A Component Platform for Experimenting with Autonomic Composition

Françoise Baude Ludovic Henrio Paul Naoumenko
INRIA Sophia-Antipolis, I3S, Université de Nice Sophia-Antipolis, CNRS
2004 route des Lucioles – B.P. 93
F-06902 Sophia-Antipolis Cedex
{First.Last}@sophia.inria.fr

ABSTRACT

In this paper, we propose a component-oriented framework that can support autonomic computing and in particular bio-inspired approaches. Starting from the Grid Component Model, a component model targeting at Grid computing and already featuring some autonomicity, we show how such a model can be used in a general autonomic computing context. Indeed the model provides hierarchical structure and reconfiguration for both functional and non-functional levels. This should ease the development of *self-** and in particular, self-evolving applications. With our approach, even the autonomic strategies themselves can evolve. We consider this model and its implementation as powerful tools for easily experimenting autonomic behaviours.

Categories and Subject Descriptors

D.1.3 [**Software Engineering**]: Concurrent Programming—*Distributed programming*; D.2.2 [**Software Engineering**]: Design Tools and Techniques

1. INTRODUCTION

The autonomic computing paradigm [25] was inspired by the (complex) human nervous system. Generally speaking, autonomous applications implement complex management strategies through a decentralized independent decision process. Their goal is to ensure the self-* properties [25], and more generally all the self-management features. Those management strategies themselves can be considered as non-functional aspects. For this reason, many researchers claim that it would be very useful for designers and developers of complex autonomic strategies to have a clear separation between the functional and non-functional aspects of the application. To this aim, it is also much easier to work with a clear representation of the functional and non-functional architecture of the application. We propose a programming model and a framework which brings solutions to potentially further ease the development of complex autonomic strategies. Our solution is ground in a component-oriented approach to develop autonomic applications. Besides, our intention is also to enable others to use this platform as a mean to easily experiment new autonomic behaviours: this requires to be able to quickly design and program new behaviours, so the capability to reuse already developed features is a strong requirement. Also, as some autonomic behaviours one may want to experiment with may be inspired by nature (as conducted for instance within the EU funded BIONETS[1] research project we are involved in), we need to provide a framework through which similar but numerous participants must be modelled and then emulated. As sometimes in nature, self-* properties may result from evolution, so a second requirement is that the autonomic behaviours are designed and programmed in a way that permits them to modify themselves even at runtime. Finally, we also foresee that emulations be computation intensive, so we require to be able to deploy and run the platform on a sufficiently big aggregation of computation resources like computing grids are, without additional burden for the experimenter. This is why the component-based framework we propose for experimenting with autonomic behaviours is grounded upon a software component model initially dedicated to the programming of Grid applications: the Grid Component Model (GCM) [15]. Besides, our effort contributes to easing the programming of autonomic Grid applications (which is an active current research track by itself, see e.g. [35, 1]).

Our purpose in this work is not to provide algorithms for autonomic strategies but to provide support for such algorithms, meaning control on the non-functional aspects of a component system, possibility to plug dynamically different management strategies, and runtime support for autonomic systems ...

A component is a software module, with a standardized description of what it needs and provides (called *server (provided)* and *client (required) interfaces*), that can be manipulated by tools for composition and deployment. Interfaces can be connected (bound) together to allow components to interact, and constitute a component assembly.

From a practical point of view, the component model we rely on is *GCM*, detailed in Section 3.2. This model has usual advantages of component models (structure, hierarchy and encapsulation) and offers some reconfiguration primitives (bind, unbind, add and remove components). One of the strong advantages of this model is to represent both the functional and the non-functional parts of the application as a component system. This allows to easily design complex

[1]Bionets EU project (IST-FET-FP6-027748) at `www.bionets.eu`

autonomic strategies. Autonomic strategies can be designed as a component system belonging to the non-functional part of the application. By using reconfiguration possibilities of such a system, these strategies can be dynamically updated. Moreover, the model has been extended to allow consistent (well-defined) interactions between non-functional and functional parts.

This paper is organized as follows. Section 2 studies the different programming methodologies that can be envisioned to program autonomic applications; from this analysis, we choose to focus on component-oriented programming. The remaining of the paper demonstrates how component models can be instrumented to integrate autonomic behaviours. Section 3.2 presents Fractal and the GCM, two related component models. The latter is targeted at large-scale distributed computing, more precisely we focus in this section 3.2 on the structure that can be given to the non-functional concerns in those two models. Section 4 explains how autonomic behaviours can be plugged in component systems, allowing the autonomic management of components; we propose a methodology for programming autonomic components which promotes re-usability and dynamic adaptation of both functional and non-functional components of the application, leading to a component platform suited for experimenting with distributed autonomic behaviours. Finally Section 5 releases the usual constraint of component systems that requires the strong coupling of components and their permanent availability: component composition becomes a plan, for which components fulfilling the services must be discovered dynamically and can disappear at any point in time.

2. PROGRAMMING AUTONOMIC APPLICATIONS

This section shortly reviews possible software techniques to describe and implement the autonomic behaviour of services (rules, aspect weaving, components, etc.) and tries to evaluate if they are effective when the behaviour is complex.

