An abstract schema modeling adaptivity management

Marco Aldinucci, Sonia Campa, Massimo Coppola, Marco Danelutto, Corrado Zoccolo, Françoise André, Jérémy Buisson

AN ABSTRACT SCHEMA MODELING ADAPTIVITY MANAGEMENT

Marco Aldinucci and Sonia Campa and Massimo Coppola and Marco Danelutto and Corrado Zoccolo
University of Pisa
Department of Computer Science
Largo B. Pontecorvo 3, 56127 Pisa, Italy
aldinuc@di.unipi.it
campa@di.unipi.it
coppola@di.unipi.it
marcod@di.unipi.it
zoccolo@di.unipi.it

Francoise André and Jérémy Buisson
IRISA / University of Rennes 1
avenue du Général Leclerc, 35042 Rennes, France
fandre@irisa.fr
jbuisson@irisa.fr

Abstract Nowadays, component application adaptivity in Grid environments has been afforded in different ways, such those provided by the Dynaco/AFPAC framework and by the ASSIST environment. We propose an abstract schema that catches all the designing aspects a model for parallel component applications on Grid should define in order to uniformly handle the dynamic behavior of computing resources within complex parallel applications. The abstraction is validated by demonstrating how two different approaches to adaptivity, ASSIST and Dynaco/AFPAC, easily map to such schema.

Keywords: Abstract schema, component adaptivity, Grid parallel component application.

1. AN ABSTRACT SCHEMA FOR ADAPTATION

Adaptivity is a concept that recent framework proposals for Computational Grid take into great account. In fact, due to the unstable nature of the Grid (nodes that disappear because of network problems, changes in user requirements/computing power, variations in network bandwidth, etc.), even assuming a perfect initial mapping of an application over the computing resources, the performance level could be suddenly compromised and the framework has to be able to take reconfiguring decisions in order to keep the expected QoS.

The need to handle adaptivity has been already addressed in several projects (AppLeS [6], GrADS [12], PCL [9], ProActive [5]). These works focus on several aspects of reconfiguration, e.g. adaptation techniques (GrADS, PCL, ProActive), strategies to decide reconfigurations (GrADS), and how to modify the application configuration to optimize the running application (AppLes, GrADS, PCL). In these projects concrete problems posed by adaptivity have been faced, but little investigation has been done on common abstractions and methodology [10].

In this work we discuss, at a very high level of abstraction, a general model of the activities we need to perform to handle adaptivity in parallel and distributed programs.

Our intention is to start drawing a methodology for designing adaptive component environments, leaving in the meanwhile a high degree of freedom in the implementation and optimization choices. In fact, our model is abstract with respect to the implemented adaptation techniques, monitoring infrastructure and reconfiguration strategy; in this way we can uncover the common aspects that have to be addressed when developing a programming framework for reconfigurable applications.
Moreover, we will validate our abstract schema by demonstrating how two completely different approaches to adaptivity fit its structure. We will discuss the Dynaco/AFPAC [7] approach and the ASSIST [4] approach and we will show how, despite several differences in the implementation technologies used, they can be firmly abstracted by the schema we propose.

Before demonstrating its suitability to the two implemented frameworks, we exemplify its application in a significant case study: component-based, high-level parallel programs. The adaptive behavior is derived by specializing the abstract model introduced here. We get significant results on the performance side, thus showing that the model maps to worthwhile and effective implementations [4].

