Towards dynamic adaptability support for the master-worker paradigm in component based applications

Françoise André, Hinde Lilia Bouziane, Jérémy Buisson, Jean-Louis Pazat, Christian Pérez

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Towards dynamic adaptability support for the master-worker paradigm in component based applications

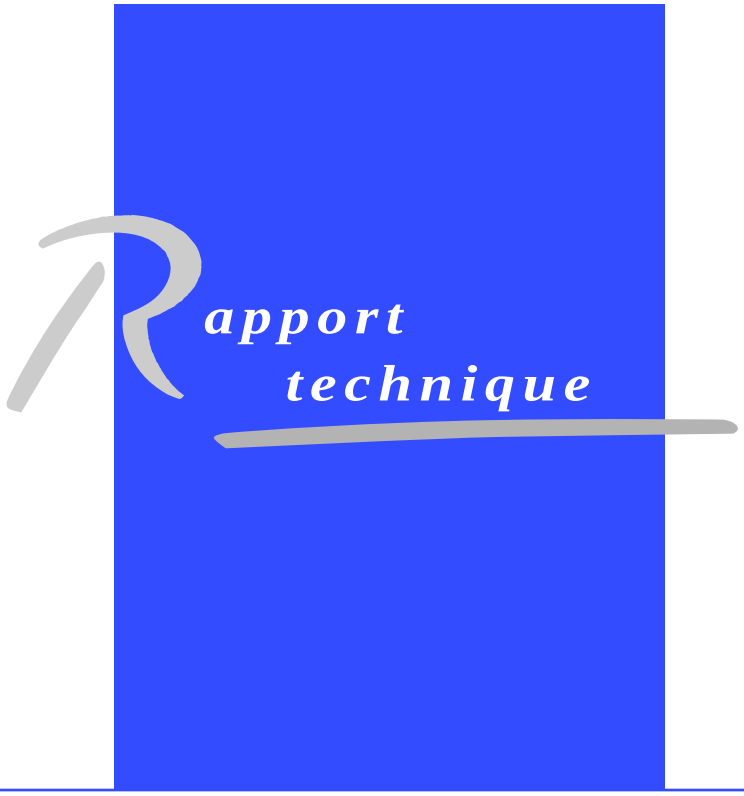
Françoise André — Hinde Lilia Bouziane — Jérémy Buisson

Jean-Louis Pazat — Christian Pérez

N° 0333

March 2007

Thème NUM



ISSN 0249-0803 ISRN INRIA/RT--0333--FR+ENG

Towards dynamic adaptability support for the master-worker paradigm in component based applications

Franoise André, Hinde Lilia Bouziane, Jérémy Buisson
Jean-Louis Pazat, Christian Pérez

Thème NUM — Systèmes numériques
Projet Paris

Rapport technique n° 0333 — March 2007 — 12 pages

Abstract: When executing scientific applications, resources that may be used can vary from multi-core processors to grids. Therefore, abstracting the programming model enables portability on various resource infrastructures. Furthermore, software component technology appears to be a very promising approach to deal with the growing complexity of scientific applications. Hence, we proposed a model to improve the support of *master-worker* paradigm in component models. Capitalizing on our experience of adaptability frameworks, we propose to enhance our model so that *master-worker* applications can adapt at runtime to varying conditions. This report studies how to introduce transparently adaptability in our model for *master-worker* applications, what impact it has on the model, and what requirements it expects from the adaptability framework.

Key-words: Software components, Grid, Master-worker, Dynamic evolution, Adaptability framework

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 —Télécopie : +33 2 99 84 71 71

Vers un support d'adaptation dynamique pour le paradigme matre-travailleur dans les applications base de composants

Résumé : Les plate-formes d'exécution d'applications scientifiques peuvent varier de processeurs multi-core aux grilles de calcul en passant par des grappes. Pour assurer la portabilité de ces applications sur ces différentes plate-formes, il est nécessaire d'avoir des modèles de programmation capables de fournir un niveau d'abstraction suffisamment élevé par rapport aux ressources utilisées. Pour cela, la programmation à base de composants apparaît très prometteuse. De plus, elle permet de simplifier la programmation des applications scientifiques, dont le code ne cesse d'accroître en complexité. Dans des travaux précédents, on a proposé d'améliorer le support du paradigme maître-travailleur dans les applications à base de composants logiciels.En nous basant sur notre expérience en adaptabilité, notre objectif est d'étudier l'adaptation d'une application maître-travailleur aux variations dynamiques des conditions d'exécution. Ce rapport étudie l'intégration d'un canevas d'adaptation de manière transparente, son impacte sur le modèle maître-travailleur proposé ainsi que les services attendus d'un canevas d'adaptabilité.