2.1 General Principles

Autonomic computing is a paradigm that proposes to add to software entities some autonomous capabilities: the entity is capable to self-adapt in reaction to context or environmental changes. Adaptation is generally designed and described in an ad hoc way, which involves trying to predict future execution conditions at development time and embedding the adaptation decisions in the application code itself. This approach has several drawbacks: increased complexity (business logic polluted with non-functional concerns) and poor reuse of software caused by a strong coupling with a specific environment. As previously noticed in the litterature (e.g. [17]) adaptations (most notably those related to resource usage) can be decoupled from pure functional concerns. This approach does not have the same drawbacks as the ad hoc way.

We believe that the context of autonomic and situated communications is specially demanding to motivate a systematic (as opposed to ad hoc) way to develop self-adaptive software based on the Separation of Concerns principle: adaptation to a specific execution context and its evolutions is considered as a concern which should be treated separately from the rest of an application. Ideally, application developers should be able to concentrate on pure business logic, and write their code without worrying about the characteristics and resource limitations of the platform(s) it will be deployed and run on. Then, the adaptation logic, which deals specifically with the adaptation concern, is added to this non-adaptive code, resulting in a self-adaptive application able to reconfigure its architecture and parameters to always fit its evolving environment. Adaptation logic which is reflexive by nature, is the code that monitors a representation of the system environment and internal state and then adapts that representation resulting in a reconfiguration of the actual system.

2.2 Conditional expression / Rule based

As mentioned in [34], the spectrum to express self-adaptability is broad. At one extreme (bottom) lie conditional expressions in the form of `If condition/then action` rules: the program evaluates an expression and alters its behavior based on the outcome. In its principle, conditional expressions combine the adaptation specification with the application specification. In its simplest form, this methodology is not very flexible (selects among predetermined alternatives), only supports localized changes because it would be difficult to unwind the control loop in a synchronized manner on all – possibly remote – software entities that are concerned at the same time, and lacks separation of concerns. That's why research on distributed rule agents coordinated by a rule engine are conducted (e.g. [28, 35]) to feature both dynamicity in rule injection and distributed rule-based complex adaptation scenarios. In DIOS++ [28], the rule engine receives at runtime rule requests from the user. It dynamically creates rule agents to manage objects if such agents don't exist yet. It then composes a script for each agent, which defines priorities. The rule engine is in charge of synchronizing all the rule agents.

2.3 Aspect Oriented Programming

Aspect oriented programming [26] has been designed to allow separation of concerns in the design of an application; with aspects one can design separately the functional and the non-functional concerns related to the system to be deployed. Embedding "autonomic rules" as aspects allow a better re-usability of the programs: different aspects can be designed corresponding to different deployment environments and can be freely composed with different business applications.

Concerning dynamic evolution however, pure aspect-oriented approach is still quite limited. Indeed, aspects are usually weaved within the functional code at compilation or instantiation time, which prevents this non-functional code from being modified at runtime. One has to weave again the adaptation code when deploying on a new infrastructure. Of course, adaptation code can be encapsulated inside an object which could be updated dynamically. This is what component models suggest: better specify the interfaces of objects in order to allow a better re-usability and dynamic evolution of the application. Such restriction w.r.t. runtime weaving motivated the introduction of Dynamic Aspect-Oriented languages, and in particular their application to autonomous systems [21], [18]. As noticed in [32] aspect orientation applied to autonomic systems constitutes a step in the right direction, but is by essence useful for tackling crosscutting concerns; whereas in the general case, adaptation may re-

quire to also modify or replace code of the application, be it devoted to functional or non-functional levels.

2.4 Autonomic Distributed Components

Component-based development has emerged as an effective approach to building complex software systems; its benefits include reduced development costs through reusing off-the-self components and increased adaptability through adding, removing, or replacing components. This is why component programming frameworks are becoming attractive in networking [5], in large-scale distributed computing (a.k.a Grid computing) be it dedicated to scientific computing [9] or enterprise computing [10], in mobile and situated autonomic communications [29], and more generally in any running context constituted of fixed or intermittently connected devices. In fact, we have reached a level of complexity, heterogeneity, and dynamism and consequently increasingly variable execution contexts, that standard programming environments and infrastructures, including component-oriented ones, have trouble to manage. This raised the necessity for adding to component models some concepts to ease the management of the distribution of the various software modules that constitute the ultimate application (for instance, by proposing that an aggregation of possibly remote components can be manipulated as a single, even if distributed, software entity, i.e. as a distributed component [8]), and also to easily plug self-* properties to those software entities. In general, the followed approach consists in wrapping around each component an autonomic manager (as the Element manager wrapping a computational element in Automate [35], or as the Component Application Manager defined in ASSIST [4]) that is guided by the interpretation of some rules or contracts ([28, 3, 33, 18]) dependent of the requested self-properties (e.g. self-healing, self-protecting, self-optimizing, self-configuring ...) and possibly injected at runtime. The actions that those rules trigger may encompass reconfiguring the component-based application. All these actions pertain to monitoring the base application and at the same time, take into account non-functional concerns due in particular to the running context. As a consequence, the monitoring itself may become so complex to express that its design and programming may be eased if relying on the use of a general-purpose structured programming language, and this is our opinion, by following a component-based approach (see next section for further details).

