

# Reconfigurable SCA Applications with the FraSCAti Platform \*

Lionel Seinturier - Philippe Merle - Damien Fournier - Nicolas Dolet  
University of Lille & INRIA Lille - Nord Europe  
LIFL UMR CNRS 8022, ADAM Project-Team  
Villeneuve d'Ascq, France  
Email: `firstname.lastname@inria.fr`

Valerio Schiavoni - Jean-Bernard Stefani  
INRIA Grenoble - Rhône-Alpes  
SARDES Project-Team  
Montbonnot, France  
Email: `firstname.lastname@inria.fr`

June 23, 2009

## Abstract

The Service Component Architecture (SCA) is a technology agnostic standard for developing and deploying distributed service-oriented applications. However, SCA does not define standard means for runtime manageability (including introspection and reconfiguration) of SOA applications and of their supporting environment. This paper presents the FraSCAti platform, which brings runtime management features to SCA, and discusses key principles in its design: the adoption of an extended SCA component model for the implementation of SOA applications and of the FraSCAti platform itself; the use of component-based interception techniques for dynamically weaving non-functional services such as transaction management with components. The paper presents micro-benchmarks that show that runtime manageability in the FraSCAti platform is achieved without hindering its performance relative to the de facto reference SCA implementation, Apache's Tuscany.

## 1 Introduction

Service-Oriented Architecture (SOA) requires appropriate software platforms for the delivery, support, and management of distributed applications conform-

---

\* Acknowledgement: This work is partially supported by the ANR (French National Research Agency) TLog SCOrWare project and the IST FP7 IP SOA4All project.

ing to its principles. The recently developed Service Component Architecture (SCA) [1] aims to fulfill that need with a specification for an SOA platform that is technology (i.e. programming language and protocol) agnostic, and that supports a view of services as software components. Several platforms have already been developed that implement the SCA specification, such as Tuscany ([tuscany.apache.org](http://tuscany.apache.org)), and Newton ([newton.codecauldron.org](http://newton.codecauldron.org)). Although SCA is not the first approach that combines software components and services (see, for example, OSGi [2]), its technology independence, its support for hierarchical component composition, and its support for distributed configurations, where remote components can be interconnected by various means, make it an interesting contender in the SOA platform space.

Unfortunately, the SCA specification falls short of providing the required level of manageability and configurability that can be expected from an SOA platform. For instance, while the SCA specification allows to control the installation and configuration of service components, it falls short of providing the required capabilities to manage at runtime a component configuration, or the association of service components with platform-provided non-functional services (such as transaction management, persistency management, etc), to control the execution of service components (e.g. to handle on-line changes in configurations), or to provide appropriate hooks for the management of the platform itself (to administer fault, performance, configuration and security in distributed SOA environments). To our knowledge, existing SCA implementations, whether supporting SCA natively such as Tuscany, or layering SCA support on top of an existing deployment and execution platform (e.g. OSGi) such as Newton, fall similarly short of providing the required capabilities.

In this paper, we present the FraSCAti platform for Java-based SCA applications. Compared to existing platforms, the main contribution of FraSCAti is to address the above issues of configurability and manageability in a systematic fashion, at the business (application components), and platform (non-functional services, communication protocols, etc.) levels. This is achieved through an extension of the SCA component model with reflective capabilities, and the use of this component model both to implement business-level service components conforming to the SCA specification, and to implement the FraSCAti platform itself.

This paper is organized as follows. Section 2 presents the SCA standard and discusses the configurability and manageability issues left open by the standard. Section 3 describes the extended SCA component model supported by the FraSCAti platform and used for the implementation of the platform itself. It also describes the use of interception techniques to support component meta-level capabilities such as non-functional services. Section 4 describes the component-based architecture of the FraSCAti platform. Section 5 reports on the implementation of the platform and on some performance measurements. Section 6 discusses related work. Finally, Section 7 concludes the paper and hints at future work.

## 2 The SCA standard and platform challenges

**The SCA standard** The Service Component Architecture (SCA) [1] is a set of specifications for building distributed applications using Service Oriented Architecture (SOA) and component-based software engineering (CBSE) principles. The model is promoted by a group of companies, including BEA, IBM, IONA, Oracle, SAP, Sun and TIBCO. The specifications are now defined and hosted by the Open Service Oriented Architecture (OSOA) collaboration<sup>1</sup> and promoted in the Open CSA section of the OASIS consortium<sup>2</sup>.