This work is structured as follows. Sec. 2 introduces the abstract model. The various phases required by the general schema are detailed with an example in Sec. 3. Sec. 4 explains how the schema is mapped in the Dynaco/AFPAC framework, where self-adapting code is obtained by semi automated restruc-

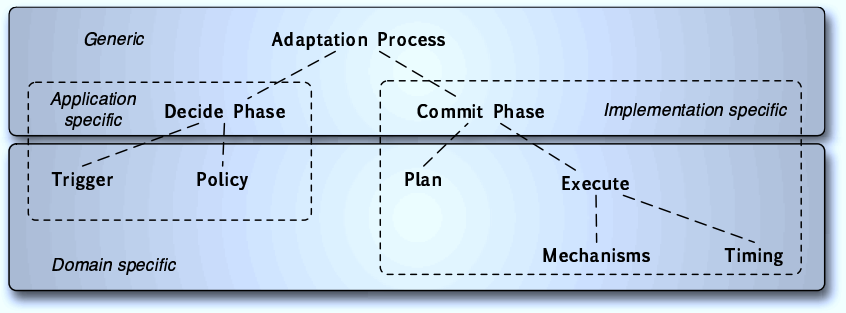

Figure 1 Abstract schema of an adaptation manager.

turing of existing code. Sec. 5 describes how the same schema is employed in the ASSIST programming environment, exploiting explicit program structure to automatically generate autonomic dynamicity-handling code. Sec. 6 summarizes those two mappings of the abstract schema.

2. ADAPTIVITY

The abstract model of dynamicity management we propose is shown in Fig. 1, where high-level actions rely on lower-level actions and mechanisms. The model is based on the separation of application-oriented abstractions and implementation mechanisms, and is also deliberately specified in minimal way, in order not to introduce details that may constrain possible implementations. As an example, the schema does not impose a strict time ordering among its leaves. The process of adapting the behavior of a parallel/distributed application to the dynamic features of the target architecture is built of two distinct phases: a decision phase, and a commit phase, as outlined in Fig. 1. The outcome of the decide phase is an abstract adaptation strategy that the commit phase has to implement. We separate the decisions on the strategy to be used to adapt the application behavior from the way this strategy is actually performed. The decide phase thus represents an abstraction related to the application structure and behavior, while commit phase concerns the abstraction of the run-time support needed to adapt. Both phases are split into different items. The decide phase is composed of:

- trigger – It is essentially an interface towards the external world, assessing the need to perform corrective actions. Triggering events can result from various monitoring activities of the platform, from the user requesting a dynamic change at run-time, or from the application itself reacting to some kind of algorithm-related load unbalance.
- policy – It is the part of the decision process where it is chosen how to deal with the triggering event. The aim of the adaptation policy is to find out what behavioral changes are needed, if any, based on the knowledge of the application structure and of its issues. Policies can also differ in

the objectives they pursue, e.g. increasing performance, accuracy, fault tolerance, and thus in the triggering events they choose to react to.
Basic examples of policy are “increase parallelism degree if the application is too slow”, or “reduce parallelism to save resources". Choosing when to re-balance the load of different parts of the application by redistributing data is a more significant and less obvious policy.

In order to provide the decide phase with a policy, we must identify in the code a pattern of parallel computation, and evaluate possible strategies to improve/adapt the pattern features to the current target architecture. This will result either in specifying a user-defined policy or picking one from a library of policies for common computation patterns. Ideally, the adaptation policy should depend on the chosen pattern and not on its implementation details.

In the commit phase, the decision previously taken is implemented. In order to do that, some assessed plan of execution has to be adopted.

- plan – It states how the decision can be actually implemented, i.e. what list of steps has to be performed to come to the new configuration of the running application, and according to which control flow (total or partial order).
- execute – Once the detailed plan has been devised, the execute phase takes it in charge, relying on two kinds of functionalities of the support code
  - the different mechanisms provided by the underlying target architecture, and
  - a timing functionality to activate the elementary steps in the plan, taking into account their control flow and the needed synchronizations among processes/threads in the application.

The actual adapting action depends on both the way the application has been implemented (e.g. message passing or shared memory) and the mechanisms provided by the target architecture to interact with the running application (e.g. adding and removing processes to the application, moving data between processing nodes and so on). The general schema does not constrain the adaptation handling code to a specific form. It can either consist in library calls, or be template-generated, it can result from instrumenting the application or as a side effect of using explicit code structures/library primitives in writing the application. The approaches clearly differ in the degree of user intervention required to achieve dynamicity.