Besides, it appears that component and aspect-oriented approaches can complement each other very well (e.g. [33, 18, 37]), because at some point, non-functional concerns may be dependent or have impact to functional ones (e.g. update the variable that represents a component instance reference to which service invocations are triggered by the functional code). So techniques from AOP can be relevant to be used. For example, an aspect-oriented implementation of a component model like AOKell [36] allows to design separately the functional and the non-functional concerns related to a component system. More precisely, non-functional concerns are first expressed as components that can be freely composed as needed, and second, they present themselves as aspects, so that they are transparently integrated within the content (functional level) of a component by the AOKell mechanics. Moreover, the system is open, that is non-functional concerns like complex autonomic behaviours can be added as needed (just create and compose new non-functional components): this allows a better re-usability because different aspects can be designed corresponding to the various facets of the different deployment environments and chosen appropriately when building a new version of the component system. Nevertheless, AOKell implements a non-distributed version of the Fractal [12] component model.

3. STRUCTURING NON-FUNCTIONAL CONCERNS WITH COMPONENTS

An application can be split between some functional code implementing the business features, and some non-functional code for managing the application, and supporting its execution. We focus in this section on the Fractal component model and on its Grid extension : the GCM (Grid Component Model) [15]. They both feature separation of concerns and hierarchical composition of components. GCM supports better structured design of components management.

In Fractal and GCM component models, the non-functional (NF) part of the components is called the *membrane*. It is composed of *controllers* that implement non-functional concerns. During their execution, components running in dynamically changing execution environments often have to adapt to these environments. The membrane of Fractal/GCM components is the adequate location to host adaptation strategies, which in theory can be as complex as needed, i.e. completely autonomic. When the behavior of a strategy is not optimal, it has to be updated or changed dynamically. To do this, the controllers architecture has to support reconfigurations at runtime, which is the case of a component system. Examples of use-cases include changing communication protocols, updating security policies or taking into account new runtime environments in case of mobile components. Adaptability and autonomicity imply that evolutions of the execution environments have to be detected and acted upon. They may of course imply interactions with the environment but also with other components for achieving management strategies.

The Grid Component Model, as some other research works around a component-oriented framework for the programming of autonomic and distributed applications [3, 33, 35], features support for a clean separation of concerns between functional and non-functional ones. The peculiarity of Fractal/GCM is that both functional and non-functional parts can be designed as component systems, possibly distributed. Consequently, adaptation and complex autonomic behaviours can be designed and dynamically reconfigured in a component-oriented way.

In this section we focus on the structure of the membrane in the Fractal and GCM component models cited above. This structure, and in particular the one adopted in GCM, is a good starting point for plugging adaptation and autonomic strategies inside the membrane. This structure is also well suited for dynamic reconfigurations.

3.1 Fractal Component Model

Fractal [12] is a hierarchical component model: each component is either *composite* (i.e., composed of other components), or *primitive* (i.e., a black box encapsulating a basic functional block). Figure 1 shows a standard Fractal composite component composed of two other components (which are inside the *functional content*). *Server interfaces* are represented on the left of a component box, *client ones*

are represented on the right; arrows are *bindings* between interfaces, that will carry messages. In standard Fractal, non-functional aspects are exposed as *controller interfaces* (on the top of the component box). Fractal controllers deal with non-functional aspects of the component, they are included inside the *membrane*. The way the controllers are implemented is left unspecified.

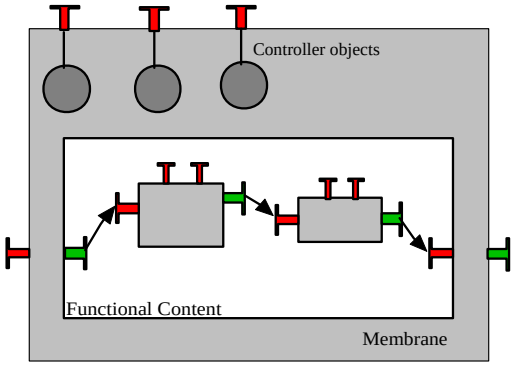

Figure 1: A standard Fractal component

3.1.1 Non-Functional support in Fractal

Fractal already features some separation of concerns: in the base model, the functional part of the application can be designed as a component system or a primitive functional block. The basic component management (configuration, life-cycle) is specified as a set of *controllers* that each component can include inside its membrane. Such controllers are generally provided by the component platform, but as the Fractal component model is extensible, new controllers can be added to implement additional management features. All the controllers are gathered in the component membrane.

In the basic Fractal component model, there is a set of well known controllers: *BindingController* for changing the bindings (connections with other components) of a component, *ContentController* for managing the functional content of a component (for example adding or removing a component), *LifeCycleController* for managing the life cycle of the component (for example starting or stopping a component) and *AttributeController* for changing the attributes of a component.

