# Flexible Adaptation Loop for Component-based SOA Applications

Cristian Ruz, Françoise Baude, Bastien Sauvan
*INRIA Sophia Antipolis Méditerranée*
*CNRS, I3S, Université de Nice Sophia Antipolis*
*France*
{*cruz,fbaude,bsauvan*}*@inria.fr*

***Abstract*—The Service Oriented Architecture (SOA) model fosters dynamic interactions of heteregeneous and loosely-coupled service providers and consumers. Specifications like the Service Component Architecture (SCA) have been used to tackle the complexity of developing such dynamic applications; however, concerns like runtime management and adaptation are often left as platform specific matters. At the same time, runtime QoS requirements stated in Service Level Agreements (SLA) may also evolve at runtime, and not only the application needs to adapt to them, but also the monitoring and management tasks. This work presents a component based framework that provides flexible monitoring and management tasks and allows to introduce adaptivity to component-based SOA applications. The framework implements each phase of the autonomic control loop as a separate component, and allows multiple implementations on each phase, giving enough runtime flexibility to support evolving non functional requirements on the application. We present an illustrative scenario that is dynamically augmented with components to tackle non-functional concerns and support adaptation as it is needed. We use an SCA compliant platform that allows distribution and architectural reconfiguration of components.**

***Keywords*-Monitoring; Management; SLA Monitoring; Reconfiguration; Component-based Software Engineering.**

## I. Introduction

According to the principles of Service Oriented Architecture (SOA), applications built using this model comprise loosely-coupled services that may come from different heterogeneous providers. At the same time, a provided service may be composed of, and consume other services, in a situation where service providers are also consumers. Moreover, SOA principles like abstraction, loosely coupling and reusability foster dynamicity, and applications should be able to dynamically replace a service in a composition, or adapt the composition to meet certain imposed requirements.

Requirements over service based applications usually include metrics about Quality of Service (QoS) like availability, latency, response time, price, energy consumption, and others, and are expressed as Service Level Objectives (SLO) terms in a contract between the service consumer and the provider, called Service Level Agreement (SLA). However, SLAs are also subject to evolution due to different providers, environmental changes, failures, unavailabilities, or other situations that cannot be foreseen at design time. The complexity of managing changes under such dynamic requirements is a major task that pushes the need for flexible and self-adaptable approaches for service composition. Self-adaptability requires monitoring and management features that are transversal to most (all) of the involved heterogeneous services, and may need to be implemented in different ways for each one.

Several approaches have been proposed for tackling the complexity, dynamicity, heterogeneity and loosely-coupling of SOA-based compositions. Notably, the Service Component Architecture (SCA) is a technologically agnostic specification that brings features from Component-Based Software Engineering (CBSE) like abstraction and composability to ease the construction of complex SOA applications. Non-functional concerns can be attached using the SCA Policy Framework. However, concrete monitoring and management tasks are usually left out of the specifications and must be handled by SCA platform implementations, mainly because SCA is design-time and not runtime focused.

Our thesis is that a component-based approach can ease the implementation of flexible adaptations in component-based service-oriented applications. Our proposed solution implements the different phases of the widely used MAPE (Monitor, Analyze, Plan, and Execute) autonomic control loop as separate components that can interact and support multiple sets of monitoring sources, conditions, strategies and distributed actions.

The rest of the paper is organized as follows. Section II presents an example situation that motivates our work and provides a general overview of our contribution. Section III describes the design of our framework from a technologically independent point of view. Section IV presents our implementation over a concrete middleware. Section V describes related work and differentiations with our solution. Finally, Section VI concludes the paper.

## II. Motivating example and Overview of Our Contribution

Consider a tourism office who has composed a smart service to assist visitors who request information from the city and provides suggestions of activities and touristic planning. The application uses a local database of touristic events and a set of attraction providers who sell tickets to parks, tours, etc. A weather service guides the proposition

ISBN: 978-1-61208-134-2

of activities, and a mapping service creates a map with directions. A payment service may be needed in some cases. Once all information is gathered, a local engine composes a PDF document and optionally prints it. The composed design of the application is show in Figure 1.

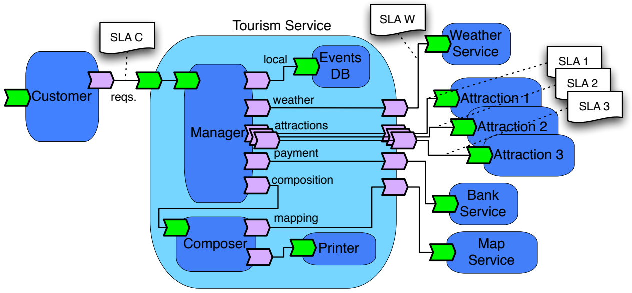

Figure 1. Scenario. SCA description of the application for tourism planning.