3. EXAMPLE OF THE ABSTRACT DECOMPOSITION

We exemplify the abstract adaptation schema on a task-parallel computation organized around a centralized task scheduler, continuously dispatching works to be performed to the set of available processing elements. For this kind of

pattern, both a performance model and a balancing policy are well known, and several different implementations are feasible (e.g. multi-threaded on SMP machines, or processes in a cluster and/or on the Grid). At steady state, maximum efficiency is achieved when the overall service time of the set of processing elements is slightly less than the service time of the dispatcher element.

Triggers are activated, for instance, when (1) the average inter-arrival time of task incoming is much lower/higher than the service time of the system, (2) on explicit user request to satisfy a new performance contract/level of performance, (3) when built-in monitoring reports increased load on some of the processing elements, even before service time increases too much.

Assuming we care first for computation performance and then resource utilization, the adaptation policy could be like the following: *i)* when steady state is reached, no configuration change is needed; *ii)* if the set of processing elements is slower than the dispatcher, new processing elements should be added to support the computation and reach the steady state *iii)* if the processing elements are much faster than the dispatcher, reduce their number to increase efficiency.

Applying this policy, the decide phase will eventually determine the increase/decrease of a certain magnitude in the allocated computing power, independently of the kind of computing resources.

This decision is passed to the commit phase, where we must produce a detailed plan to implement it (finding/choosing resources, devising a mapping of application processes where appropriate).

Assuming we want to increase the parallelism degree, we will often come up with a simple plan like the following: *a)* find a set of available processing elements $\{P_i\}$; *b)* install code to be executed at the chosen $\{P_i\}$ (i.e. application code, code that interacts with the task scheduler and for dinamicity handling) ;*c)* register with the scheduler all the $\{P_i\}$ for task dispatching; *d)* inform the monitoring system that new processing element have joined the execution. It is worthwhile that the given plan is general enough to be customized depending on the implementation, that is it could be rewritten/reordered on the basis of the desired target.

Once the detailed plan has been devised, it has to be executed and its actions have to be orchestrated, choosing proper timing in order that they do not to interfere with each other and with the ongoing computation.

Abstract timing depends on the implementation of the mechanisms, and on the precedence relationship that may be given in the plan. In the given example, steps 1 and 2 can be executed in sequence, but without internal constraint on timing. Step 3 requires a form of synchronization with the scheduler to update its data, or to suspend all the computing elements, depending on actual implementation of the scheduler/worker synchronization. For the same reason,

execution of step 4 also may/may not require a restart/update of the monitoring subsystem to take into account the new resources.

We also want to point out that in case of data parallel computation (as a fast Fourier transformation, as instance), we could again use policies like *i)-iii* and plans like *a-d*.

4. DYNACO/AFPAC: A GENERIC FRAMEWORK FOR DEVELOPERS TO MANAGE ADAPTATION

Dynaco is a framework allowing developers to add dynamic adaptability to software components without constraining the programming paradigms and tools that can be used. While Dynaco aims at addressing general adaptability problems, AFPAC focuses on the specific case of parallel components.

4.1. DYNACO: GENERIC DYNAMIC ADAPTATION FRAMEWORK

Dynaco provides the major functional decomposition of dynamic adaptability. It is the part that is the closest from the abstract schema described in section 2. Its design has benefited from the joint work about the abstract schema. As depicted by Fig. 2, Dynaco defines 3 major functions for dynamic adaptability: decision-making, planning and execution. Coarsely, those decision-making and execution functions match respectively the decide and commit phases of the abstract schema.