3.1.2 Componentizing Membranes in Fractal

Unfortunately, in the Fractal component model composition of non-functional aspects is left unspecified. Each controller corresponds to a NF interface that can be discovered (introspected) but not bound to. NF invocations are only direct invocations on a NF interface. Consequently, in the first Fractal implementations, controllers were implemented as a set of objects. This approach had several limitations for management strategies. First, dynamic reconfiguration of object controllers is very difficult because the architecture is designed to be static. And as client NF interfaces were not supported, composition of NF aspects was impossible. Several solutions have been proposed to overcome such limitations [31, 37]. Among them, the most conservative w.r.t. Fractal is probably *AOKell*, allowing to design component membranes as composite components. The main advantage of this approach is its minor increment to the Fractal component model. Membranes are implemented as composite components and are injected inside functional components (that we call host components). Connecting membranes means getting a reference on those composite components and binding them together. The client non functional interfaces of a host component belong to an inner component (the composite representing the membrane). Though effective, this solution breaks the encapsulation of components which may threaten their re-usability and management as highlighted in [27].

3.2 GCM Component Model

As the GCM is a component model for Grid computing based on Fractal, it inherits hierarchy, introspection and the basic controllers from Fractal, and extends it using asynchronous method calls for dealing with possibly high latency. As this model is targeting the Grid, GCM components are distributed by essence. They are units of composition and deployment. A GCM component system can be composed of a set of components deployed on several machines. A composite GCM component is a distributed logical entity that can be composed of several components spread over different hosts.

The GCM also defines collective interfaces which ease design and implementation of multiple parallel components, that can be targeted in a collective manner: a client interface may be a multicast interface, meaning that a call towards this interface can be distributed to many server interfaces depending on the distribution used. Similarly, a server interface may be a gathercast interface, meaning that multiple client calls will be synchronised and a single call will be performed towards the service component.

Coming back to NF concerns, in the GCM we want to provide tools to plug and dynamically reconfigure autonomic strategies inside the membrane. This means that these tools have to manage (re)configuration of controllers inside the membrane and the interactions of the membrane with membranes of other components. For this, we provide a model and an implementation, using a standard component-oriented approach for both the application (functional) level and the control (NF) level. Having a component-oriented approach for the non-functional aspects also allows them to benefit from the structure, hierarchy and encapsulation provided by a component-oriented approach. This has already been adopted or advocated in [36, 31, 22].

The solution that is suggested by the GCM is to allow, like in [36, 31], to design the membrane as a set of components that can be reconfigured. The GCM description[15] suggests the possibility to implement the membrane as a set of components. [7] goes more into details and suggests a structure for the composition of the membrane and an API for manipulating it. In GCM, non-functional components that are included inside the membrane can be distributed, just like the functional ones. Note that it does not seem reasonable to implement, like in AOKell, the membrane as a composite component because of the distributed nature of components: a composite GCM component would in general involve a much higher overhead than a Fractal one, and crossing it systematically in order to access a non-functional feature might be needlessly costly.

In order to be able to compose non-functional aspects, the GCM requires the NF interfaces to share the same specification as the functional ones: role, cardinality, and contingency. For example, comparatively to Fractal, the GCM

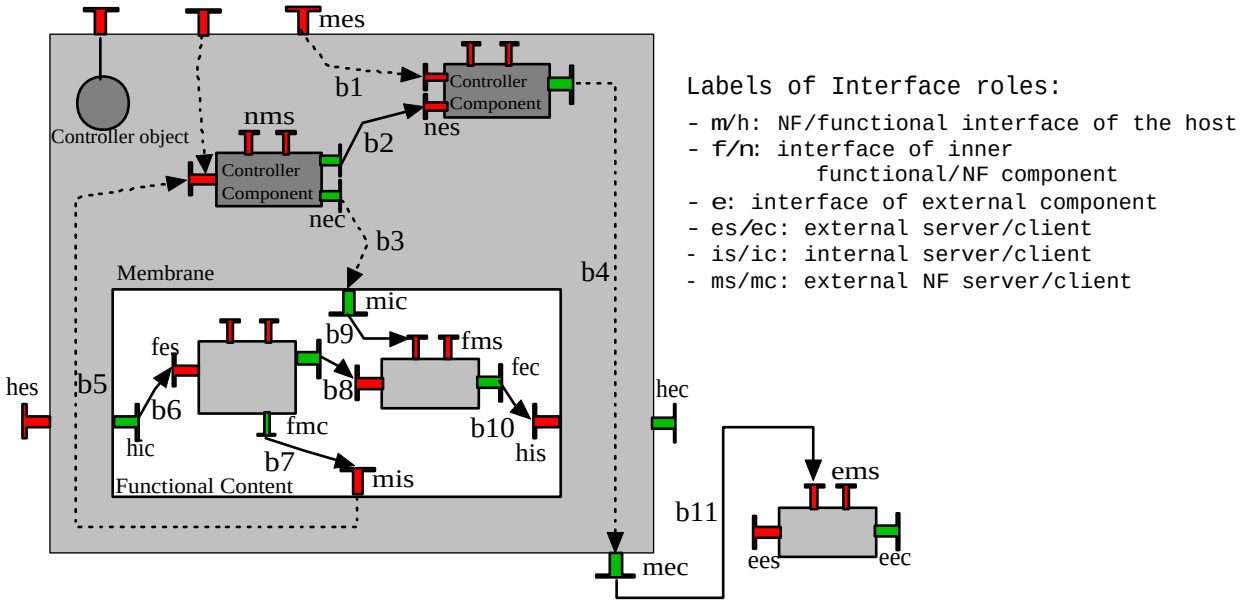

Figure 2: New structure for the membrane of Fractal/GCM components

adds *client non-functional interfaces* to allow for the composition of non-functional aspects and reconfigurations at the non-functional level.