Such a composition involves some contracts for service provisioning established in SLAs. For example, the Tourism Service should propose a plan within 30 sec.; the Weather Service charges a fee for each forecast; the Mapping Service is free but has no guarantees on response time or availability; the Banking Service ensures 99% of availability.

The runtime compliance to the SLAs may determine later decisions of the service. For instance, some of the Attraction Services may take too much time to deliver an answer, or the Mapping Service may not be available or have a poor performance at some moment. Situations like those may require that a runtime decision be taken to avoid violating the SLA with the customer. For instance, taking actions like the removal or replacement of a slow provider; or changing some parameter on the composer to ignore the map while composing the document. To be able to make such kind of decisions, a precise and efficient runtime monitoring and SLA compliance system is required.

The monitoring requirements may be different for each service; for example, in the case of the printer it is important to measure the amount of paper or ink; in the case of the composer it is important to know the time it takes to create a document; some of the external services may provide their own monitoring metrics and, as they are not locally hosted and only accesible through a predefined API, it may not be possible to add specific monitoring on their side. This situation imposes a requirement for supporting heterogeneous services and adaptable monitoring

As there are several external providers involved, the conditions expected from each one of them may change, and so the monitoring requirements over them. The Weather Service may decide to modify their charging plans; or some attractions may offer temporary promotions, which may influence the strategy to select them.

The composition of the application may also change. The composer service may decide to provide an alternative bluetooth service to transmit the composed document to a smartphone, less costly than the printer service. In that case, a new component must be added to the composition, and the monitoring and management infrastructure must be changed accordingly.

### A. Concerns

As it can be seen from the example, concerns about SLA and QoS can be manifold. A monitoring system may be interested in indicators like performance, energy consumption, price, robustness, security, availability, etc., and the set of required values may be different for each monitored service. Plus, not only the values of these indicators change at runtime, but also the set of required indicators, as the monitoring requirements can also evolve. Moreover, due to the heterogeneous nature of the providers, some of the services may require specific protocols to retrieve monitoring values or to perform modifications on them.

In general, the evolution of the SLA and the required indicators can not be foreseen at design time, and it is not feasible to prepare a system where all possible monitorable conditions are ready to be monitored. Instead, it is desirable to have a flexible system where only the required set of monitoring metrics are inserted and the required conditions checked, but as the application evolves, new metrics and SLA conditions may be added and others removed minimizing the intrusion of the monitoring system in the application.

### B. Contribution

We argue that a component-based approach can tackle the dynamic monitoring and management requirements of a composed service application while also providing self-adaptivity. We propose a component-based framework to add flexible monitoring and management concerns to a running component-based application.

In this proposition we separate the concerns involved in a classical autonomic control loop (MAPE) and implement those concerns as separate components. These component are attached to each managed service, in order to provide a custom and composable monitoring and management framework. The framework allows to build distributed monitoring and management architectures that are associated to the actual functional components in an integrated way. Our framework leverages the monitoring and management features of each service to provide a common ground in which monitoring, SLA checking/analysis, decisions, and actions can be carried on by different components, and they can be added or replaced separately.

We believe that the dynamic inclusion and removal of monitoring and management concerns allows (1) to add only the needed monitoring operations, minimizing the overhead, and (2) to better adapt to evolving monitoring needs, without enforcing a redeployment and redesign of the application, and increasing separation of concerns.

## III. Design of the Component-Based Solution

Our solution relies on the separation of the phases of the classical MAPE autonomic control loop. Namely, we envision separate components for monitoring, analysis, planning, and execution of actions. These components are attached to each managed service.

The general structure of our design is shown for an individual service $C$ in Figure 2. Service $C$ is extended with one component for each phase of the MAPE loop and converted into a *Managed Service* $C$ (dashed lines). The small interfaces are aggregated by our framework to allow interaction with other managed components.