For the decision-making function, the *decider* decides whether the component should adapt itself or not. If it should, a strategy is produced that describes the configuration the component should adopt. The framework states that the *decider* is independent from the actual component: it is a generic decision-making engine. It is specialized to the actual component by a *policy*, which plays the same role as its homonym in the abstract schema. While the abstract schema reifies in trigger the events triggering the decision-making, Dynaco

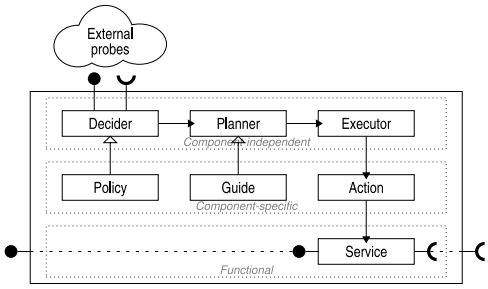

Figure 2 Overall architecture of a Dynaco component.

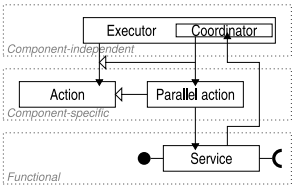

Figure 3 Architecture of AFPAC as a specialization of Dynaco.

does not: the *decider* only exports interfaces to the outside of the component. Monitoring engines are considered to be external to the component and to its adaptability, even if the component can bind to itself in order to be one of its monitors.

The planning function is implemented by the *planner*. Given a *strategy* that has been previously decided, it aims at determining a *plan* that indicates how to adopt the *strategy*. The *plan* matches exactly its homonym of the abstract schema. Similarly to the *decider*, the *planner* is a generic engine that is specialized to the actual component by a *guide*.

While not being a phase in the abstract schema, planning has been promoted to a major function within Dynaco, at the same level as decision-making and execution. As a consequence, Dynaco introduces a planning *guide* in order to specialize the planning function in the same way that there is a *policy* that specializes the decision-making function. On the contrary, the abstract schema exhibits a plan which actually links the decide and commit phases. This vision is consistent with the goal of not constraining possible implementations. Dynaco is one interpretation of the abstract schema, while another would have been to have the decide phase directly produce the plan, for example.

The execution function is realized by the *executor* that interprets the instructions of the *plan*. Two kinds of instructions can be used in *plan*s: invocations of elementary *action*s, which match the mechanisms of the abstract schema; and control instructions, which match the timing functionality of the abstract schema. While the former are provided by developers as component-specific entities, the latter are implemented by the *executor* in a component-independent manner.

4.2. AFPAC: DYNAMIC ADAPTATION OF PARALLEL COMPONENTS

As seen by AFPAC, parallel components are components that encapsulate a parallel code, such as GridCCM [11] components: they have several processes that execute the *service* they provides. AFPAC is depicted by Fig. 3. It is a specialization of Dynaco's *executor* for parallel components. Through its *coordinator* component, which partly implements the timing functionality of the abstract schema, AFPAC provides an additional control instruction for expressing *plan*s. This instruction makes all of *service* processes execute an *action* in parallel. Such an action is labeled *parallel action* on Fig. 3. This kind of instruction is particularly useful to execute redistribution in the case of data-parallel applications.

AFPAC addresses the consistency problems of the global states from which the parallel *action*s are executed. Those problems have been discussed in [7]; we have proposed in [8] an algorithm that chooses the next upcoming consistent