The following of this section provides details about the structure a membrane can adopt. Figure 2 represents the structure of a membrane and gives a summary of the different kinds of interface roles and bindings a GCM component can provide. Here we give a more detailed explanation about possible roles of NF interfaces exposed by the membrane, making references to Figure 2:

- *mes*: *external server interfaces* as exist in the Fractal specification; they allow external entities to access controllers inside the membrane;
- *mec*: *external client interfaces*; they give to inner NF components access to external NF interfaces;
- *mis*: *internal server interfaces*; they give inner functional components access to interfaces of inner NF components;
- *mic*: *internal client interfaces*; they give to inner NF components access to interfaces of inner functional components.

All these interfaces give the membrane a better structure and enforce decoupling between the membrane and its externals. For example, to connect *nec* with *fms*, our model adds an additional stage: we have first to perform binding *b3*, and then binding *b9*. This avoids *nec* to be strongly coupled with *mic*: to connect *nec* to another *mic*, only binding *b9* has to be changed.

In Figure 2, some of the links are represented with dashed arrows. Those links are not real bindings but "alias" bindings (e.g. b3); the source interface is the alias and it is "merged" with the destination interface. These bindings are similar to the export/import bindings existing in Fractal (b6, b10) except that no interception of the communications on these bindings is allowed. To conclude, the GCM provides possibilities consisting in implementing as components (part of) the membrane and thus benefiting from the component structure, which allow membrane to be designed in a component-oriented way, and to be reconfigurable. The model is also flexible, as all server NF interfaces can be implemented by both objects or components controllers.

4. PLUGGING AUTONOMIC BEHAVIOURS INSIDE COMPONENTS

We focus now on autonomic behaviours, as a particular non-functional concern. We showed in the preceding section that in the GCM this aspect can be programmed as a component system which is itself evolutive.

Existing Autonomic Components.

The GCM can encompass two kinds of autonomic behaviours: the one consisting in (autonomously) adapting a component to its changing environment; and the one consisting in (autonomously) adapting the components to evolving user-requirements for the applications (generally concerning the quality of services). The second approach has been widely addressed for behavioural skeletons [2, 3]; this approach is able to let component autonomously adapt to a required quality of services, by adapting the quantity and quality of resources allocated for a given task. The authors actually design the autonomous component as functional ones because non-functional components were not yet available in the preliminary version of the GCM but their experiences show that this approach will strongly benefit from a componentization of the membrane. Similar works can be found also in [6].

Autonomic Components = GCM Components + Autonomic Managers Inside Membranes.

Because of the existing experiments and its capabilities, we consider thus that the component model presented in Section 3.2 is particularly adapted to the programming of non-functional concerns, especially in a distributed environment.

Our proposal for autonomic control of components is to encapsulate a set of manager components inside the membrane of each component. These components can be easily changed dynamically depending on the environment or the evolution of the component system. Interface and typing of components allow those autonomic managers to be bound together in a very structured way, which enables the distribution of the decision process: each autonomic manager can be responsible of the management of its component, and still communicate with other managers or with the external world if necessary. Moreover, hierarchical structure of components help managing autonomously a set of components and scales better: a decision process can be taken by delegating sub-decisions to sub-components of a composite one.

Our proposal just consists in a structure for plugging autonomic behaviours: it is not our purpose here to define new algorithm and decision processes for autonomicity, but rather to provide a framework in which autonomic behaviours can easily be programmed and integrated.

The advantage of this structure is that it can easily integrate most of the programming methodologies for autonomicity presented in section 2. For example a particular autonomic manager can in fact be a rule interpreter, which is sent rules and is able to trigger actions (e.g., reconfiguration of the component system) autonomously according to the rules. Autonomic behaviours (defined by the autonomic manager components) can be weaved inside business code using aspect methodology, like in AOKell. But most importantly, this design allows a better dynamic evolution and code re-use for autonomicity, making our platform a particularly adapted environment for the experimentation of autonomic strategies. Indeed, autonomic GCM components can be distributed on the Grid and consequently experimented with on a large scale basis, for example using the GCM/ProActive component framework (an implementation of the GCM model over the ProActive[13] Grid middleware).

We showed how the GCM component framework allows to implement autonomous components with self-adaptation capabilities, and self improvement of their ability to better fulfill the required service and quality. But, in a distributed and possibly low coupled environment, this task becomes more complex because the lack of reliability of the links between components entails a possible degradation of the quality of service achieved by the system; and more importantly, a possible loss of some of the functionality or services the system relies on. Next section shows how the GCM can be adapted in a lighter notion of component composition to reason about autonomic component systems in a loosely coupled environment.