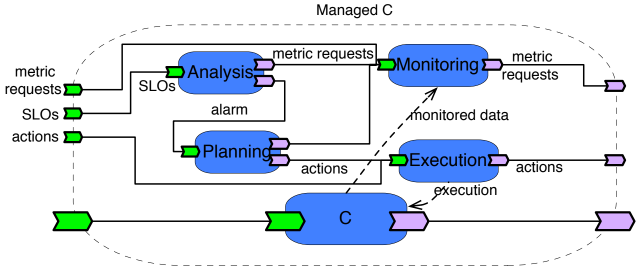

Figure 2. SCA component $C$ with all its attached monitoring and management components

The *Monitoring* component collects monitoring data from service $C$ using the specific means that $C$ may provide. From the collected monitoring data, the *Monitoring* component provides a set of *metrics* through an interface. The *Analysis* component provides an interface for receiving and storing SLOs and checks them at runtime using the metrics obtained from the *Monitoring* component. Whenever an SLO is not fulfilled, the *Analysis* component sends an alarm to the *Planning* component, which uses a strategy to create an adaptation plan, as a sequence of actions. The actions are executed by the *Execution* component, which includes the specific means to make them effective over service $C$, completing the loop.

Although simple, this component view of the autonomic control loop has several advantages. First, by separating the control loop from the component implementation, we obtain a clear separation of concerns between functional content and non-functional activities. Second, the component-based approach allows to have separate implementations for each phase of the loop; as each phase may require complex tasks, we abstract from their implementation, that may be specific for each service, and allow them to interact only through predefined interfaces, so that each phase may be implemented by different experts. Third, as each phase can be implemented independently, we allow to compose each phase to have possibly multiple components, for example, multiple sensors, condition evaluators, planning strategies, and connections to concrete effectors as required, so that the implementation of each phase can be adapted at runtime.

The framework allows to add and remove at runtime different components of the loop, which means that, for example, a service that does not need monitoring information extracted, does not need to have a Monitoring component and may only have an Execution component to modify some parameter of the service. Later, if needed, it is possible to add other components of the framework to this service.

In the following, we describe the components considered in the monitoring and management framework, their function and some design decisions that have been taken into account.

### A. Monitoring

The *Monitoring* task consists in collecting information from a service, and computing a set of indicators or *metrics* from it. The *Monitoring* component includes sensors specific for a service or, alternatively, supports the communication with sensors provided by the target service according to a particular protocol. This way, the *Monitoring* component is effectively attached to the service, which becomes a “monitored service”.

In the presence of a high number of services, the computing and storage of metrics can be a high-demanding task, specially if it is done in a centralized manner. Consequently, the monitoring task must be as decentralized and low-intrusive as possible. Our design considers one *Monitoring* component attached to each monitored service, that collects information from it, and exposes an interface to obtain the computed metrics. This approach is decentralized and specialized with respect to the monitored service. On the other side, some metrics may require additional information from other services: for example, to compute the cost of running a composition, the *Monitoring* component would require to know the cost of all the services used while serving some request. To address this situation, the *Monitoring* component is capable of connecting to the *Monitoring* components of other services. This way, the set of *Monitoring* components are inter-connected forming an architecture that reflects the composition of the monitored services and forming a “monitoring backbone” as shown in Figure 3. The metrics computed at each service can flow and can be used by another component.

### B. SLA Analysis

The *Analyzer* component checks the compliance to a previously defined SLA. An SLA is defined as a set of simpler terms called *Service Level Objective*s (SLOs), which are represented by conditions that must be verified at runtime. The conditions may be very simple ones, f.e., triples $\langle metric, comparator, value \rangle$ expressing, for instance, “$respTime \leq 30sec$”; or more complex expressions involving other metrics or operations on them like

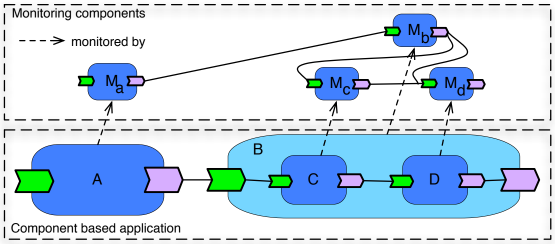

Figure 3. Monitoring layer for an SCA application

“$cost(weatherService) < 2 \times cost(mappingService)$”, where the metrics used by different services are required. The *Analyzer* obtains the values of the metrics it needs from the *Monitoring* component, as exemplified in Figure 4, and thanks to the interconnected *Monitoring* components, it can obtain metrics from other services as well.