SOA, e.g. when based on Web Services, provides a way for exposing coarse grained and loosely coupled services which can be remotely accessed. But the SOA approach does not address the issue of the way these services should be implemented. SCA fills this gap by defining a component model for SOA applications. SCA entities are software *components* which may provide interfaces (called *services*), require interfaces (called *references*) and expose properties. References and services are connected through *wires*. The model is hierarchical with components being implemented either by primitive language entities or by subcomponents (the component is then said to be *composite*). Figure 1, which is taken from the SCA specifications [1], provides the graphical notation for these concepts. A XML-based assembly language is available to configure and assemble components.

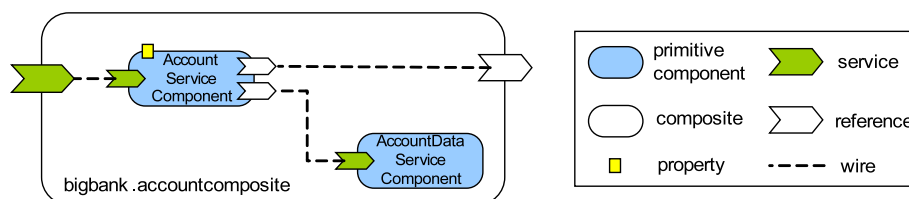


Figure 1: SCA component architecture (example from [1]).

Four main principles underlie the design of SCA. They are meant to define a service architecture which is as independent as possible from underlying implementation technologies.

*Independence from programming languages.* SCA does not assume that components will be implemented with a unique programming language. Rather, several language mappings are supported and allow programming SCA components in Java, C++, PHP, BPEL or COBOL. The Java language mapping takes advantage of Java 5 annotations for supporting component-based notions such as dependency and property injection.

*Independence from interface definition languages.* SCA components provide (resp. require) functionalities through precisely defined interfaces. SCA does not assume that a single interface definition language (IDL). Rather, several IDL are supported such as WSDL and Java interfaces.

<sup>1</sup>[www.osoa.org](http://www.osoa.org)

<sup>2</sup>[www.oasis-opencsa.org](http://www.oasis-opencsa.org)

*Independence from communication protocols.* Although Web Services are the preferred communication mode for SCA components, this solution may not fit all needs. In some cases, protocols with different semantics and properties (e.g. performance) may be needed. For that, SCA provides the notion of *binding*: a service or a reference can be bound to a particular communication protocol such as SOAP for Web Services, Java RMI, Sun JMS or REST.

*Independence from non-functional properties.* Non-functional properties may be associated to an SCA component with the notion of *policy set* (the notion is also referred to with the term *intent*). The idea is to let a component declare the set of policies (non-functional services) that it depends upon. The platform is then in charge of guaranteeing that these policies are enforced. So far, security and transactions have been included in the SCA specifications. Yet, developers may need other types of non-functional properties (e.g. persistence, logging). For that, the set of supported policy sets may be extended with user-specified values.

These principles offer a broad scope of solutions for implementing SCA-based applications. Developers may think of incorporating new forms of language mappings (e.g., SCA components programmed with EJB or OSGi), IDL (e.g., CORBA IDL), communication bindings (e.g., JBI bindings) and non-functional properties (e.g., persistence, logging, etc.)

**SCA platform challenges** In our opinion, two important challenges are to be met by SCA platform providers. First, if the SCA specifications offer, at the application level, the mechanisms for declaring a broad range of variation points (as discussed above), nothing is said about the architecture of the platform which is supposed to implement these variations. It is thus a matter of performing the adequate design choices to obtain a platform which is flexible and extensible enough to accommodate and integrate smoothly these variations.

Second, the SCA specifications are centered around the task of describing the assembly and the configuration of the components which compose the application. This assembly is meant to be taken as input by the deployment service of the platform to instantiate and initialize the application. The SCA specifications do not address the runtime management of the application, which typically includes monitoring and reconfiguring it. Importantly, the SCA specification does not address either the runtime management of the platform itself. Yet, we believe that these properties are almost mandatory for modern SOA platforms in order to be able to adapt to changing operating conditions, to support online evolution, and to be deployed in dynamically changing environments (e.g. cloud computing environments).

In the next section, we present the design of the FraSCAti platform which addresses these two challenges.

