A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures

Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, Jean-Bernard Stefani

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **00**:1–26
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures

Lionel Seinturier[1,*], Philippe Merle[1], Romain Rouvoy[1], Daniel Romero[1], Valerio Schiavoni[2], Jean-Bernard Stefani[2]

[1] *Univ. Lille 1 & INRIA Lille – Nord Europe, LIFL UMR CNRS 8022, ADAM Project-Team, Villeneuve d'Ascq, France*
[2] *INRIA Grenoble – Rhône-Alpes, SARDES Project-Team, Montbonnot, France*

SUMMARY

The *Service Component Architecture* (SCA) is a technology-independent standard for developing distributed *Service-Oriented Architectures* (SOA). The SCA standard promotes the use of components and architecture descriptors, and mostly covers the life-cycle steps of implementation and deployment. Unfortunately, SCA does not address the governance of SCA applications and provides no support for the maintenance of deployed components. This article covers this issue and introduces the FRASCATI platform, a run-time support for SCA with dynamic reconfiguration capabilities and run-time management features. The article presents the internal component-based architecture of the FRASCATI platform, and highlights its key features. The component-based design of the FRASCATI platform introduces many degrees of flexibility and configurability in the platform itself and for the SOA applications it can host. The article reports on micro-benchmarks highlighting that run-time manageability in the FRASCATI platform does not decrease its performance when compared to the *de facto* reference SCA implementation: Apache TUSCANY. Finally, a smart home scenario illustrates the extension capabilities and the various reconfigurations of the FRASCATI platform. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

1. INTRODUCTION

The emergence of *Service-Oriented Architecture* (SOA) as an important design paradigm for online and Web-based services requires appropriate software platforms for the delivery, support and management of distributed applications. The *Service Component Architecture* (SCA) [4] fulfills this requirement with an extensive set of specifications defining an SOA infrastructure that is technology—*i.e.*, programming language and protocol—agnostic, and that reflects services as software components.

Although SCA is not the first approach that combines software components and services (see, for example, OSGi [47]), its technology independence, its support for hierarchical component composition, and its support for distributed configurations, makes it an interesting contender in the SOA world. Unfortunately, the SCA specification falls short of providing the required level of manageability and configurability that can be expected from a modern SOA platform. For example, the SCA specification specifies how to control the installation and the configuration of service components. But, it fails: *i)* to provide the required capabilities to manage at run-time

*Correspondence to: Lionel.Seinturier@inria.fr

Contract/grant sponsor: EU, ANR; contract/grant number: ICT FP7 IP SOA4All project, ARPEGE ITEmIS project.

a component configuration, *ii)* to provide the association of service components with platform-provided non-functional services (*e.g.*, transaction management, security management), *iii)* to control the execution of service components (*e.g.*, to handle on-line changes in configurations), and *iv)* to provide appropriate hooks for the management of the platform itself (to administer fault, performance, configuration, and security in distributed SOA environments). Overall, this challenge, which has been identified by Papazoglou et al. in [37] as key for service foundation, consists in providing a dynamically reconfigurable run-time architecture for SOA.

In this article, we introduce the FRASCATI platform for hosting 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, both at the business (application components) and at the 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 to implement both business-level service components conforming to the SCA specification and the FRASCATI platform itself.

A first version of the FRASCATI platform has been presented in a previous conference paper [57]. The present article reports on the second version of the FRASCATI platform with its improved component-based structure, and details the originality of the platform internal architecture. The platform is also put into perspective with a detailed usage scenario and examples of dynamic reconfiguration policies in the context of pervasive computing environments.

The remainder of this article is organized as follows. Section 2 introduces the SCA standard and discusses the configurability and manageability issues left open. Section 3 describes on the component-based architecture of the FRASCATI platform. Section 4 presents a detailed usage scenario which illustrates, among other things, the reconfiguration and extension capabilities of the platform. Section 5 reports on the implementation of the platform and on some performance measurements. Finally, Section 6 discusses related work while Section 7 concludes the paper and gives some directions for future work.