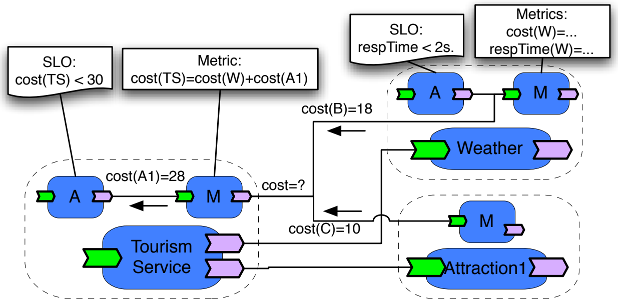

Figure 4. SCA Components with Analysis (A) and Monitor (M) components. *Tourism Service* and *Weather* have different SLAs. Metric *cost* is computed in *Tourism Service* by calling the monitors of *Weather* and *Attraction1*.

As input, the *Analyzer* receives a set of conditions (SLOs) to monitor, expressed in a predefined language. The *Analyzer* checks the compliance of all the stored SLOs according to the metrics obtained from the *Monitoring* component. The *Analyzer* checks if the SLA is being fulfilled, and if not, it sends an alarm notification through a client interface. The consequences of this alarm are out of the scope of the *Analyzer*. The *Analyzer* may also be configured in a proactive way to detect not only SLA violations, but also foreseeable SLA violations, which may be more useful in some contexts, as it can allow to take preventive actions [1].

By having the *SLA Analyzer* attached to each service, the conditions can be checked closely to the monitored service and benefit of the hierarchical composition. This way, the services do not need to take care of SLAs in which they are not involved.

### C. Planning

The *Planning* component contains the strategy defined for reacting to an alarm notified by the *Analyzer* component. The implemented logic can be a very simple strategy like changing the parameter of a service, or replacing one service for another service selected from a list; or a more complex strategy that requires collecting metrics of other components in order to select a subset of services that maximizes an objective function.

As input, this component receives a notification from the *Analyzer* component indicating that some condition is not (or may not be) fulfilled. The *Planning* component executes an strategy and generates a sequence of actions that aim to take the application to an objective state. If required, it can use the *Monitoring* Component to request certain metrics. The generated actions, once again, can take a very simple form (a shellscript) or a more complex one, as a sequence of actions described in a domain-specific language that can be interpreted and executed by a set of actuators.

This encapsulation allows the framework to replace at runtime the strategy to reach the objective, for example from a cost-optimizing planner to an energy-efficiency planner, or well taking no action at all.

### D. Execution

The *Execution* component carries on the decided modifications to the service, or to a set of services, as indicated by the *Planning* component.

The execution requires an integrated means to access the managed service in order to execute the actions upon it. In a similar way to the *Monitoring* component, the *Execution* component must implement any specific protocol required by the managed service in order to be able to trigger adaptations on it.

The set of actions demanded may involve not only the managed service, but also different service(s). For this reason, the *Execution* component is also able to communicate with the *Execution* components attached to some other components and send actions to them as part of the main reconfiguration action. The set of connected *Execution* components forms an “execution backbone” that propagates the actions from the component where the actions have been generated to each of the specific components where some part of the actions must take place, possibly hierarchically down to their respective inner components. This approach allows to decentralize the execution of the actions.

One of the challenges in the execution phase is to ensure that the reconfiguration or adaptations actions will not make the application enter in an unsafe state. This problem is left to the execution implementation.

Figure 5 shows a sample situation in which the *Analyzer* component of the *Tourism Service* component finds out that the cost of the compostion if exceeding a desired threshold. The *Analyzer* notifies the *Planning* component, which executes an strategy oriented to replace the component with the higher cost by a cheaper one. The *Planning* component uses its *Monitoring* component to get the cost metric of several components, and decides to replace the component *Attraction1* by the (functionally equivalent) *Attraction2*. The action is carried on by the *Execution* component.

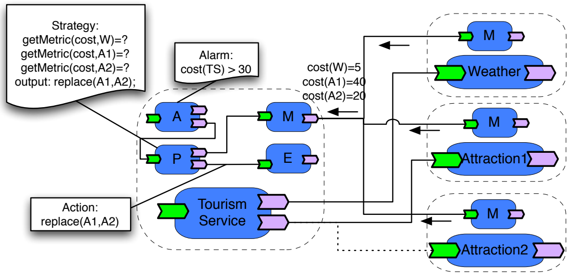

Figure 5. SCA Components Tourism Service reacts to an SLA violation by replacing the component that features the highest cost.

## IV. Implementation

This section describes our prototype implementation over the GCM model. We describe the pieces of the framework that we have implemented and exemplify how they can be used to provide self-adaptability in the context of the scenario described in Section II.