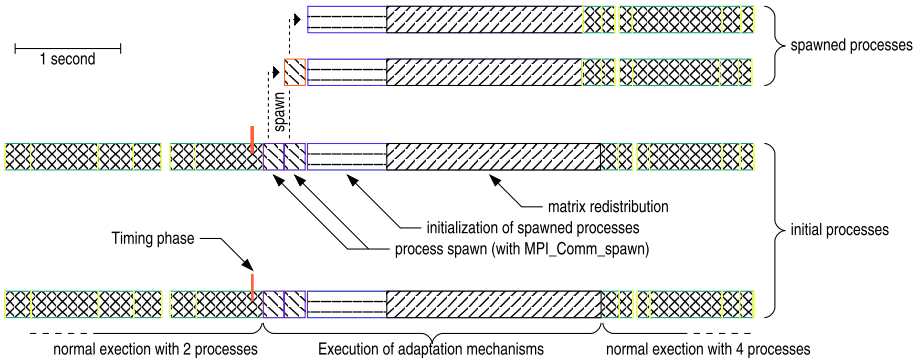

Figure 4 Scenario of an adaptation with AFPAC

global state. To do so, it relies on *adaptation point*s: a global state is said consistent if every service process is at such a point. It also requires control structures to be annotated thanks to aspect-oriented programming in order to locate *adaptation point*s as the execution progresses. The algorithm and the consistency criterion it implements suits well to SPMD codes such as the ones using MPI.

Fig. 4 shows the sequence of actions when a data-parallel code working on matrices adapts itself thanks to AFPAC. In this example, the application spawns 2 new processes in order to increase its parallelism degree up to 4. Firstly, the timing phase of the abstract schema is executed by the *coordinator* component concurrently to the normal execution of the parallel code. During this phase, the *coordinator* takes a rendez-vous with every executing *service* process at an *adaptation point*. When *service* processes reach the rendez-vous *adaptation point*, they execute the requested *action*s. Once every action of the *plan* has been executed, the *service* resumes its normal execution. This experiment shows well that most of the overhead lies in incompressible *action*s like matrix redistribution.

5. ASSIST: MANAGING DYNAMICITY USING LANGUAGE AND COMPILATION APPROACHES

ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of (possibly) parallel modules, interconnected by typed streams of data. A parallel module (*parmod*) coordinates a set of concurrent activities called *Virtual Processes* (VPs). Each VP execute a sequential function (that can be programmed using standard sequential languages e.g. C, C++, Fortran) on input data and internal state.

Groups of VPs are grouped together in processes called *Virtual Processes Manager* (VPM). VPs assigned to the same VPM execute sequentially, while different VPMs run in parallel: therefore the actual parallelism exploited in a *parmod* is given by the number of VPMs that are allocated.

Overall, a *parmod* may behave in a data-parallel (e.g. SPMD/for-all/apply-to-all) or task-parallel way (e.g. farm, pipeline), and it can nondeterministically accept from one or more input streams a number of input items, which may be decomposed in parts and used as function parameters to activate VPs. A *parmod* may also exploit a distributed shared state, which survives between VP activations related to different stream items. More details on the ASSIST environment can be found in [13, 3].

An ASSIST module (or a graph of modules) can be declared as a component, which is characterized by *provide* and *use* ports (both one-way and RPC-like), and by *Non-Functional* ports. The latter are responsible of specifying those aspects related to the management/coordination of the computation, as well as the required performance level of the whole application or of the single component. As instance, among the non-functional interfaces there are those related to QoS control (performance, reconfiguration strategy and allocation constraints).

Each ASSIST module in the graph encapsulated by the component is controlled by its own MAM (Module Adaptation Manager), a process that coordinates the configuration and adaptation of the module itself. The MAM dynamically decides the number of allocated VPMs and their mapping onto the processing elements acquired through a retargetable middle-ware, that can be adapted to exploit clusters as well as grid platforms.

Hierarchically, the set of MAMs is coordinated by the Component Adaptation Manager (CAM) that manages the configuration of the whole component. At a higher level, these lower-level entities are coordinated by a (possibly distributed) Application Manager (AM), to pursue a global QoS for the whole application.

The starting configuration is determined at load time by hierarchically splitting the user provided QoS contract between each component and module. In case of a QoS contract violation during the application run, managing processes react by issuing (asynchronous) adaptation requests to controlled entities [4]. According to the locality principle, violations and corrective actions are detected and issued as near as possible to the leaves of the hierarchy (i.e. the modules with their MAM). Higher-level managers are notified of violations when lower-level managers cannot handle them locally. In these cases, CAMs or the AM can coordinate the actions of several MAMs and CAMs (e.g. by renegotiating contracts with them) in order to implement a non-local adaptation strategy.

