the off-line coordinator

(iii) changes to the current baseline architecture a are conducted leading to a new baseline architecture a', reflecting the already deployed base-line implementation i',

(iv) as before,

(v) as before.

6 Generalization to Dynamic Service Infrastructures

Our dynamic service infrastructure described in the previous section can be abstracted as a general solution pattern to introduce dynamism to originally static composition scenarios. This section introduces such a general *Dynamic Service Infrastructure*. It does also present the important design considerations and decisions made during elaboration and design. In the previous section, key properties and requirements were introduced. A conclusive statement describing these would be *flexibility* and *quality of service*.

The architecture described below is designed with a *conceptual level* and a *physical level*. This adheres to the principles of model-driven architecture [6]. The conceptual architecture is used for off-line tasks like, experiment set-up and implementation mapping. The physical architecture used at run-time is a stripped architecture similar to the one discussed in Section 5.1.

Before we begin with the technical discussion on the architecture, we provide our understanding of architecture and its applications. The architecture of a system [10] is defined by its configuration of computational components and their connectors. As noted in previous works, e.g Bass et.al [3, p 32.], an architecture "inhibits or enables" quality properties in an application.

In the process of finding a suitable architecture candidate, these properties are used for evaluating different candidates.

Even if the design of an architecture result in a configuration that fulfills several of the quality requirements, there are still other qualities are "run-time discernible" [3, p. 79], hence require support at run-time, for instance performance and scalability. Several of the quality requirements can be achieved by "systematically controlling the inter-component communications" [5]. System wide properties in "component based" systems will "require dedicated support outside of the participating components" [11,2]. Over the years, the software architecture community has proposed a number of different "tools" to achieve realization of this class of system properties.

6.1 Architecture

The service architecture we propose is a middle-ware style system, i.e. a run-time system with an accompanying framework and tools. The middle-ware style provides provisioning a flexible processing environment, prepared for, quality-driven, flexible application architecture configuration management.

The architecture is a composite instantiation, influenced by the architectural patterns proposed in the authors previous work [1]. In order to achieve the manifested requirements, it is important to find a good structuring principle and defines a clear interface between different component types. The resulting architecture separates conceptual concerns from physical representation.

6.1.1 Conceptual architecture

Below we present the two aspects of the conceptual architecture, which we depict in Figure 4. We factor out control concerns into a separate package, thus distinguishing from control components. The processing architecture capture the core behavior of a specific experiment as a configuration of connected applications, while the Control architecture is concerned with architecture evolution.

Processing Architecture

The processing architecture contains three basic component types; *Application*, *Connector*, and *Configuration*. We depict these components and the relationships in figure 4. A *Application* is a container component where computations are performed. All interactions with a *Application* goes through a typed interface. A *Connector* captures component interactions. The typed end-points of a *Connector* connects to *Application* interface points. A *Config-*

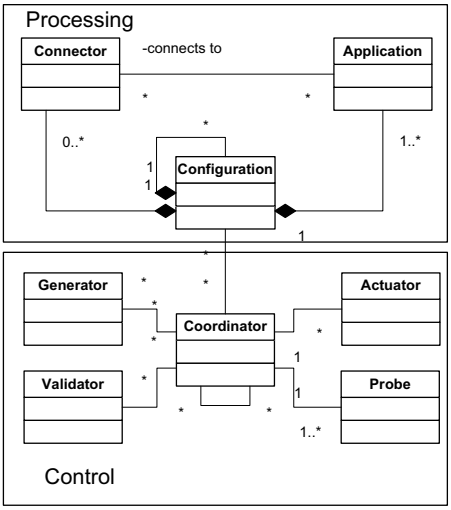

Figure 4. Conceptual Architecture

uration is a composition of *Application* and *Connector* components. Some engineering tasks require hierarchical configurations where a *Configuration* contains other configurations, thus we provide provisioning for these situations.

Control Architecture

The control architecture consist of five component types; *Probe*, *Actuator*, *Generator*, *Validator*, and *Coordinator*. The *Probe* and *Actuator* are the bridging point where control connects to processing. A *Probe* is a configurable monitoring process that monitor processing components and generate events that are communicated back to its Coordinator. The *Coordinator* component is the decision maker in the service architecture responsible for coordination and delegation of control tasks. A *Coordinator* is goal-driven and can be employed on different levels in the architecture. The top-most *Coordinator* component strives for fulfilling the quality of service requirements posed by the different applications, while other *Coordinator* instances are responsible for coordination of sub-architectures (applications). In this process they employ different *Generator* and *Validator* components, for instance for creating new application configurations or rule-sets for coordination control and monitoring. The meta-data is provided by the processing architecture elements described above. The *Coordinator* uses *Actuator* components to directly affect application configurations and/or application component instances..

6.1.2 Physical architecture

The physical architecture have two principle tasks assigned to it.

(i) manage dynamic reconfigurations

(ii) event management

In its most general case one could even consider to assign a responsibility for evolving a system, generating new configurations on the fly. In most computational applications we could not expect a 1-to-1-mapping between the conceptual architecture and a working implementation architecture and generating mappings (e.g schedules) is a costly activity. In our current setting this is not feasible, but we would not like to rule that out as impossible in the future.