### A. Background: SCA compliant GCM/ProActive

The ProActive Grid Middleware [2] is a Java middleware, which aims to achieve seamless programming for concurrent, parallel and distributed computing, by offering an uniform active object programming model, where these objects are remotely accessible via asynchronous method invocations with futures. Active Objects are instrumented with MBeans, which provide notifications about events at the implementation level, like the reception of a request, and the start and end of a service. The notification of such events to interested third parties is provided by an asynchronous and grid enabled JMX connector.

The Grid Component Model (GCM) [3] is a component model for applications to be run on computing grids, that extends the Fractal component model [4]. Fractal defines a component model where components can be hierarchically organized, reconfigured, and controlled offering functional server interfaces and requiring client interfaces (as shown in Figure 6). GCM extends that model providing to the components the possibility to be remotely located, distributed, parallel, and deployed in a grid environment, and adding collective communications (multicast and gathercast interfaces). In GCM it is possible to have a componentized membrane [5] that allows the existence of non-functional (NF) components, also called *component controllers* that take care of non-functional concerns. NF components can be accessed through NF server interfaces, and components can make requests to NF services using NF client interfaces (shown respectively on top and bottom of $C$ in Figure 6).

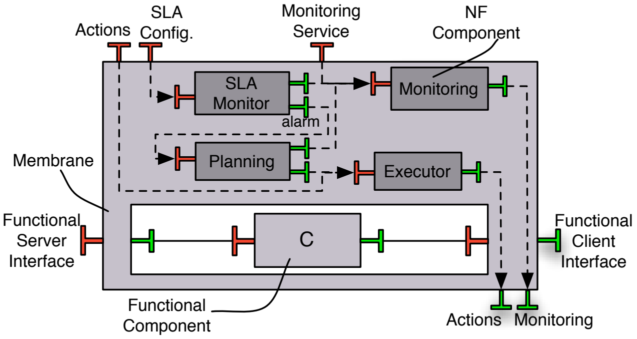

Figure 6. Framework implementation weaved to a primitive GCM component $C$

The use of NF components instead of simple object controllers as in Julia, the Fractal reference implementation, allows to have a more flexible control of NF concerns and to develop more complex implementations, as the NF components can be bound to other NF components within a regular component application. In this sense, this paper complements some previous ones about the componentized membrane [3], [5], [6], particularly addressing the concerns of self-adaptability in service-oriented contexts.

GCM/ProActive is the reference implementation of GCM, within the ProActive middleware, where components are implemented by Active Objects, which can be used to implement new services, or wrap existent legacy applications like C/Fortran MPI code, or a BPEL code.

The GCM/ProActive platform provides asynchronous communications with futures between bound components through GCM bindings. GCM bindings are used to provide asynchronous communication between GCM components, and can also be used to connect to other technologies and communications protocols, like Web Services, by implementing the compliance to these protocols via specific controllers in the membrane. These controllers have been used to allow GCM to act as an SCA compliant platform, in a similar way as achieved by the SCA FraSCAti [7] platform, which however bases upon non distributed components (Fractal/Julia) in contrary to GCM ones.

### B. Framework Implementation

The framework is implemented in the GCM/ProActive middleware as a set of NF components that can be added or removed at runtime to the membrane of any GCM component, which becomes a managed service of the application. The ability to reconfigure the composition of the membrane at runtime is provided by the middleware [5].

We have designed a set of predefined components that implement each one of the elements we have described in Section III. This is just one of possible implementations, and particularly this has been designed to provide self-adaptable capabilities to the composition.

The general implementation view for a single GCM component is shown in Figure 6 (using the GCM graphical notation [3]), and resembles the design presented in Figure 2. The framework is weaved in the primitive GCM component *C* by inserting NF components in its membrane. Monitoring and management features are exposed through the NF server interfaces "Monitoring Service", "SLA Config" and "Actions" (top of Figure 6). NF components can communicate with the NF components of other GCM components through the NF client interfaces "Monitoring" and "Actions" (bottom of Figure 6). The sequence diagram of the self-adaptability loop is shown in Figure 7.