### 3 The FraSCAti component model

In order to meet the configurability and manageability challenges discussed above, the FraSCAti platform supports an extended SCA component model, where components can be equipped with reflective capabilities to allow their introspection, monitoring, control, and dynamic configuration. In particular, these reflective capabilities allow the dynamic integration of non-functional services and properties associated with components. To support these reflective capabilities, FraSCAti associates with each component a form of generalized *container*. The integration of non-functional services and properties is realized by means of interception techniques. We describe these two main elements in the sub-sections below.

#### 3.1 Container architecture

Most component frameworks provide a computing infrastructure for hosting and running components in the form of so-called "containers", such as EJB ones, that provide in a quasi-transparent manner platform-wide classical middleware services to application components.

In FraSCAti, we generalize the notion of container to a notion of component meta-level that provides access to various services. Each of these services can be seen as implementing a fine-grained framework to handle a particular facet of the management of an SCA component. Figure 2 illustrates the design of the FraSCAti container. Each square-cornered box symbolizes a different service. Far from being independent, these services need to collaborate to provide the overall behavior to the business logic instance hosted in this container. This collaboration scheme is captured in a software architecture. Each link symbolizes a *use* relationship between two services. This software architecture constitutes the backbone of the implementation of a FraSCAti component container.

We thus obtain a two-level architecture where each SCA component is hosted by a container which is itself implemented as a component-based architecture. The services provided by a FraSCAti container are listed below.

1. **Component Wiring.** This service provides the ability, for each component, to query the list of existing wires, to register new wires and to remove wires. These operations can be performed on a running SCA application.
2. **Component Instantiation.** The SCA specifications define 4 modes when instantiating a component: **STATELESS** (all instances of a component are equivalent), **REQUEST** (an instance is created per request), **CONVERSATION** (an instance is created per conversation with a client), and **COMPOSITE** (singleton wrt the enclosing composite component). The Component Instantiation service allows creating component instances according to one of these 4 modes.
3. **Component Property.** This service enables setting and getting the values of component properties.

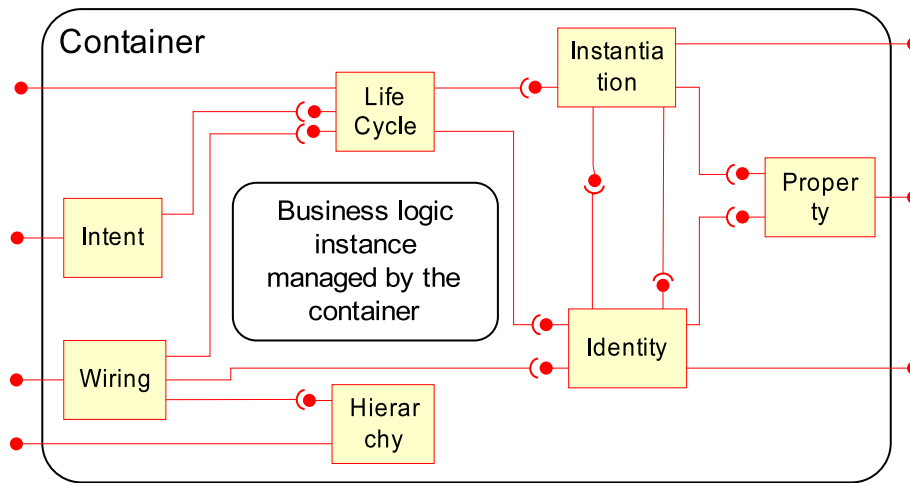


Figure 2: FraSCAti container architecture.

4. **Component Identity.** This service manages the identity of a component and allows querying the set of provided services and required references by the component. The purpose is similar to that of the `IUnknown` interface in the COM component framework and allows to dynamically discover the capabilities and the requirements of a component.
5. **Component Hierarchy Management.** The SCA component model is hierarchical in the sense that a component is either primitive or composite. Composite components contain subcomponents which are themselves, either primitive or composite. The resulting hierarchy is a tree. The management of this hierarchy is performed by two services: one for adding/querying/removing the subcomponents of a composite, and one for retrieving the parent of a component.
6. **Component Lifecycle.** When dealing with multithreaded applications (the general case of distributed applications targeted by the SCA specifications), reconfiguration operations, such as the ones mentioned in the previous items, can not be performed in an uncontrolled way. Indeed, modifying a wire while a client request is being served may lead to inconsistencies and wrong results or errors returned to clients. For that, the lifecycle service ensures that reconfiguration operations are performed safely and consistently in isolation with client requests. This service is said to control the lifecycle of the components in the sense that it strictly delimits the time intervals during which reconfiguration operations can be performed and those during which application level requests can be processed.
7. **Intent Management.** This service manages the non-functional proper-

ties which are attached to an SCA component. This service is described in detail in Section 3.2.