Mots-clés : Composants logiciels, Grille, Maître-travailleur, évolution dynamique, canevas d'adaptation

1 Introduction

While computing Grids are becoming more and more common, the question of their programmability is raising attention. The underlying motivation not only stems from the high complexity of grids that shall be hidden to programmers but it also comes from the increasing complexity of applications. In order to take advantage of the huge possibilities of grids, more complex applications like code coupling applications are getting popular.

Software component technology appears very promising to handle the complexity of both grids and applications. Code reuse enables to build complex applications based on validated building blocks while component composition provides a mechanism to support complex relationships independently of the architecture of the execution platform.

An example of such a relationship is the *master-worker* paradigm. While it is an algorithmic concept, its implementation varies quite a lot depending on the execution platform. Hence, we defined a high level master-worker relationship between components [5, 4]. While it provides a model close to the abstract concept to the programmers, it can be configured by the execution environment to fit to the actual resources. However, this previous work did not consider dynamic adaptation. For example, the number of workers may change depending on the number of incoming requests or the number of available machines. The goal of this report is to study how to introduce adaptability support in a master-worker paradigm and to evaluate the impact on adaptation frameworks.

The remainder of this report is organized as follow. Section 2 briefly presents the model to handle *master-worker* relationship between components as well as an analysis of the various levels of adaptability. Section 3 introduces a framework of adaptability while Section 4 discusses different strategies to introduce adaptability within the *master-worker* relationship. An example is presented in Section 5. Section 6 concludes the report and presents some future works.

2 A high-level *master-worker* composition model

We proposed in [5] to increase the abstraction level of component models with respect to the *master-worker* (M-W) paradigm. Our motivation is twofold. First, we aim to relieve programmers of dealing with resource dependencies, such as the number of *workers* to instantiate or request transport concerns. Second, we target to reuse existing *master-worker* environments, like DIET [7], as they implement advanced request transport and scheduling algorithms.

2.1 Overview

We present in a Fractal formalism our generic model, which is already projected to specific component models Fractal [5], Ccm and Cca [4].

The model is based on the concept of *collection*, which is defined as a set of *exposed* ports, bound to some internal component type ports. A collection behaves like a component: it

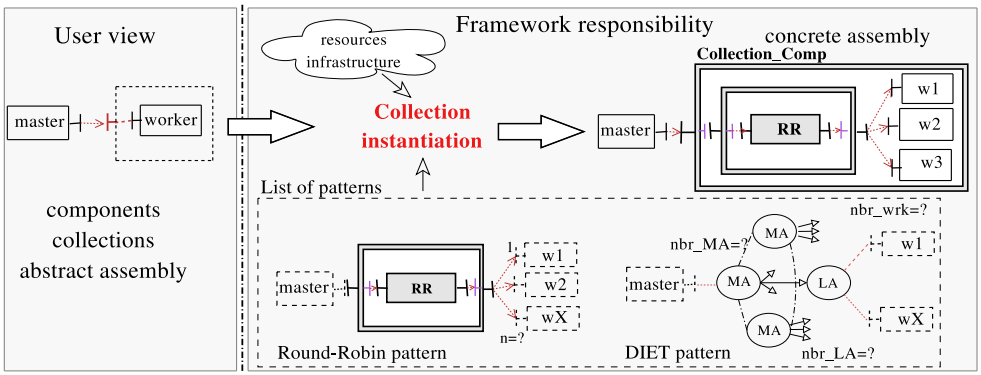


Figure 1: Overview of the *master-worker* model from the user and framework points of view.

can be connected to other components. However, such a composition is done by an *abstract* architecture description, which represents the user's view of the application. Ideally at deployment time, a collection is turned into a composite by defining an initial number of internal component instances and by selecting a *request transport pattern*. A *pattern* represents a requests transport algorithm that may be used between *master* and *worker* components. It is a composite whose implementation should be done by some experts and can or can not be based on software components, such as DIET [7]. Request transport *patterns* are defined independently of a collection. Fig. 1 presents an overview of the concepts of the proposed model.