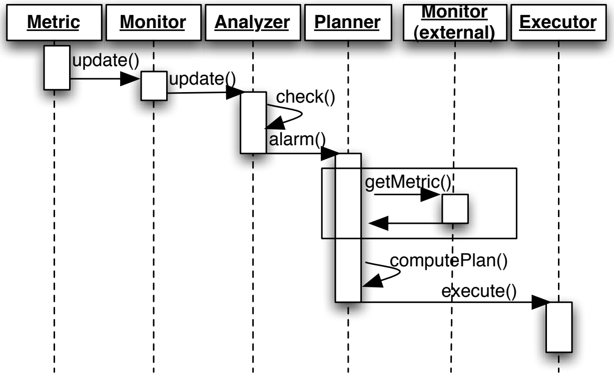

Figure 7. Sequence diagram for the autonomic control loop

### C. Monitoring

We have designed a set of probes, f.e. for CPU load and memory use, and incorporated them along with the events produced by the GCM/ProActive platform [8]. Over them, we provide a *Monitoring* component, shown on Figure 8, which includes (1) an *Event Listener* that receives events from a GCM component and provides a common ground to access them; (2) a *Record Store* to store records of monitored data that can be used for later analysis; (3) a *Metric Store* that stores objects that we call *Metrics*, which actually compute the desired metrics using the records stored, or the events caught; and (4) a *Monitor Manager*, which provides the interface to access the stored metrics, and add/remove them to/from the Metrics Store.

The *Monitor Manager* receives a Metric as a Java object with a *compute* method, and inserts it in the *Metric Store*. The *Metric Store* provides to the Metrics the connection to the sources that they may need; namely, the *Record Store* to get already sensed information, the *Event Listener* to receive sensed information directly, or the *Monitoring* component of other external components, allowing access to the distributed set of monitors (i.e., to the monitoring backbone). For example, a simple *respTime* metric to compute the response time of requests, requires subscription to the *Event Listener* for events related to the start and finish times of the service of a request.

As a more complex example, the Tourism Service needs to know the decomposition of the time spent while serving a specific request $r_0$. For this, a metric called *requestPath* for a given request $r_0$ can ask the *requestPath* to the *Monitoring* components of all the services involved while serving $r_0$, which can repeat the process themselves; when no more calls are found, the composed path is returned with the value of the *respTime* metric for each one of the services involved in the path. Once the information is gathered in the *Monitoring* component of the Tourism Service, the complete path is built and it is possible to identify the time spent in each service.

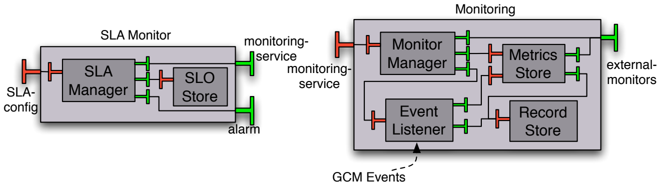

Figure 8. Internal Composition of the Monitoring component (right) and the SLA Monitor component (left)

### D. SLA Monitor

The *SLA Monitor* is implemented as a component that queries the *Monitoring* component. The *SLA Monitor* consists in (1) an *SLA Manager*, which exposes an interface that allows to add/remove SLOs expressed in a specific format, checks the fulfillment of the SLOs, and sends a notification when some of them are not fulfilled; and (2) an *SLO Store*, which maintains the list of SLOs. The composition is shown in the left side of Figure 8.

In this implementation, an SLO is described as a triple $\langle metricN, comparator, value \rangle$, where *metricN* is the name of a metric. The SLA Monitor subscribes to the *metricN* from the *Monitoring* component to get the updated values and check the compliance of the SLO.

For example, the Tourism Service service includes the SLO: "All requests must be served in less than 30 secs", described as $\langle respTime, <, 30 \rangle$. The *SLA Manager* receives this description and sends a request to the *Monitoring* component for subscription to the *respTime* metric. The condition is then stored in the *SLO Store*. Each time an update on the metric is received, the *SLA Manager* checks all the SLOs associated to that metric. In case one of them is not fulfilled, a notification is sent, through the *alarm* interface including the description of the faulting SLO.

### E. Planning

The *Planning* component, shown on the left side of Figure 9, includes a *Strategy Manager* that receives an alarm message and, depending on the content of the alarm, it triggers one of several bound *Planner* components. Each one of the *Planner* components implements the logic required to make a decision to restore the state of the application to comply with a failed SLO. Each *Planner* component can access the *Monitoring* components to retrieve any additional information they may need about the composition; the output is expressed as a list of actions in a predefined language.