Compared to the SCA Assembly Language which allows for describing only the initial static configuration of an application, the novelty of FraSCAti is to make this configuration accessible and modifiable while the application is being executed. For example, based on the application illustrated in Figure 1, a reconfiguration scenario can consist in replacing component `AccountDataServiceComponent` by component `NewAccountDataServiceComponent`. The process is composed of five steps: 1) stop the component (bringing it to a quiescent state), 2) remove the existing wire, 3) create a new component, 4) recreate the wire with the new component, 5) restart the component. Note that steps 1 and 5 are meant to ensure that the reconfiguration is consistent with respect to client requests. Stopping the component ensures that no client request is processed while the reconfiguration takes place. This limitation could be removed, and the corresponding steps (1 and 5) could be skipped if one does not need such a guarantee. Due to space limitation, we do not provide the details of the API which implements these operations. Readers can refer to [3] for further information.

By providing a runtime API, the FraSCAti platform enables the dynamic introspection and modification of an SCA application. This feature is of particular importance for designing and implementing agile SCA applications, such as context-aware applications and autonomic applications [4]. For instance, [5] shows that the combination of wiring, hierarchy management, property, identity, and lifecycle are required meta-level capabilities in a component model to support fully self-repair in a component-based distributed system. The same reconfiguration capabilities have been exploited to automate overload management in component-based cluster systems [6].

### 3.2 Integration of non-functional services

SCA provides a mechanism for attaching metadata (Java 5 annotations in the case of the Java language mapping) to component assemblies with the SCA Policy Framework specification. These metadata elements influence the way applications behave by triggering the execution of non-functional services. For example, the `@Confidentiality`, `@Integrity` and `@Authentication` metadata ensure confidentiality, integrity and authentication of service invocations. Some general purpose metadata such as `@Intent` and `@Requires` are also available for associating any other kind of non-functional services to SCA applications. However the SCA standard does not define any mechanism for binding and managing the non-functional services. This is left as a platform-specific issue.

We propose with the FraSCAti platform, two innovative solutions for putting this binding into practice and for facilitating the integration of new non-functional services into SCA applications: (i) to implement non-functional services as regular SCA components, (ii) to provide an interception mechanism to glue the non-functional components with application components.

By using SCA components to implement non-functional services, we provide an integrated solution where there is only one paradigm for implementing business and technical concerns. Note that this component may be composite and be the result of the assembling of several other components.

By using interceptors to integrate technical services with business code, we keep these concerns cleanly separated not only at design time, but also at runtime. Figure 3 illustrates the interception mechanism. Each interceptor registers with a particular policy metadata. When an SCA Assembly Descriptor is parsed by the FraSCAti platform, the interceptors are added on the services and/or references annotated with the registered metadata. When a client request is served by the corresponding component, the request is first trapped and handled by the interceptor which applies its logic. After that, the request is transmitted to the component. Interceptors act as filters on the business logic control flow.

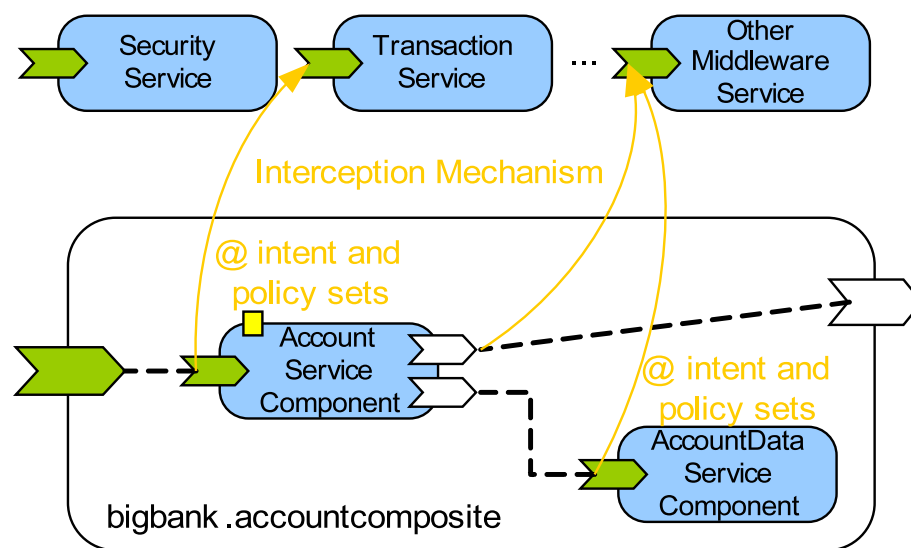


Figure 3: FraSCAti interception mechanism.