5. AUTONOMIC DISTRIBUTED SERVICES

We presented a framework supporting highly autonomic strategies for both composition of services and adaptation to evolving requirements. But autonomic computation goes beyond the autonomic composition and improvement through evolution of services. Indeed, the next step is to consider a network with low connectivity where entities involved in the computation can be disconnected at any time. This demanding context underlines the limitations of the classical component design of applications: components composed together are assumed to correspond to runtime entities that are to be involved, and so present, during a computation. Self-repairing component frameworks exist, allowing when a component disappears to create a new one (with possibly the old status of the previous one) at a new place in the network. In usual networks featuring a lot of resources, a relatively low failure rate, and enough structure to allow some form of centralized decision to be taken, this solution is probably the safest and the most efficient.

Consequences of Low Connectivity.

However, when one switches to distributed computing environments featuring low connectivity as those relying on a mobile ad-hoc network, this is not sufficient. Indeed, first disappearing nodes (and consequently, disappearing services that were running on those nodes) can appear afterwards. Moreover, the network is not structured enough to allow the creation of a unique new entity replacing the missing one easily. In this section, we release the connectivity constraint, and replace it by additional new constraints fitting more to a self-organized distributed environment:

- Any binding can disappear at any time, sometimes reappearing later on.
- When a service is needed, this one can be discovered at runtime according to a description of the desired service, by the medium (which can be the sender of a request, or some other entities involved in the network),
- Some services may appear, for example fitting a given description (as a side note, notice that it is not the purpose of this paper to propose solutions to decide whether a service should be created or looked for).

Better Decoupling Communications.

A first aspect to consider in applications resulting from the composition of distributed services or components is the way communications are performed. A lot of work has been performed concerning communication protocols and middleware in the context of ad-hoc networks but as highlighted by [16, 19], only few efforts have been devoted to programming models for applications that must take for granted that they rely on such a low-coupled and transient communication mode. In [16, 19], the authors suggest a communication model, called *AmbientTalk*, based on an actor-like language but with several queues for both sending and receiving messages whenever possible, i.e. whenever connected. Their communication model is quite similar to the ASP calculus presented in [14] that we are currently revisiting by removing the assumption of establishing a "rendez-vous", i.e. a handshake, in the transmission of each service request or reply. This reference to ASP is interesting because ASP defines the basic communication mode that the GCM relies upon for transmitting service messages among bound components. However, their resulting programming model is slightly different because there is no blocking synchronization in AmbientTalk, whereas ASP relies on a more coupled programming principle allowing to write programs more deadlocks prone but in a more intuitive manner (similar to a classical sequential program instead of relying on events and handlers). To sum up, it should be easy to turn GCM into a programming model resilient to disconnected communication modes, taking profit of ideas introduced in *AmbientTalk* like delaying service request or reply transmissions.

Also, the service publication and discovery machinery that is intrinsic in AmbiantTalk because it is required to dynamically get references to service instances could very easily be represented as a non-functional – and so evolvable – concern accessible through the membrane of GCM components.

Composition Plan: A composition paradigm releasing the strong connectivity constraint of components.

No matter which programming model is chosen, we argue that, at the composition level some features have to be provided to better compose the basic blocks, resulting in a low-coupled application. Indeed, programming low-coupled applications easily results in a lot of code for dealing with the coupling process itself because this process is dynamic due to the possible volatility of those basic blocks, making difficult and error-prone the basic functional programming. Our objective is to be able to propose a composition method for such applications following a component-like approach : taking advantage of its qualities concerning structure of the program and high re-usability of the components; but authorizing low-coupling in the sense that some of the components may disappear during the process, and (if it is compatible with the purpose of the application) some other components fulfilling the same service may be found to replace them dynamically.

From the observation that component programming is very convenient for composing applications but entails too much coupled interactions, we nevertheless choose to rely on components but in the following way: components are composed in an abstract way and so provide a structure for the application that we can consider as an execution plan (a specification), which is structured because it is hierarchical; but we require a strong separation between application specification and implementation, allowing at runtime entities fulfilling the component description to be discovered, and to disappear. In other words, the application is designed by a component composition, but at runtime the services appear and disappear and must be bound in a much more dynamic way compared to what component models usually support. Fortunately, in our case, GCM natively supports dynamic reconfiguration, and these reconfigurations can be triggered dynamically by dedicated autonomous controllers. Finally, the application does not have to deal explicitly with disappearing services.

As services appear and disappear, a definition of a component involved in a composition may be too precise, i.e. too constraining. Consequently, the framework must be able to discover services based on a partial description, or even replace a service whose description does not correspond to any available service by a different service resulting from a dynamic (possibly hierarchical) composition of some more elementary but currently available services. Because it supports reconfigurable and adaptive components, but also for its structured programming model, Fractal/GCM provides an adequate environment for autonomic services.

The stability of composition is completely lost in such a world, but this is partially compensated by the stability of the execution plan: *the execution plan never disappears.*

To summarize, the *composition plan* is a stable plan, and can be considered as the *specification of a service.* The composition plan is different from the instance of the composition which can evolve depending on the available services or entities, i.e. this instance can more or less fit the specification. In case the evolved instance appears to better fulfill the initial service specification as indicated by the plan, we may envision to backup the resulting new architecture, i.e. to save this architecture and use it as a starting plan in the future.