2.2 Need for dynamic behavior

The proposed model dealt with building a static *master-worker* application because the transition from the abstract collection to a concrete composite fixes the number of workers as well as a pattern at deployment time. However, such choices have to be dynamic to take into account modifications of the application behavior and/or of the resources. The application behavior encompasses collection level behaviors like the frequency and the kind of incoming requests, the number of requests waiting for a worker, or the number of connected masters. It also comprises application's level behavior when there are several collections within an application. Resource behaviors are made of standard considerations like availability, end of a resource reservation, etc.

For a collection, there are three elements that may be dynamically modified: *1)* the number of workers, *2)* the used pattern and *3)* the tuning of the pattern.

For example, let consider an increase of the number of waiting requests. If the pattern is not the bottleneck, the solution is to add more workers if there are available resources. However, if the pattern is the bottleneck, either the pattern may by optimized or it has to be changed by a more scalable one.

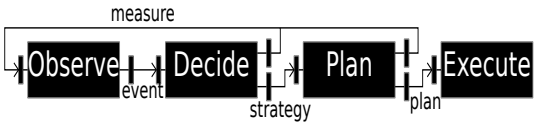


Figure 2: Overview of the DYNACO architecture as an assembly of FRACTAL components.

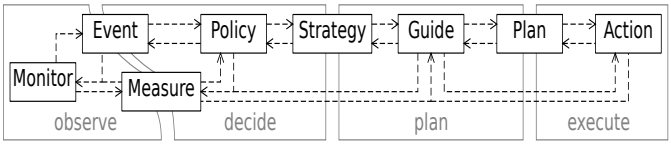


Figure 3: Detailed architecture of the DYNACO framework: entities and dependencies belonging to each FRACTAL component.

In order to help decision making, validity constraints may be attached to a pattern. For instance, a round-robin pattern can be adequate for one or a few connected masters, for equivalent request load and for homogeneous processors. If at least one of these conditions are not met, another pattern should be considered, like for example a load-balacing pattern or DIET.

A collection can also be modified to optimize resources usage. For instance, if there is a lot of workers compared to the number of requests, it can be suitable to remove some workers to release resources. Last, more complex situations occurs when an application contains several instances of the *master-worker* paradigm. In such a case, re-structuring a collection should be coordinated with other collections.

3 A framework for adaptability

In a previous work [1, 6], we have studied how to make applications suit varying conditions, relying on the notion of adaptability. This work led us to develop a generic component framework for adaptability, DYNACO. Benefiting from a joint work with the university of Pisa [1], this framework splits adaptability into four sub-functionalities: *1)* the framework has to be able to *observe* characteristics of the environment in order to trigger adaptability; *2)* when a change is detected, the framework has to *decide* an adaptation strategy according to observed measures; *3)* once a strategy has been decided, the framework has to *plan* actions to implement it; at last, *4)*, planned actions have to be *executed* synchronously with the execution of applicative code. In Fig. 2, each sub-functionality is captured by a FRACTAL component.

Rather than reimplementing the components of the framework specifically to each application, developers are encouraged to focus on application-specific issues, thanks to the reuse of existing generic engines. For instance, we have experimented 3 generic engines for the

decide component: *1)* a JAVA virtual machine, such that the decision procedure is implemented with a general-purpose language, allows easy implementations of intuitive decision procedures; *2)* the JESS [10] expert system, such that the decision procedure is expressed with a domain-specific rule language (i.e. as a collection of ordered rules that looks like the following statement: *"decide a given strategy when an associated condition becomes true"*), allows efficient implementations of complex rule-based decision procedures; and *3)* a genetic algorithm, such that the decision procedure is expressed as a function to optimize (e.g. the performance model of the application), allows to implement straightforwardly decision procedures when the application's behavior is well formalized, possibly with a higher runtime cost. As described, each engine proposes a different trade-off.

The same applies to the *plan* and *execute* components. For the former, for instance, a pattern matching based mapping from strategies to predefined plans suits well simple cases; while more sophisticated formalisms such as STRIPS [12] (developers only declare the collection of possible actions as pre- and post-conditions) may be relevant when developers cannot predefine plans by hand. Similarly, synchronizing adaptation actions with the applicative code depends mostly on the applicative programming model: we have proposed an algorithm (AFPAC [6]) for any SPMD application. Other algorithms could be used such as ASSIST [3] when using its *parmod* skeleton programming model.