2. THE SCA STANDARD AND PLATFORM CHALLENGES

This section provides the necessary background material on the *Service Component Architecture* (SCA) and describes the key software engineering challenges related to the implementation of flexible component-based platforms for service-oriented architectures.

2.1. The SCA Standard

The *Service Component Architecture* (SCA) [3, 4] is a set of specifications for building distributed applications based on 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 defined and hosted by the *Open Service Oriented Architecture* (OSOA) collaboration (`http://www.osoa.org`) and standardized in the Open CSA section of the OASIS consortium (`http://www.oasis-opencsa.org`).

While SOA provides a way of exposing coarse-grained and loosely-coupled services that can be remotely accessed, SOA does not specify how these services should be implemented. SCA fills this gap by defining a component model for structuring service-oriented applications, whose first-class entities are *software components*. Components can be included in *composite* components. Components provide and require *services*, and expose *properties*. Required services are designated under the term *references*. References and services are either connected through *wires* or can be exposed at the level of the enclosing composite. In this last case, they are said to be *promoted*. Figure 1 provides a graphical notation for these concepts.

The SCA standard [4] is organized around four sets of specifications: *assembly language*, *component implementations*, *bindings*, and *policies*. They are meant to define a service architecture that is as independent as possible from underlying communication protocols and programming languages.

DOI: 10.1002/spe

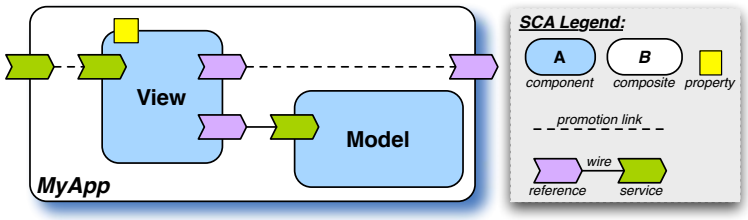

Figure 1. SCA Component-based Architecture.

Assembly Language. The assembly language configures and assembles components by using an XML-based grammar exploiting the notions introduced previously. For example, Listing 1 depicts the descriptor corresponding to the assembly of Figure 1. The composite `MyApp` (lines 1–20) encloses two components: `View` (lines 3–12) and `Model` (lines 13–18). In addition, `MyApp` exposes the service interface `run` (line 2), which is promoted from `View`. `View` and `Model` are implemented in Java by the classes `SwingGuiImpl` (line 4) and `ModelImpl` (line 14), respectively. `View` provides the service interface `run` (lines 5–7) and requires the service interface `model` (lines 8–10), which is provided by the component `Model` (lines 15–17). The explicit wiring between these two service interfaces is specified in line 19. Nonetheless, SCA also supports implicit wiring of services by making use of an *autowire* mechanism similar to the resolving mechanism of OSGi [47].

```
<composite name="MyApp">                                                    1
    <service name="run" promote="View/run"/>                                2
    <component name="View">                                                 3
        <implementation.java class="app.gui.SwingGuiImpl"/>                 4
        <service name="run">                                                5
            <interface.java interface="java.lang.Runnable"/>                6
        </service>                                                          7
        <reference name="model">                                            8
            <interface.java interface="app.ModelService"/>                  9
        </reference>                                                        10
        <property name="orientation">landscape</property>                   11
    </component>                                                            12
    <component name="Model">                                                13
        <implementation.java class="app.ctrl.ModelImpl"/>                   14
        <service name="model">                                              15
            <interface.java interface="app.ModelService"/>                  16
        </service>                                                          17
    </component>                                                            18
    <wire source="View/model" target="Model/model"/>                        19
</composite>                                                                20
```

Listing 1: Assembly language descriptor for Figure 1.