To conclude, this new notion of component composition is particularly adapted to loosely coupled systems, because it does not rely on stable bindings between composed entities. For example it would be particularly well fitted to the design of a component model for Ambient-oriented programming.

How to Get a Composition Plan.

A major question remaining is how this composition plan can be obtained; several approaches can be envisioned to create this plan. They range from an ADL-like [30], hand-made composition which is provided by an application composer; to semantically driven automatic composition of services in order to achieve a user-defined goal given in a more or less natural language [20].

A realistic compromise relies on an initial state provided by a programmer, and describing a set of services and composition. Then the system evolves by itself, possibly relying on some bio-inspired evolution strategy, and on the user's feedback and requirements; this step would strongly benefit from a semantic description of the services allowing to gear the evolution toward the desired goals [24, 23].

6. CONCLUSION: TOWARDS BIO-INSPIRED SERVICE COMPOSITION MODELS

We believe that the programming model and the framework we presented in this paper is particularly adapted to experiment with bio-inspired or nature-inspired models for autonomicity and self-organisation, especially autonomic composition [11], for the following reasons:

- Autonomic GCM components will be very convenient for decentralized decision systems; indeed, independent components scattered on a Grid will help emulating independent individuals with their own activity and governance rules scattered on any kind of loosely coupled network.
- The composition plan for specifying component systems allows to program and deploy loosely coupled components, where few hypothesis on stability and on communication are made.
- GCM relies on ProActive, an implementation of a still very intuitive and powerful programming model, based on object-based asynchronous activities, particularly adapted to autonomous entities (indeed, activities communicating by asynchronous requests and replies fit much better to autonomous entities than multi-threaded objects communicating via synchronous remote method invocations).
- Thanks to components inside the membrane that are able to encapsulate bio-inspired evolution mechanisms, the autonomicity algorithm, and thus the biology-inspired strategy itself can be modified, even at runtime.
- Because of decentralization and of pluggable autonomic managers, one can experiment with the coexistence

and interaction of two species with two different evolution mechanism.

On the long term, this framework will be very convenient to experiment with a set of predefined bio-inspired components, and see how they fit with newly defined business components. This would allow emulation and experimentation on a wide range of bio-inspired communication and computation mechanism over many different applications.

Acknowledgements.

This work has received the financial support of the Conseil régional Provence-Alpes-Côte d'Azur, and of the IST FET funded IP BIONETS project.

7. REFERENCES

[1] M. Aldinucci, C. Bertolli, S. Campa, M. Coppola, M. Vanneschi, and C. Zoccolo. Autonomic Grid Components: the GCM Proposal and Self-optimising ASSIST Components. In *Joint Workshop on HPC Grid programming Environments and Components and Component and Framework Technology in High-Performance and Scientific Computing at HPDC'15*, June 2006.

[2] M. Aldinucci, S. Campa, M. Danelutto, P. Dazzi, D. Laforenza, N. Tonellotto, and P. Kilpatrick. Behavioural Skeletons for Component Autonomic Management on Grids. Technical Report TR-80, CoreGRID, 2007. Proceedings of the CoreGRID Workshop, Heraklion, June 12–13.

[3] M. Aldinucci, M. Danelutto, and M. Vanneschi. Autonomic QoS in ASSIST Grid-Aware Components. In *PDP '06: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.

[4] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In *11th Intl Euro-Par 2005: Parallel and Distributed Computing*, Lisboa, Portugal, Aug. 2005.

[5] H. Algestam, M. Offesson, and L. Lundberg. Using Components to Increase Maintainability in a Large Telecommunication System. In *Ninth Asia-Pacific Software Engineering Conference (APSEC'02)*.

[6] F. André, H. L. Bouziane, J. Buisson, J.-L. Pazat, and C. Pérez. Towards dynamic adaptability support for the master-worker paradigm in component based applications. In *CoreGRID Symposium in conjunction with Euro-Par 2007 conference*, Rennes, France, 27-28 August 2007. to appear.

[7] F. Baude, D. Caromel, L. Henrio, and P. Naoumenko. A flexible model and implementation of component controllers. In *CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments*, 2007.

[8] F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA)*, number 2888 in LNCS, pages 1226–1242, Catania, Sicily, Italy, 3-7 November, 2003. Springer Verlag.

[9] D. Bernholdt, B. Allan, R. Armstrong, F. Bertrand, and K. Chiu. A component architecture for high performance scientific computing. *ACTS Collection special issue, Intl. J. High-Perf. Computing Applications*, 20, 2006.

[10] S. Bouchenak, F. Boyer, E. Cecchet, S. Jean, A. Schmitt, and J.-B. Stefani. A component-based approach to distributed system management - a use case with self-manageable J2EE clusters. In *11th ACM SIGOPS European Workshop, September 2004*.

[11] G. Briscoe and P. D. Wilde. Digital ecosystems: Evolving service-orientated architectures. In *First IEEE International Conference on Bio Inspired mOdels of NETwork, Information and Computing Systems (BIONETICS), 2006*.