The *observe* component does not adhere to the same design: *monitor*s are facilities provided by the environment itself that are wrapped into adapter components, which gather, aggregate and preprocess raw measures and events to their expected formats. The whole *observe* component is almost independent of the application and does not need any particular specialization.

As shown in Fig. 3, application-specific code is captured in *policy*, *guide* and a collection of *action*s, which respectively specialize the *decide*, *plan* and *execute* components. Using generic engines in that way is what makes DYNACO highly generic and open, while it encourages effective code reuse.

4 Design choices for adaptability in the M-W paradigm

This section studies how to make use of an adaptability framework such as DYNACO in the *master-worker* paradigm. It analyzes two major design choices we have identified: the choice of the adaptability strategy and of the architecture. The discussion is done with respect to three criteria: modularity, accuracy, and scalability. *Modularity* measures the possibility to compound strategies such as at the collection level and at the pattern level. *Accuracy* stands for the kinds of allowed adaptations while *scalability* refers to the number of components in the collection.

4.1 Strategy level

The first choice concerns the way to logically design the adaptation strategy, which can be *monolithic*, *independent* or *coordinated*.

Considering a single *monolithic* strategy, the global strategy should handle any possible situation and adaptation for the whole collection. Especially, it should consistently handle the adaptation at the level of the collection, the pattern, and the pattern implementation. For instance, observing that the request queue lengthens, instantiating new workers may increase the heterogeneity of processors, such that the pattern should be replaced by a more suited one (e.g. switching from round-robin to DIET). A monolithic strategy is able to handle those two adaptations at once. Assuming now that the bottleneck is the pattern, which may not be able to perform better, not even with a different implementation nor with additional resources. Being aware of all of the implemented patterns, a monolithic strategy has sufficient knowledge to detect such a situation and prevent useless workers. Therefore, high accuracy is provided. However, the major drawback is the poor modularity. Indeed, the tight entanglement between adaptations makes it particularly difficult to add incrementally the support for new patterns, as well as to maintain the strategy, as any local modification may have an impact on the whole strategy. Worse, in the case of a multi-collection application, adaptations for all of the collections have to be handled by a single strategy at the level of the whole application.

Rather than designing the strategy as a whole, it may be better to decompose it such that the specification of the strategy for each adaptation is close to what is adapted. Basically, in order to allow good modularization, 3 sub strategies would be designed: the first one, attached to the collection composite, adapts the number of workers; the second one, attached to the pattern, selects a convenient pattern; and the last one, attached to the pattern implementation, optimizes the pattern. Two alternatives can be derived from this compound strategy. Each sub strategy may be *independent* or otherwise it may be *coordinated*. In the former case, independence means that no explicit interaction occurs from one sub strategy to the others. The latter case allows explicit interactions between sub strategies such that they can coordinate the adaptations of the elements of the collection. Any technique can be used to implement the coordination, such as triggering adaptations from other adaptations (propagating adaptations) or running a negotiation protocol (agreeing on adaptations).

Focusing on the *independent* approach, let us consider first the above example of adding worker instances that increase heterogeneity, which may result from different processors or from different implementations. Independence implies that the pattern switches its implementation on its own when it observes that heterogeneity increases, once the collection (independently) has instantiated new workers. Thus, despite their independence, the sub strategies achieve together the same adaptations as the single monolithic strategy. However, that way of observing effects of adaptations is not always enough to implement accurate adaptations. Consider that the queue lengthens. An accurate strategy does not instantiate new workers if the pattern would not be able to dispatch requests at a sufficient pace; and it does not optimize the pattern if there can't be enough workers to handle requests. However, independence of the sub strategies prevents the collection from knowing whether the pattern would be able or not to dispatch requests to additional workers; and it prevents the pattern from knowing whether the collection would be able to instantiate new workers. In such a situation, this strategy would desperately preserve the *status quo*, even if the collection

Strategy	Accuracy		Modularity	Scalability
	Collection	Application		
Monolithic	High	None	None	Low
Independent	Low	Low	High	High
Coordinated	High	High	High	High

Figure 4: Summarized features of each alternative for the strategy level.

would be able to perform better; while lowering accuracy may lead to instantiate useless workers or to over-optimize the pattern.

Last, the *coordinated* strategy promises to bring the advantages of the two other strategies without their drawbacks (Fig. 4). It preserves compound strategies for modularity and scalability while letting a global vision to be built for accuracy. However, several adaptation modules have to be interconnected.

4.2 Achitecture level

The second design choice concerns the architecture of the adaptability. Two alternatives are identified: *centralized* and *distributed*.