Interceptors with FraSCAti are dynamic in the sense that they can be woven or removed at runtime. For that, an API which is similar to the ones which can be found in FAC [7] or JBoss AOP [8] is provided. Due to space limitation, we do not provide the details of the API which implements these operations. Readers can refer to [3] for further information.

## 4 The FraSCAti platform architecture

We present in this section the architecture of the FraSCAti platform, depicted in Figure 4. It is component-based, and uses the same component model than FraSCAti applications, described in the previous section. The platform has four

main components, which we present below.

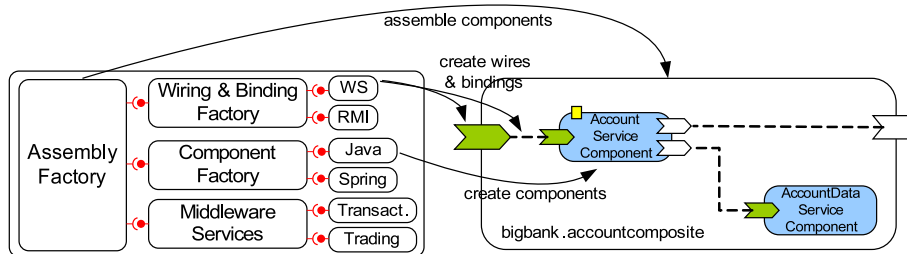


Figure 4: FraSCAti platform architecture.

1. **Component Factory.** This part is in charge of creating *containers* (which are discussed in Section 3) and components.
2. **Wiring & Binding Factory.** This part is in charge of creating wires between components. As stated in Section 2, SCA is independent from communication protocols: components are specified first, and are next bound using a selected distribution technology. This allows decoupling the tasks of designing the business logic of the component from the details of the protocol which implements the communications. Besides creating internal wires in an SCA application, the Wiring & Binding Factory is also in charge of exporting services and references (the outer most services and references of an SCA application) to the "external" world.
3. **Middleware Services.** This part is a repository for the non-functional services which are made available to the applications. The gluing of these services with the SCA components has been explained in Section 3.2.
4. **Assembly Factory.** This part is responsible for parsing the SCA assembly language descriptors, interpreting the XML tags and creating the corresponding component assemblies. Whenever necessary, the Assembly Factory relies on the Component Factory for creating components, on the Wiring & Binding Factory for creating wires, exporting services and references and on the Middleware Services for integrating non-functional concerns into applications.

Wiring and binding between components is fully dynamic in FraSCAti: communication protocols are encapsulated as binding components, which can be instantiated and wired to applications components at runtime. In addition, the first three main components above are implemented as general purpose component factories and allow creating in new managers (e.g. for a new communication protocol, or a new non-functional service) whenever needed. This enables the hot pluggability of new managers to tailor the platform to new usage conditions, unforeseen at startup. By default, SCA Java and Spring are the two available

plugins for the Component Factory, SOAP and Java RMI are the ones for the Wiring & Binding Factory, and Transaction and Trading are the two registered plugins with the Middleware Services. Figure 4 illustrates the respective roles of the core elements in setting up and managing an SCA application.

## 5 Implementation and evaluation

This section reports briefly on the implementation of the FraSCAti platform and provides some performance measurements. These measurements show that the extra capabilities in terms of extensibility and reconfigurability which have been introduced in the FraSCAti platform, and which are not available in the de facto current reference implementation of SCA, do not penalize performance.

### 5.1 Implementation

The FraSCAti platform is implemented in Java and can be downloaded from [frascati.ow2.org](http://frascati.ow2.org). Note that, although the SCA specification is independent from programming languages, a platform that implements the specification is implemented in some programming language (Java in our case) and is thus preferentially tied to a programming language. FraSCAti can be run in two modes: as a standalone application server or embedded as a service engine in the OW2 PEtALS<sup>3</sup> JBI Enterprise Service Bus.