The first task of the physical architecture is to manage dynamic reconfigurations. In order to perform a robust fine-grained deployment, the run-time coordinators need access to meta-information describing the current conceptual level and its mapped implementation. This information describe a *Configuration* of *Application* and *Connector* instances at different levels. The run-time system provide a modification language for the actual execution of a re-configuration of the implementation as well as a language for conceptual modifications. These includes primitives for the fundamental activities such as creation and connection.

The second assigned task is event management. Events generated externally (e.g a new schedule arrives) and internally (e.g increased sample rate) will be responded to proper actions taken. This is governed by a coordinator application. This coordinator is also responsible for feeding information back to the conceptual coordinator, for instance forwarding events that initiates a new lookahead schedule.

6.2 Validation

This control architecture provide provisioning for the requirement discussed in previous sections. Reconfiguration and evolution is supported by *Coordinator*, *Generator*, and *Validator* components. For external initiation, (i.e Constructive Architecture style [1]), *Probe* and *Actuator* components are exported and included in external management applications. Application triggered re-configurations are supported by the connecting *Probe* and *Actuator* components to *Application* and *Connector* entities in the processing architecture. For system level initiation, coordination is supported on different levels. *Coordinator* components present in different applications connect to *Coordinator* entities on the system level. The network of *Probe* and *Actuator* instances also work on this level. Making these entities available for both on-line and off-line reconfiguration creates a highly flexible environment.

7 Conclusions

The paper discusses systems of data-parallel applications, e.g. stemming from the field scientific computing. These applications usually have high requirements on performance. The requirements on flexibility used to be rather low, since they where exclusively executed on dedicated processors of parallel computers.

However, with the introduction of multiple applications sharing such a parallel computer resource, e.g. in GRID computing, the requirements for flexibility increased since applications where added and removed unpredictably. The performance requirements still remained high. Since the applications are independent of another, well researched load balancing techniques where the natural solution.

In our scenario, requirements for flexibility are even higher since applications are not only added / removed. Instead the system architecture might change depending on the results of some of the applications. Moreover, applications build on another and the performance requirements still remains high.

The main contribution of the paper is the introduction of a dynamic service infrastructure for data-parallel applications, cf. Section 5. We built on a static composition and optimization model of data-parallel programs, we introduced a system infrastructure allowing both high performance computing and dynamic change of the systems architecture. Key to our solution are precomputations of possible changes in the architecture based on run-time events. These evolved system architectures are translated, optimized in parallel to the execution of the actual system. On occurrence of a change triggered by any of those events, the new optimized system can be deployed without further preparations. Additionally, such an event triggers the translation and optimization of the new generation of systems the currently running will possibly evolve to.

Future work will implement the dynamic service architecture proposed in our own test bed from the LOIS project context. Here, we are are also concerned with practical questions like administration of adding/removing applications and prioritizing applications.

On the theoretical level, we are interested in extending our cost model towards the compilation and scheduling processes of the applications. Together with a modelling of the expectations of different system events, we might then be able to prioritize the creation of specific evolved systems including even the creation of systems for more than one evolution step in the future.

References

[1] J. Andersson. A classification of dynamic software architectures. Technical report, Department of Computer Science, Växjö universitet, 2003.

[2] Jesper Andersson. *Towards Reactive Software Architectures*. PhD thesis, 1999.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1997.

[4] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 93)*, pages 1–12, 1993. published in: SIGPLAN Notices (28) 7.

[5] R. E. Filman. Achieving ilities. In *Workshop on Compositional Software Architectures (Monterey, California)*, 1998.

[6] D.S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, 2003.

[7] W. Löwe, D. Trystram, and W. Zimmermann. On scheduling send-graphs and receive-graphs under the logp-model. *Information Processing Letters*, 2001.

[8] W. Löwe and W. Zimmermann. Scheduling balanced task-graphs to logp-machines. *Parallel Computing*, 26(9):1083–1108, 2000.

[9] Welf Löwe and Wolf Zimmermann. Upper time-bounds for executing PRAM-programs on the logP-machine. In *1995 International Conference on Supercomputing*, pages 41–50, Barcelona, 1995. ACM Press.

[10] M. Shaw and D. Garlan. *Software Architecture in Practice – Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[11] C. Szyperski and R. Vernik. A case for tired component frameworks. In *Workshop on Compositional Software Architectures (Monterey, California)*, 1998.

[12] W. Zimmermann and W. Löwe. On integrating scheduling into compilers for parallel languages. *Submitted to Special issue of Journal of Parallel and Distributed Scientific and Engineering Computing (JPDSEC)*, 2004.

[13] Wolf Zimmermann and Welf Löwe. An approach to machine-independent parallel programming. In *CONPAR '94, Parallel Processing*, volume 854 of *LNCS*, pages 277–288. Springer, 1994.

[14] Wolf Zimmermann, Welf Löwe, and Johannes Gottlieb. On the design and implementation of parallel algorithms for solving inverse problems. In *Parameter Identification and Inverse Problems in Hydrology, Geology, and Ecology*, pages 283–300. Kluwer Academic Publishers, 1996.