A *centralized* architecture locates the whole adaptability management into a single location. With respect to the model presented in Section 2, it has to be into the membrane of the collection. Bindings are, nevertheless, present to enable it to control the whole collection. The *centralized* approach is compatible with all adaptation strategies described in Section 4.1. It also simplifies the implementation of the *coordinated* strategy as the communication between the different strategies may be embedded into the same adaptation framework. However, it raises an issue for the compound strategies with respect to the composition of components: the adaptation part of sub components needs to be injected into the adaptation part of the collection. Hence, the connection operator of the component model turns out to be more complex. As far as we know, there is no standard component models that permits it.

With a *distributed* architecture, the adaptability management is spread over the whole collection, and in particular in the membranes of the collection, of the pattern and of the pattern implementation components. The *distributed* architecture is not straightfowardly compatible with the *monolithic* strategy. However, it perfectly fits with the counpound strategies provided that the communications of the *coordinated* strategy are quite simply done through some ports. Considering the advantages of the *coordinated* strategy, we conclude that this strategy with a *distributed* architecture appears to be the best choice to deal with dynamic change in a collection.

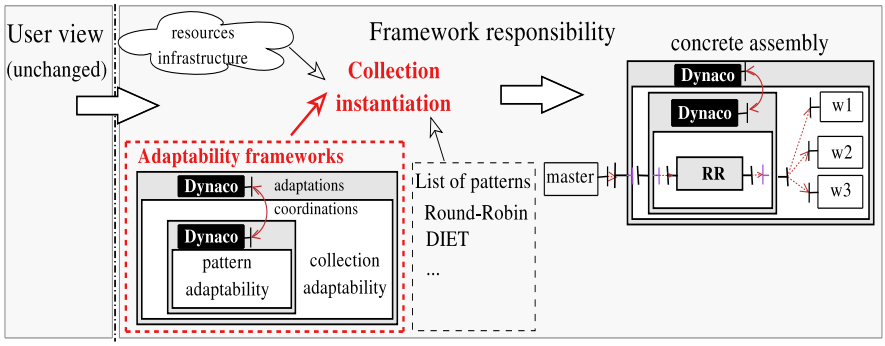


Figure 5: Introducing dynamic management in the *master-worker* model.

4.3 Positionning

Only a couple of adaptability frameworks address the problem of coordinating and distributing adaptations. DYNACO is neutral as it does not prevent policies to coordinate on their own, but it does not provide any specific support.

Among the other frameworks, ACEEL [8] and PLASMA [11], are fairly close to the constructive approach defended here, i.e. building global adaptations as the collaboration of individual local adaptations. With our previous framework ACEEL, each component contacts other components before adapting, in order to ensure consistent and synchronized adaptation of the whole assembly. With PLASMA, components impose their adaptations to the other ones through a simpler propagation mechanism. The contract-driven approach of ASSIST [2] is different: considering a hierarchical component model, composite components divide their contract in order to assign recursively subcontracts to their sub components. Coordination of adaptation is enforced by the submission of contracts that are consistent with another. However, this approach requires composite components to have precise understanding of the composition of their immediate subcomponents, in order to devise subcontracts.

Those frameworks are however tied to the programming models for which they have been specifically designed, often restricted to a fixed collection of predefined adaptations; while focusing only on adaptability, DYNACO integrates gracefully to any programming model. Consequently, DYNACO is a better start point.

4.4 Impact on the *master-worker* model

Section 4.2 concludes with the choice of a *coordinated* strategy with a *distributed* architecture. Let us consider now the impact of such choice on the model presented in Section 2. Achieving the objective of transparent dynamic management, there is no need to modify the model at the user view level. The collection instantiation process seems to be more appropriate to introduce an adaptability framework. A collection implementation, in particular the collection and pattern membranes, are determined at this stage. Adding adaptability

framework as controllers in appropriate membranes appears to be straightforward. However, the diversity of resource infrastructures and resource management systems lead to various adaptability policies. For instance, a policy can be more constrained by resource availability when resource sharing is privileged, otherwise it can be more constrained by application requirements. As a consequence, similarly as for patterns, the framework has to do a selection from a set of adaptability implementations. Fortunately, the specificity of DYNACO to be component-based allows the use of different implementations. The *master-worker* model extended with adaptability support is presented in Fig. 5.