Besides Java, the FraSCAti platform and its component model rely on Fractal [9], a lightweight and open component framework. The granularity of a Fractal component is finer than that of an SCA component and closer to that of an object. The Fractal framework can be extended to customize the execution semantics of a component. Its extensibility and its lightweight character are key factors to support the dynamicity and the reconfiguration properties which are brought by FraSCAti to SCA applications.

The FraSCAti platform has been used to implement the demonstrators of the ANR SCOrWare project: service-oriented scientific computing [10], a new generation of collaborative development forge, a business-to-business platform and system monitoring.

### 5.2 Performance evaluation

In order to evaluate the performances on our platform, we compare it to Apache Tuscany Java SCA version 1.3.2 which is the de facto current reference implementation of SCA. We devised a simple microbenchmark to compare the memory consumption and the execution time of FraSCAti with that of Tuscany. The measurements have been conducted on an Intel Core Duo T2300 1.66 GHz PC with 2GB of RAM running Windows XP and JDK 1.6.0-07.

The first series of measurements evaluates the cost of the FraSCAti container infrastructure. Figure 5 compares the evolution of memory usage depending on

---

<sup>3</sup>[petals.ow2.org](http://petals.ow2.org)

the number of instantiated components. The assembly takes the form of a tree of components. All instantiated components share the same implementation. The measurements stop when the Java VM runs out of memory. The measurements show that FraSCAti scales better and, whereas Tuscany fails from instantiating, in our configuration, more than 6,500 components, FraSCAti succeeds in instantiating assemblies more than twice as large.

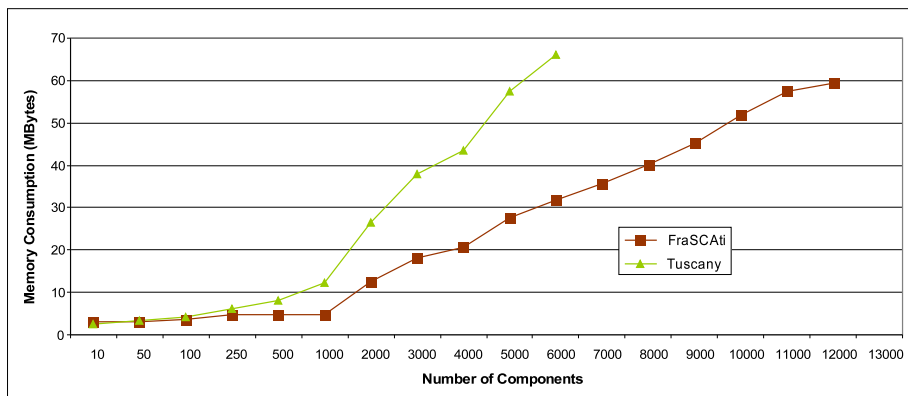


Figure 5: Memory consumption in MBytes per number of instantiated components.

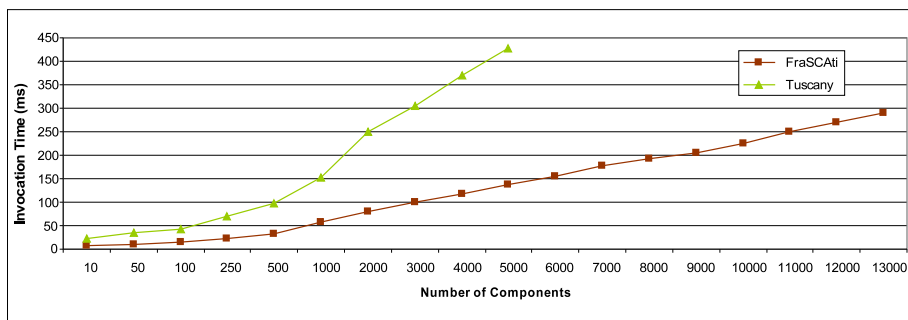


Figure 6: Invocation time in ms per number of instantiated components.

The second series of measurements concerns the cost induced by FraSCAti when invoking a service. Figure 6 compares the execution time over local wires. The scenario consists in invoking the root component of the assembly which, in its turn, invokes its two child components. The invocation is repeated by each node component in the tree until the leaves are reached and the invocations are returned. The purpose of the experiment is thus to measure the cost of invoking an SCA component over a local wire. As with the previous experiment, the measurements cannot be performed with Tuscany for assemblies larger than 5,000 components. The process goes on with FraSCAti up to 13,500 components.