Component Implementations. The component implementation specifications define how SCA services are implemented. SCA does not assume that components will be implemented with a single technology, but rather supports either traditional programming languages, such as Java, C++, COBOL, C, script languages such as PHP, or advanced Web-oriented technologies such as Spring beans, EJB stateless session beans or BPEL orchestrations. This choice offers a broad range of solutions for integrating and implementing business services and therefore promotes independence from programming languages.

Binding Specifications. The binding specifications define how SCA services are accessed. This includes accesses to other SCA-based applications or to any other kind of service-oriented technologies, such as EJB [7] or OSGi [47]. Although Web Services are the preferred

DOI: 10.1002/spe

communication technology for SCA, they may not fit all needs. In some cases, technologies with different semantics and properties (*e.g.*, reliability, performance) may also be required. Consequently, SCA defines the concept 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, EJB, and JSON-RPC.

In addition to the concept of binding, SCA does not assume that a single *Interface Description Language* (IDL) is available. Rather, several languages are supported, such as *Web Service Description Language* (WSDL) and Java interfaces. These two forms of independence, from communication protocols and interface description languages, encourage interoperability with other middleware and SOA technologies.

Policy Frameworks. Non-functional properties may be injected into an SCA component via the concept of *policy set* (also known as *intent*) so that the component can declare the set of non-functional services that it depends upon. The SCA platform must then guarantee that these policies are enforced. Security and transactions [45] are two policies included in the SCA specifications. Yet, developers may need other types of non-functional properties (*e.g.*, persistence, logging). Therefore, the set of supported policy sets may be extended with user-specified values.

Overall, these openess principles offer a broad range of solutions for implementing SCA-based applications. Developers may think of incorporating new forms of programming language mappings (*e.g.*, SCA components programmed with Scala [41] or XQuery [6]), interface definition languages (*e.g.*, CORBA IDL [43]), communication bindings (*e.g.*, JBI [63], REST [23]) and non-functional properties (*e.g.*, timing, authentication). Therefore, supporting this variability in terms of technologies requires the definition of a modular infrastructure for deploying heterogenous application configurations.

2.2. SCA Platform Challenges

Two important challenges are to be met by SCA platform providers. First, if the SCA specifications offer at the application level all the mechanisms for declaring a broad range of variation points, nothing is said about the architecture of the platform that is supposed to implement these variations. To *design a platform that is flexible and extensible enough for smoothly accommodating and integrating these variations* is a first challenge.

Second, the SCA specifications focus on the task of describing the assembly and the configuration of the components that compose an SOA application. This assembly is used as input to instantiate and initialize the application. However, the SCA specifications do not address the run-time management of the application, which typically includes monitoring and reconfiguration. Furthermore, the SCA specification does not address either the run-time management of the platform itself. Yet, 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 or ubiquitous environments).

In the next section, we present the design of the FRASCATI platform which addresses these challenges. The first challenge is inherent to SCA, while the second one relates to our contribution.

3. AN OVERVIEW OF THE FRASCATI PLATFORM

FRASCATI provides a reflective view of middleware systems where the *run-time platform*, the *non-functional services*, and the *business applications* are designed and implemented with the same paradigm: the SCA component model.

Figure 2 depicts a general overview of the architecture of the FRASCATI platform. The topmost part, labeled Application Level, corresponds to end-user SCA applications, which are designed and implemented by developers. The four underlying layers correspond to the SCA infrastructure, which