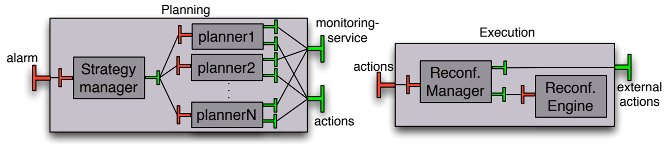

Figure 9. Internal Composition of the Planning and the Execution components

In our implementation we profit of selective 1-to-N communications provided by GCM to decide which *Planner* component to trigger. For example, if the SLO violated is related to response time, we may trigger a performance-oriented recomposition; or if a given cost has been surpassed, we may trigger a cost-saving algorithm. The decision of what strategy to use is taken in the *Strategy Manager* component. However, the possibility of having multiple strategies might be a source for conflicting decisions; while we do not provide a method to solve these kind of conflicts, we assume that the conflict resolution behaviour, if required, is provided by the *Strategy Manager*.

We have implemented a simple planning strategy that, given a particular request, asks to compute the *requestPath* for that request, then finds the component most likely responsible for having broken the SLO, and then creates a plan that, when executed, will replace that component for another component from a set of possible candidates. Applied to the Tourism Service, suppose a request has violated the SLO $\langle respTime, <, 30\rangle$. The *Strategy Manager* activates the *Planner* component that obtains the *requestPath* for that request along with the corresponding response time, selects the component that has taken the highest time, then obtains a set of possible replacements for that component and obtains for each of them the *avgRespTime* metric. The output is a plan expressed in a predefined language that aims to replace the slowest component by the chosen one.

Clearly this strategy does not intend to be general, and does not guarantee an optimal response in several cases. Even, in some situations, it may fail to find a replacement and in that case the output is an empty set of actions. However, this example describes a planning strategy that can be added to implement an adaptation for self-optimizing and that uses monitoring information to create a list of actions.

### F. Execution

The *Execution* component, shown on the right side of Figure 9, includes a *Reconfiguration Engine*. This engine uses a domain specific language called PAGCMScript, an extension of the FScript [9] language (designed for Fractal components), which supports GCM specific features like distributed location, collective communications, and remote instantiation of components.

The *Execution* component receives actions from the *Planning* component. As many strategies may express actions using different formats, a component called *Reconfiguration Manager* may need to apply a transformation to express the actions in an appropriate language for the *Reconfiguration Engine*. The *Reconfiguration Manager* may also discriminate between actions that can be executed by the local component, or those that must be delegated to external *Execution* components.

In the example, once the "Attraction2" provider has been selected, it can be unbound from the "Tourism Service" using a PAGCMScript command like the following, whose effect can be seen in Figure 5:

```
unbind($tourism/interface::"attraction2")
```

### G. Generalization

As GCM is an SCA compliant platform, the GCM-based framework, as shown in Figure 6 can be described in SCA terms providing a view that can be realized for any SCA runtime platform like that in Figure 2. The deployment of the framework may be done by injecting the required SCA description in the SCA ADL file. However, in order to allow this modification to occur at runtime, we have provided a console application that can use the standard non-functional API of GCM components to insert or remove at runtime the required components of the framework.

The console, while not being itself a part of the framework, shows that an external application can be built and connected to the NF interfaces of the running application and handle at runtime the composition and any subsequent reconfiguration, if needed, of the monitoring and management framework itself.

## V. Related Work

Several works exist regarding monitoring and management of service-oriented applications. Most of them tackle separately monitoring infrastructures [10], SLA monitoring and analysis [11], SLA fulfillment [1], and planning strategies for adaptation [12], [13], [14]. A few others, like us, propose a complete framework. The work [15] is similar to ours in that they propose a generic context-aware framework that separates the steps of the MAPE control loop to provide self-adaptation; their work allows the implementation of self-adaptive strategies, though not much is mentioned about runtime reconfigurability, or the possibility to have multiple strategies. Also, we do not necessarily consider that all services require the same level of autonomicity.

CEYLON [16] is a service-oriented framework for integrating autonomic strategies available as services and use them to build complex autonomic applications. They provide the managers that allow to integrate and adapt the composition of the autonomic strategies according to evolving conditions. In CEYLON, autonomicity is a main *functional* objective in the development of the application, while in our case, we aim to provide autonomic QoS-related capabilities to already existing service based applications. Also, we take benefit of the business-level components intrinsec distribution and hierarchy to split the implementation of monitoring and management requirements across different levels, thus enforcing scalability.

## VI. Conclusions and Perspectives

We have presented a generic component-based framework for supporting monitoring and management tasks of component-based SOA applications. The component based approach allows a clear separation of concerns between the functional content and the management tasks. We have implemented a prototype that provides a self-adaptation loop for component-based services, thanks to the composition of appropriate monitoring, SLA management, planning and reconfiguration components. This prototype has been developed in the context of an SCA compliant platform that includes dynamic reconfiguration and distribution capabilities.