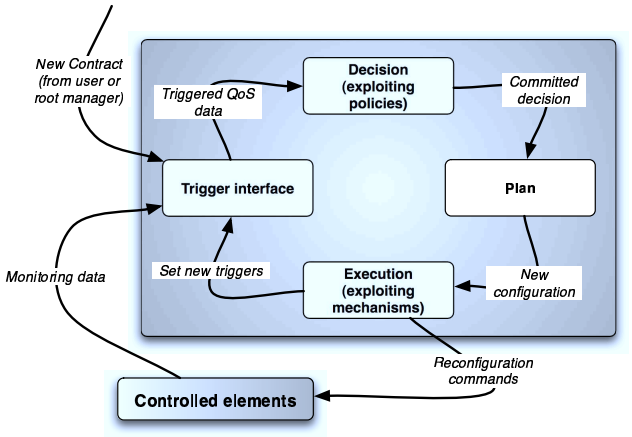

Figure 5 ASSIST framework.

The corrective actions that can be undertaken in order to fulfill the contracts, eventually lead to the adaptation of component configurations, in terms of parallelism degree, and process mapping [4].

Reconfiguration requests to the adaptation mechanism are triggered by new QoS needs or by monitoring feedbacks. Such requests flow in an autonomic manner through the AM to the lower level managers of the hierarchy (or vice versa). If the contract is broken a new configuration is defined by evaluating the related performance model. It is then applied at each involved party (component or module), in order to reach a state in which the contract is fulfilled.

The adaptation mechanisms adopted in ASSIST completely instantiates the abstract schema provided above by organizing its leafs, left to right in an autonomic control loop (see Fig.5). The trigger functionality is represented by the collection of the stream of monitoring data. Such data come from the running environment and can cause a framework reaction if a contract violation is observed. A component performance model is evaluated (policy phase) on the basis of the collected monitoring data, according to a selected goal (currently, in ASSIST we implemented two predefined policies, pursuing two different goals; for special needs, user-defined policies can be programmed).

If the QoS contract is broken, a decision has to be taken about how to adapt the component: such decision could involve a single component or a compound component. In the latter case, the decision has to flow through the hierarchy of managers in order to harmonize the whole application performance. The decision phase uses the policies in order to reach the contract requirements. Examples of policies are: reaching a *desired service time* (as seen above, it could happen if one VPM becomes overloaded), or realizing the *best effort* in the performance/resource trade-off (by releasing unused PE, as instance). The decision phase result is a target for the commit phase (increasing of computing

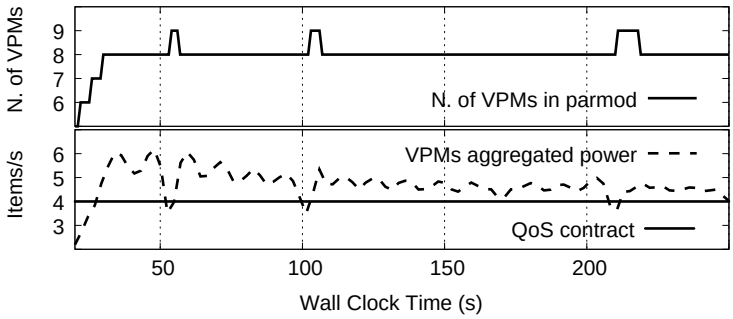

Figure 6 MAM's reaction to a contract violation

power, as an example). Such target is represented by a plan provided by the homonymous phase that lists the actions (e.g. add or remove resource to/from computation and computation remapping, with associated data migration and global state consolidation) to be taken.

Finally, the execute functionality exploits support code statically generated by the ASSIST compiler, and coordinates it with services provided by the component framework to interface to the middle-ware (e.g. for resource recruiting), according to the schedule provided by timing functionality.