[12] ï. Bruneton, T. Coupaye, M. Leclercq, V. Quï£¡ma, and J.-B. Stefani. The fractal component model and its support in java. *Software Practice and Experience (SPE), special issue on "Experiences with Auto-adaptive and Reconfigurable Systems*, 36, 2006.

[13] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton. ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology*, 12(1):69–77, 2006.

[14] D. Caromel and L. Henrio. *A Theory of Distributed Objects*. Springer-Verlag New York, Inc., 2005.

[15] CoreGRID Programming Model Virtual Institute. Basic features of the grid component model (assessed), 2006. Deliverable D.PM.04, CoreGRID, Programming Model Institute.

[16] T. V. Cutsem, J. Dedecker, and W. D. Meuter. Object-oriented coordination in mobile ad hoc networks. In A. L. Murphy and J. Vitek, editors, *COORDINATION*, volume 4467 of *LNCS*, pages 231–248. Springer, 2007.

[17] P. David and T. Ledoux. Towards a framework for self-adaptive component-based applications. In *Proceedings of Distributed Applications and Interoperable Systems 2003, the 4th IFIP WG6.1 International Conference, DAIS 2003*, volume 2893 of *LNCS*, pages 1–14. Springer-Verlag, 2003.

[18] P. David and T. Ledoux. An aspect-oriented approach for developing self-adaptive fractal components. In *Proceedings of 5th International Symposium, SC 2006, sattelite event of ETAPS*, volume 4089 of *LNCS*, pages 82–97. Springer-Verlag, 2006.

[19] J. Dedecker, T. V. Cutsem, S. Mostinckx, T. D'Hondt, and W. D. Meuter. Ambient-oriented programming in ambienttalk. In D. Thomas, editor, *ECOOP*, volume 4067 of *LNCS*, pages 230–254. Springer, 2006.

[20] K. Fujii and T. Suda. Semantics-based dynamic service composition. *IEEE Journal on Selected Areas in Communications (JSAC), special issue on Autonomic Communication Systems*, 23(12), 2005.

[21] P. Greenwood and L. Blair. Using Dynamic Aspect-Oriented Programming to Implement an Autonomic System. In *Proceedings of Dynamic Aspects Workshop (DAW) (held with AOSD 2004)*. Published as Research Institute for Advanced Computer Science (RIACS) Technical Report 04.01., Lancaster, UK, March 2004.

[22] C. Herault, S. Nemchenko, and S. Lecomte. A Component-Based Transactional Service, Including Advanced Transactional Models. In *Advanced Distributed Systems: 5th International School and Symposium, ISSADS 2005, Revised Selected Papers*, number 3563 in LNCS, 2005.

[23] C. Jacob, D. Linner, H. Pfeffer, and I. Radusc. Bio-inspired processing and propagation of semantics in loosely coupled computing environments. *International Journal of Semantic Computing (IJSC)*, 1(1):121–144, 2007.

[24] J. Keeney, K. Carey, D. Lewis, D. O. Sullivan, and V. Wade. Ontology-based semantics for composable autonomic elements. In *Workshop on AI in Autonomic Communications at 19th International Joint Conference on Artificial Intelligence, 2005*.

[25] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1), 2003.

[26] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 1997.

[27] M. Léger, T. Coupaye, and T. Ledoux. Contrôle dynamique de l'intégrité des communications dans les architectures à composants. In S. V. R. Rousseau, C. Urtado, editor, *Langages et Modèles à Objets*, pages 21–36. Hermès-Lavoisier, 2006.

[28] H. Liu and M. Parashar. DIOS++: A Framework for Rule-Based Autonomic Management of Distributed Scientific Applications. In *9th Int. Euro-Par Conference (EuroPar 2003)*, LNCS.

[29] P. Marrow and A. Manzalini. The CASCADAS Project: a Vision of Autonomic Self-organising Component-ware for ICT Services. In *International Conference on Self-Organization and Autonomous Systems in Computing and Communications (SOAS 2006)*.

[30] N. Medvidonic and R. Taylor. A classification and comparison framework for Software Architecture Description Languages. *IEEE Trans .on Software Engineering*, 26(1), 2000.

[31] V. Mencl and T. Bures. Microcomponent-Based Component Controllers: A Foundation for Component Aspects. In *Proceedings of 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, pages 729–737. IEEE Computer Society Press.

[32] A. Mukhija and M. Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *18th International Conference on Architecture of Computing Systems (ARCS 2005)*, pages 124–138.

[33] A. Mukhija and M. Glinz. The CASA Approach to Autonomic Applications. In *Proceedings of the 5th IEEE Workshop on Applications and Services in Wireless Networks (ASWN 2005)*, pages 173–182.

[34] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, may/june 1999.

[35] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. Automate: Enabling autonomic grid applications. *Cluster Computing: The Journal of Networks, Software Tools, and Applications, Special Issue on Autonomic Computing*, 9(2), 2006.

[36] L. Seinturier, N. Pessemier, and T. Coupaye. AOKell: an Aspect-Oriented Implementation of the Fractal Specifications, 2005. `http://www.lifl.fr/~seinturi/aokell/javadoc/overview.html`.

[37] L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A Component Model Engineered with Components and Aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering*, 2006.