DOI: 10.1002/spe

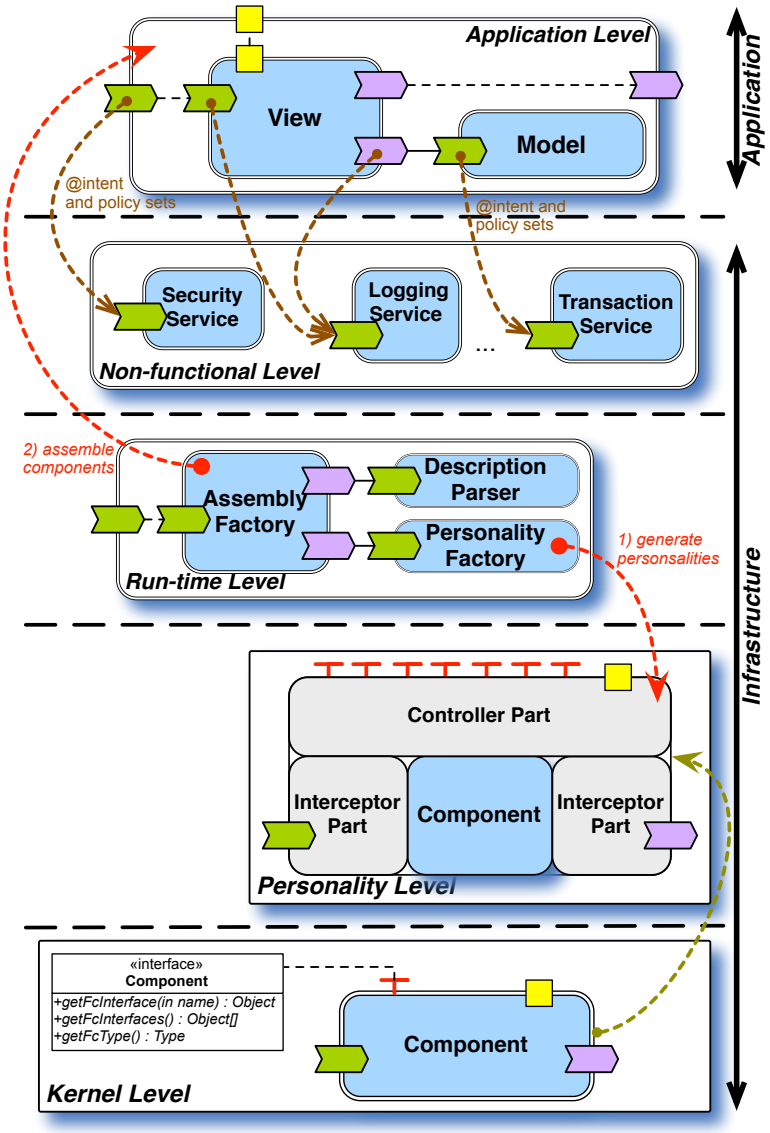

Figure 2. FRASCATI Platform Architecture.

is provided for deploying and hosting these applications. They are presented in details in the next sections. For each layer, we emphasize the reconfiguration capabilities we brought to SCA.

3.1. *Kernel Level:* Defining a Reflective Component Model

Technically, FRASCATI is built on top of the FRACTAL component model [11]. FRACTAL is a programming language independent component model, which has been specified for the construction of highly configurable software systems. The FRACTAL model combines ideas from three main sources: *software architecture*, *reflective systems*, and *distributed configurable systems*. From software architecture [58], FRACTAL inherits basic concepts for the modular construction of software systems, encapsulated components, and explicit connections between them. From reflective systems [60], FRACTAL inherits the idea that components can exhibit meta-level activities and reify through control interfaces part of their internal structure. From configurable distributed systems, FRACTAL inherits explicit component connections across multiple address spaces, and

DOI: 10.1002/spe

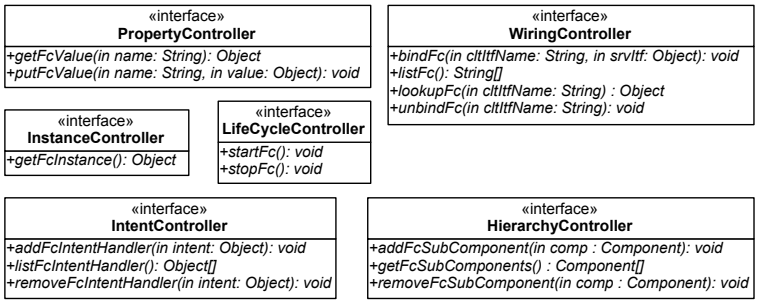

Figure 3. SCA Personality API.