Timing functionality is related to the so-called *reconf-safe* points [4], i.e. points in the application code where the distributed computation and state are known to be consistent and can be efficiently synchronized. Each mechanism that is exploited to reconfigure the application at run time can take advantage (e.g. can be optimized) of reconf-safe points to appropriately orchestrate synchronizations in a transparent manner. Moreover, the generated code is tailored for the given application structure and features, exploiting the set of concrete mechanisms provided by the language run-time support. For instance, no state migration code is inserted for stateless computations, and depending on the parallelism pattern (e.g. stream versus data parallel), VPMs involved in the synchronization can be a subset of those within the component being reconfigured.

Our experiments [4] show that the adaptation mechanisms do not introduce overhead with respect to non-adaptive versions of the same code, when no configuration change is performed, and that issued adaptations are achieved with minimal (of the order of milliseconds) impact on the ongoing computation.

Fig. 6 shows the behavior of an ASSIST parallel application with adaptivity managers enabled, run on a collection of homogeneous Linux workstations interconnected by switched Fast Ethernet. In particular, it shows the reaction of a MAM to a sudden contract violation with respect to the number of VPMs. The application represents a farm computing a simple function with fixed service

time on stream items flowing at a fixed input rate. In this scenario, a contract violation occurs when one of the VPMs becomes overloaded, causing the VPMs aggregated power to decrease. The MAM reacts to such decrement by mapping as many VPMs as needed to satisfy the contract (only one in this case) onto fresh computing resources.

In this example, when a new VPM mapping occurs because of the overloading of one (or more) of the allocated ones, removing the overloaded one does not lead to a contract violation. Therefore the MAM, that is also responsible to manage over-dimensioned resource usage, removes the overloaded PE almost immediately. The MAM can reduce resource usage also (not shown in this example) when the incoming data rate decreases or the contract requirements are weakened.

6. A COMPARATIVE DISCUSSION

As it is clear from the previous presentations, Dynaco (and its parallel-component specialization AFPAC) and ASSIST fit the abstract schema proposed in Section 2 in different manner. The frameworks have been developed independently with each other but both aim at offering a platform to handle dynamically adaptable components.

Dynaco can be seen as a pipelined implementation of the abstract schema feeded by an external monitoring engine. In particular, the three major functions (decision-making, planning and execution) are specialized by component-specific sub-phases. On the other hand, ASSIST provides a circular implementation of the schema leafs, while decision and commit can be seen as macro-steps of the autonomic loop.

The decision-making functionalities are triggered by the external monitoring engine in Dynaco, while in ASSIST the concept of performance contract is exploited in order to specify the performance level to be guaranteed.

In ASSIST the code related to the execution phase is automatically generated at compile time, while the Dynaco developer is asked to provide the code for policy, guide and action entities. Both the frameworks offer the possibility to configure certain points of the code as “safe-points” from which recovery/reconfiguration is possible. In Dynaco such points are defined by aspect-oriented technologies, while in ASSIST they are defined by the language semantics, and determined by the compiler.

From the discussion above, it is clear that each framework affords the adaptivity problem by means of individual solutions. What we want to point out in this work is that, despite their technological diversity, both solutions can be inscribed in the general abstract schema presented in Section 2. Such schema is general enough to abstract from any kind of implementative solution but it is also sufficiently strong to catch the salient aspects a model has to consider

while designing adaptive component frameworks. By summing up, it can be seen as a reference guide for modeling adaptable environments independently from the implementations, technologies, languages, constraints or architectures involved.

7. CONCLUSIONS

We have described a general model to provide adaptive behavior in Grid-oriented component-based applications. The general schema we have shown is independent of implementation choices, such as the responsibility for inserting the adaptation code (either left to the programmer, as it happens in the Dynaco/AFPAC framework, or performed by exploiting knowledge of the high level program structure, as it happens in the ASSIST context). The model also encompasses user-driven as well as autonomic adaptation.