Furthermore, one can witness that the cost grows more linearly with FraSCAti. Finally, for assemblies of 5,000 components the invocations is almost twice as fast with FraSCAti than with Tuscany.

These performance measurements show that the design of the FraSCAti platform provides comparable or better performance than the reference platform of the domain, while providing introspectability and reconfigurability. The relevance of these microbenchmarks lies in the fact that they reveal the memory and the CPU consumed by the infrastructure which brings the SCA related functionalities to the hosted business components. Although the reasons why FraSCAti performs better than Tuscany would require a careful and deep analysis, our first investigations point to a design choice: FraSCAti generates statically the proxies which are needed for the component containers. This improves the performance compared to solutions based on dynamic proxies.

## 6 Related work

This section compares FraSCAti with other approaches in terms of SCA platforms and component models.

**SCA platforms** Several implementations of the SCA specifications are available, either commercial ones (e.g., HydraSCA from Roque Wave Software, IBM WebSphere Application Server Feature Pack for SOA, Oracle Event-Driven Architecture Suite) or open source ones: Tuscany, Newton, fabric3<sup>4</sup> and the FraSCAti platform that we present in this paper. The Open SOA web site provides a comprehensive list of available solutions.

Whereas the coverage by Tuscany of the different standards defined by the Open SOA collaboration around SCA is broader, FraSCAti focuses on the core features of SCA for Java in order to obtain a runtime kernel which is lighter and faster. Tuscany is implemented in pure Java, Newton is based on OSGi, whereas the implementation of FraSCAti is based on an extended SCA component model, itself derived from the Fractal model [9]. Compared to Tuscany, Newton and fabric3, the novelty of FraSCAti is to introduce reflective capabilities in the SCA programming model to allow dynamic introspection and reconfiguration of an SCA application and of the supporting platform. With FraSCAti, SCA assemblies and components can be introspected to query and discover their structure at runtime, assemblies can be modified in order to reconfigure the application for addressing new requirements, and components can be dynamically created and modified. These features open new perspectives for bringing agility to SOA and for the runtime management of SCA applications and of their supporting platform.

**Component models** Compared to the most well-known component models such as EJB, COM/.NET and CCM, SCA brings the notion of a software archi-

---

<sup>4</sup>[fabric3.codehaus.org](http://fabric3.codehaus.org)

ture and provides an architecture description language (ADL) for supporting this vision of a disciplined way of assembling software components. FraSCAti extends the SCA model with (componentized) reflective capabilities inherited from the Fractal [9] and FAC [7] models.

FraSCAti shares with component platforms such as OpenCOM [11], Hadas [12], Prism [13], LegORB [14] and K-Component [15] several characteristics such as reconfigurability. However, components with these models are finer-grained than SCA components with FraSCAti (they are more comparable to Fractal components, which are used in the implementation of FraSCAti). The target domain of these models are middleware platforms such as OpenORB [11] which is designed and implemented with OpenCOM. FraSCAti targets distributed SOA applications.

OSGi is another component model for SOA. Various platforms such as Equinox from Eclipse, Felix from Apache and Knopflerfish exist. OSGi has been extended recently, e.g. with the iPOJO [16] framework, to better support features such as composite components. OSGi is centered around Java, whereas SCA supports several language mappings. OSGi puts the focus on component lifecycle and discoverability, whereas SCA emphasizes an architecture-centered approach for deploying services. FraSCAti brings to SCA reconfiguration and reflective capabilities which go beyond those available in OSGi and iPOJO.

## 7 Conclusion

We have presented the FraSCAti platform for developing Service Component Architecture (SCA) [1] based distributed systems. SCA is a standard for distributed Service-Oriented Architectures (SOA). The novelty of FraSCAti is to bring runtime adaptation and manageability properties to SCA applications and their supporting platform. With FraSCAti, an SCA application can be introspected to discover at runtime its structure, dynamically modified to add new services or to remove existing ones, reconfigured to take into account new operating conditions. This flexibility and openness at the application level is also offered at the platform level.