This approach provides a high degree of flexibility as the skeleton we have provided for the autonomic control loop can be personnalized to e.g., support different planning strategies, and leverage heterogeneous monitoring sources to provide the input data that these strategies may need (for example, performance, price, energy consumption, availability). Early evaluation shows a small overhead no bigger than $4\%$ in the execution of basic reconfiguration operations (namely, insertion of a new SLO and metrics, communication between functional and non-functional components, and runtime architectural rebindings), with respect to the performance before attaching the components of the framework. We expect to provide a set of benchmarks to clearly establish this overhead. The hierarchical approach is expected to provide high scalability, though a bigger experimentation set is still required.

## References

[1] P. Leitner, B. Wetzstein, F. Rosenberg, A. Michlmayr, S. Dustdar, and F. Leymann, "Runtime prediction of service level agreement violations for composite services," in *Proceedings of the 2009 international conference on Service-oriented computing*, ser. ICSOC/ServiceWave'09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 176–186.

[2] "ProActive Parallel Suite," accessed on 24-Mar-2011. [Online]. Available: http://proactive.inria.fr/

[3] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez, "GCM: a grid extension to Fractal for autonomous distributed components," *Annals of Telecommunications*, vol. 64, no. 1-2, pp. 5–24, 2009.

[4] E. Bruneton, T. Coupaye, M. Leclercq, V. Quma, and J.-B. Stefani, "The fractal component model and its support in java," *Software: Practice and Experience*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[5] F. Baude, L. Henrio, and P. Naoumenko, "Structural reconfiguration: An autonomic strategy for gcm components," in *Proceedings of the 2009 Fifth International Conference on Autonomic and Autonomous Systems*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 123–128.

[6] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo, "A framework for experimenting with structured parallel programming environment design," in *Parallel Computing - Software Technology, Algorithms, Architectures and Applications*, ser. Advances in Parallel Computing. North-Holland, 2004, vol. 13, pp. 617 – 624.

[7] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani, "Reconfigurable sca applications with the frascati platform," *Services Computing, IEEE International Conference on*, vol. 0, pp. 268–275, 2009.

[8] C. Ruz, F. Baude, and B. Sauvan, "Enabling SLA Monitoring for Component-Based SOA Applications – A Component-Based Approach," in *Proceedings of the Work in Progress Session SEAA 2010*. Johannes Kepler University Linz, 2010, pp. 41–42.

[9] P.-C. David, T. Ledoux, M. Lger, and T. Coupaye, "Fpath and fscript: Language support for navigation and reliable reconfiguration of fractal architectures," *Annals of Telecommunications*, vol. 64, pp. 45–63, 2009.

[10] A. Van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst, "Continuous Monitoring of Software Services: Design and Application of the Kieker Framework," p. 26, 2009.

[11] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, "Comprehensive QoS monitoring of Web services and event-based SLA violation detection," in *Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing*, ser. MWSOC '09. New York, NY, USA: ACM, 2009, pp. 1–6.

[12] G. Canfora, M. Di Penta, R. Esposito, F. Perfetto, and M. Villani, "Service composition (re)binding driven by applicationspecific qos," in *Service-Oriented Computing ICSOC 2006*, ser. Lecture Notes in Computer Science, A. Dan and W. Lamersdorf, Eds. Springer Berlin / Heidelberg, 2006, vol. 4294, pp. 141–152.

[13] V. Cardellini and S. Iannucci, "Designing a broker for qos-driven runtime adaptation of soa applications," *Web Services, IEEE International Conference on*, vol. 0, pp. 504–511, 2010.

[14] C. Ghezzi, A. Motta, V. Panzica, L. Manna, and G. Tamburrelli, "QoS Driven Dynamic Binding in-the-many," *QoSA 2010*, pp. 68–83, 2010.

[15] F. Andre, E. Daubert, and G. Gauvrit, "Towards a generic context-aware framework for self-adaptation of service-oriented architectures," *Intl. Conf. on Internet and Web Applications and Services, ICIW*, vol. 0, pp. 309–314, 2010.

[16] Y. Maurel, A. Diaconescu, and P. Lalanda, "CEYLON: A Service-Oriented Framework for Building Autonomic Managers," *7th IEEE Intl. Conf. and Workshops on Engineering of Autonomic and Autonomous Systems*, pp. 3–11, Mar. 2010.