5 Example of *master-worker* application adaptation

Based on the preceding analysis, this section presents a concrete example of design for the adaptability of a *master-worker* application. As outlined in Section 2.2, the adaptation aims at preventing the request queue from unacceptably growing, while making the queue contain enough requests to feed continuously the workers. In order to enforce that objective, we propose the following intuitive compound strategy, using the coordinated approach:

- **at the level of the collection:** if the request queue lengthens beyond a threshold, if the pattern is able to increase its dispatch rate accordingly and if there are available resources, then instantiate new workers; if the request queue shortens under a threshold, then terminate some workers.
- **at the level of the pattern:** if the number of masters or the variability of request durations increases above a threshold, or if the heterogeneity of workers increases beyond a threshold, then switch to the DIET pattern; otherwise, under a threshold switch to round-robin.

In this strategy, coordination occurs before the collection instantiates new workers. It actually asks the pattern whether it would be able to dispatch requests at a sufficient rate, for instance involving a contract renegotiation protocol. The length of the request queue cannot always be observed directly; lengthenings and shortenings can nevertheless be deduced from the comparison between arrival and service rates. Other observations are almost obvious. The examples shows that the *coordinated* and *distributed* design suits well and that the necessary monitoring does not breach encapsulation.

6 Conclusion

The report analyses how to design dynamic adaptability support for component-based *master-worker* applications. Among the discussed possibilities, coordinating several distributed adaptations appears to be the best-suited solution with regard to modularity, scalability and accuracy. In addition, integrating adaptability at the level of the *master-worker* abstraction achieves the goal of hiding the management of execution resources from the developers' sight.

Among adaptability frameworks, none fully meets the requirements of our proposal. Its genericity and openness make DYNACO be the best start point. Based on the experience

we gained in our previous work on ACEEL [8, 9], we plan to extend DYNACO with specific support for the coordination of distributed adaptations, so that it meets the requirements. We will also evaluate the proposed model on synthetic *master-worker* benchmarks as well as the possibilities to write generic adaptation policies at the collection and application levels.

References

[1] M. Aldinucci, F. André, J. Buisson, S. Campa, M. Coppola, M. Danelutto, and C. Zoccolo. An abstract schema modelling adaptivity management. In Sergei Gorlatch and Marco Danelutto, editors, *Integrated Research in GRID Computing*, CoreGRID. Springer, 2007.

[2] M. Aldinucci, M. Danelutto, and M. Vanneschi. Autonomic qos in assist grid-aware components. In *14th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, February 2006.

[3] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in assist. In José C. Cunha and Pedro D. Medeiros, editors, *Proceedings of the 11th International Euro-Par Conference*, volume 3648 of *Lecture Notes in Computer Science*, pages 771–781, Lisbon, Portugal, September 2005. Springer.

[4] G. Antoniu, H. L. Bouziane, M. Jan, C. Pérez, and T. Priol. Combining data sharing with the master-worker paradigm in the common component architecture. In *The 15th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, Paris, France, June 2006.

[5] H. L. Bouziane, C. Pérez, and T. Priol. Modeling and executing master-worker applications in component models. In *11th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, Rhodes Island, Greece, April 2006.

[6] J. Buisson, F. André, and J.-L. Pazat. Afpac: Enforcing consistency during the adaptation of a parallel component. *Scalable Computing: Practice and Experience*, 7(3):83–95, September 2006. electronic journal (http://www.scpe.org/).

[7] E. Caron, F. Desprez, F. Lombard, J.M. Nicod, M. Quinson, and F. Suter. A Scalable Approach to Network Enabled Servers. In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International EuroPar Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag.

[8] D. Chefrour. *Plate-forme de composants logiciels pour la coordination des adaptations multiples en environnement dynamique.* PhD thesis, Université Rennes 1, November 2005.

[9] D. Chefrour and F. André. Développement d'applications en environnements mobiles à l'aide du modèle de composant adaptatif ACEEL. In *Langages et Modèles à Objets. Actes publiés dans la revue STI*, volume 9 of *série L'objet*, Vanne, France, 2003.

[10] Jess, the rule engine for the java platform. http://herzberg.ca.sandia.gov/jess/.

[11] O. Layaida and D. Hagimont. Designing self-adaptive multimedia applications through hierarchical reconfiguration. In L. Kutvonen and N. Alonistioti, editors, *DAIS'05*, volume 3543 of *LNCS*, pages 95–107. Springer, 2005.

[12] N. Nilsson and R. Fikes. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803