To achieve this, FraSCAti is based on three original characteristics. First, FraSCAti adopts a component-based structure for the platform itself, using the same component model as for SCA applications. Second, FraSCAti extends, in an upward compatible fashion, the SCA component model with reflective capabilities. Third, FraSCAti exploits interception techniques for gluing (extending) SCA components with non-functional services, themselves programmed as SCA components. This results in a component-based structure that is highly modular, extensible and dynamically reconfigurable. Importantly, as suggested by our evaluation relative to the Tuscany open source reference SCA implementation, the built-in flexibility of the FraSCAti platform is not detrimental to performance.

As for future work, we plan to study the integration of new language mappings (notably OSGi). This will leverage the use of SCA at different levels of

system granularity. We also plan to extend the interception mechanism to obtain a full-fledged AOP (Aspect-Oriented Programming) [17] development technique for SCA. This can be achieved by extending the grammar of the SCA Assembly Language Descriptors with AOP notions such as a pointcut language. This will provide the ability to experiment with a tight integration of components, aspects and services.

## References

- [1] Beisiegel, M. et al, “Service Component Architecture,” Nov. 2007, [www.osoa.org](http://www.osoa.org).
- [2] *OSGi Service Platform Core Specification Release 4*, OSGi Alliance, Aug. 2005, [www.osgi.org](http://www.osgi.org).
- [3] D. Fournier, P. Merle, and al., “ANR SCOrWare Project: WP1 - SCA Platform Specification,” Apr. 2009, [www.scorware.org/](http://www.scorware.org/).
- [4] J. Kephart and D. Chess, “The Vision of Autonomic Computing,” *IEEE Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.
- [5] S. Sicard, F. Boyer, and N. D. Palma, “Using Components for Architecture-based Management: The Self-Repair Case,” in *Proc. of 30th Intl Conference on Software Engineering (ICSE 2008)*. ACM, 2008, pp. 101–110.
- [6] S. Bouchenak, N. D. Palma, D. Hagimont, and C. Taton, “Autonomic Management of Clustered Applications,” in *Proc. of the 2006 IEEE Intl Conference on Cluster Computing (CLUSTER’06)*, 2006.
- [7] N. Pessemier, L. Seinturier, L. Duchien, and T. Coupaye, “A Model for Developing Component-Based and Aspect-Oriented Systems,” in *Proc. of the 5th Intl Symposium on Software Composition (SC’06)*, ser. LNCS, vol. 4089. Springer, Mar. 2006, pp. 259–274.
- [8] B. Burke, “It’s the Aspects,” *Java’s Developer’s Journal*, Dec. 2003.
- [9] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, “The Fractal Component Model and its Support in Java,” *Soft. Pract. and Exp.*, vol. 36, no. 11-12, pp. 1257–1284, 2006.
- [10] S. Bagnier and J. Forrest, “SCOrWare Service Component Architecture (SCA) to make SOA a reality,” in *Proc. of 20th Intl Conference on Software and Systems Engineering and their Applications (ICSSEA’07)*, Dec. 2007.
- [11] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski, “The Design and Implementation of Open ORB v2,” *IEEE Distributed Systems Online Journal*, vol. 2, no. 6, November 2001.

- [12] I. Ben-Shaul, O. Holder, and B. Lavva, "Dynamic Adaptation and Deployment of Distributed Components in Hadas," *IEEE Transactions on Software Engineering* vol. 27 no 9, 2001.
- [13] S. Malek, M. Mikic-Rakic, and N. Medvidovic, "A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems," *IEEE Trans. Soft. Eng.*, vol. 31, no. 3, 2005.
- [14] F. Kon, J. Marques, T. Yamane, R. Campbell, and M. Mickunas, "Design, Implementation, and Performance of an Automatic Configuration Service for Distributed Component Systems," *Soft. Pract. and Exp.*, vol. 35, no. 7, pp. 667–703, Jun. 2005.
- [15] J. Dowling and V. Cahill, "The K-Component Architecture Meta-Model for Self-Adaptative Software," in *Proc. of Reflection'01*, ser. LNCS, vol. 2192. Springer, Sep. 2001, pp. 81–88.
- [16] C. Escoffier and R. Hall, "Dynamically Adaptable Applications with iPOJO Service Components," in *Proc. of the 6th Intl Symposium on Software Composition (SC'07)*, ser. LNCS, vol. 4829. Springer, Mar. 2007, pp. 113–128.
- [17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Longtier, and J. Irwin, "Aspect-Oriented Programming," in *Proc. of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, ser. LNCS, vol. 1241. Springer, Jun. 1997, pp. 220–242.