The abstract model helps in separating application and run-time programming concerns of adaptation, exposing adaptive behavior as an aspect of application programming, formalizing the concerns to be addressed, and encouraging an abstract view of the run-time mechanisms for dynamic reconfiguration.

This formalization gives the basis for defining a methodology. The given case study provide with valuable clues about how to solve different concerns, and how to identify common parts of the adaptation that can be generalized in support frameworks. The model can be thus also usefully applied within other programming frameworks, like GrADS, which do not enforce a strong separation of adaptivity issues into design and implementation.

We expect that such a methodology will lead to more portable and understandable adaptive applications and components, and it will also promote layered software architectures for adaptation, simplifying implementation of both the programming framework and the applications.

Acknowledgments

This research work is carried out under the FP6 Network of Excellence *CoreGRID* funded by the European Commission (Contract IST-2002-004265), and it was partially supported by the Italian MIUR FIRB project *Grid.it* (n. RBNE01KNFP) on High-performance Grid platforms and tools.

References

[1] M. Aldinucci, F. André, J. Buisson, S. Campa, M. Coppola, M. Danelutto, and C. Zoccolo. Parallel program/component adaptivity management. In *Proc. of Intl. PARCO 2005: Parallel Computing*, Sept. 2005.

[2] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for high

performance grid programming in grid.it. In V. Getov and T. Kielmann, editors, *Proc. of the Intl. Workshop on Component Models and Systems for Grid Applications*, CoreGRID series, pages 19–38, Saint-Malo, France, Jan. 2005. Springer.

[3] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*, chapter 10, pages 230–256. Springer, Jan. 2006.

[4] M. Aldinucci, A. Petrocelli, A. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of Grid-aware applications in ASSIST. In José. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005 Parallel Processing: 11th International Euro-Par Conference, Lisbon, Portugal, August 30 - September 2, 2005. Proceedings*, volume 3648 of *LNCS*, pages 711–781. Springer-Verlag, August 2005.

[5] F. Baude, D. Caromel, and M. Morel. On hierarchical, parallel and distributed components for Grid programming. In V. Getov and T. Kielmann, editors, *Workshop on component Models and Systems for Grid Applications*, ICS '04, Saint-Malo, France, June 2004.

[6] F. D. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Supercomputing '96: Proc. of the 1996 ACM/IEEE Conf. on Supercomputing (CDROM)*, page 39, 1996.

[7] J. Buisson, F. André, and J.-L. Pazat. Dynamic adaptation for grid computing. In P.M.A. Sloot, A.G. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, editors, *Advances in Grid Computing - EGC 2005 (European Grid Conference, Amsterdam, The Netherlands, February 14-16, 2005, Revised Selected Papers)*, volume 3470 of *LNCS*, pages 538–547, Amsterdam, June 2005. Springer-Verlag.

[8] J. Buisson, F. André, and J.-L. Pazat. Enforcing consistency during the adaptation of a parallel component. In *The 4th Int.l Symposium on Parallel and Distributed Computing*, July 2005.

[9] B. Ensink, J. Stanley, and V. Adve. Program control language: a programming language for adaptive distributed applications. *Journal of Parallel and Distributed Computing*, 63(11):1082–1104, November 2003.

[10] M. McIlhagga, A. Light, and I. Wakeman. Towards a design methodology for adaptive applications. In *Mobile Computing and Networking*, pages 133–144, May 1998.

[11] Christian Pérez, Thierry Priol, and André Ribes. A parallel corba component model for numerical code coupling. *The International Journal of*

High Performance Computing Applications (IJHPCA), 17(4):417–429, 2003.

[12] S. Vadhiyar and J. Dongarra. Self adaptability in grid computing. *International Journal Computation and Currency: Practice and Experience*, 2005. To appear.

[13] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, December 2002.