the ability to define meta-level activities for run-time reconfiguration. The FRACTAL model has been used as a basis for the development of several kinds of configurable middleware platforms, and has been successfully used for building automated, architecture-based, distributed systems management capabilities, including deployment and (re)configuration management capabilities [1, 14, 18, 25], self-repair capabilities [8, 59], overload management capabilities [9], and self-protection capabilities [16]. The FRACTAL model is currently defined by a formal specification [40] based on the Alloy language [32].

The originality of FRACTAL is to enable the customization of the execution policy associated with a component. We will refer to a particular component execution policy implemented in the FRACTAL framework under the term of *component personality* (or for short *personality*). [11] demonstrates the programming of two personalities: JULIA, which is the reference personality for components with reconfiguration facilities, and DREAM, which is a personality for implementing message-oriented middleware solutions.

Component personalities are implemented by so-called *controllers* and *interceptors*. Each controller implements a particular facet of the personality, such as the *lifecycle management* or the *binding management*. Controllers expose their services through *control interfaces*. Interceptors modify and extend component behaviors when requests are received or emitted.

All FRACTAL components are equipped with a so-called `Component` control interface. The purpose of this interface is similar to the one of the `IUnknown` interface in the COM component framework [10] and allows the dynamic discovery of the capabilities and the requirements of a component. In other words, the `Component` control interface defines the identity of a component and plays, for components, a role similar to that of `Object` in object-oriented languages, such as Java or C#.

The API of the `Component` control interface is illustrated in the lower part of Figure 2. It includes methods for retrieving the component interfaces and the component type. This isolates the service interface of the FRACTAL component kernel. On top of this kernel, FRACTAL supports the modular definition of different personalities refining the execution policy associated with a component and providing different sets of control interfaces.

3.2. *Personality Level:* Implementing the SCA Standard

The term component personality refers to the structural and run-time characteristics associated to a component. This includes elements, such as how a component should be instantiated, started, wired with peers, activated, reconfigured, how requests should be processed, how properties should be managed, etc. In order to accommodate different application domains and execution environments, *e.g.* from grid computing, to Internet applications, to embedded systems, to wireless sensor networks, these characteristics can differ a lot. Contrary to component frameworks like EJB [7] where these meta-level activities are hard-coded in containers and can not be changed, the originality of the FRACTAL component framework is to enable the design and the programming of different component personalities.

DOI: 10.1002/spe

Designing a component personality therefore consists in defining the controllers which are needed to implement these meta-level activities. Six of these controllers are included in the FRASCATI component personality. Figure 3 describes their API.

Wiring Controller. This controller provides the ability, for each component, to query the list of existing wires (`lookupFc`), to create new wires (`bindFc`), to remove wires (`unbindFc`), and to retrieve the list of all existing wires (`listFc`). These operations can be performed at run-time.

Instance Controller. The SCA specifications define four 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). The instance controller then creates component instances according to one of these four modes. The `getFcInstance` method provided by this controller returns the component instance associated with the currently running thread.

Property Controller. This controller enables attaching a property—*i.e.*, a key-value pair to a component (`putFcValue`) and retrieving its value (`getFcValue`).

Hierarchy Controller. The SCA component model is hierarchical: a component is either *primitive* or *composite*. Composite components contain subcomponents that are themselves either primitive or composite. The management of this hierarchy is performed by the hierarchy controller, which provides methods for adding/querying/removing the subcomponents of a composite.

Lifecycle Controller. When dealing with multithreaded applications (the general case of distributed applications targeted by the SCA specifications), reconfiguration operations cannot be performed in an uncontrolled way. For example, modifying a wire while a client request is being served may lead to inconsistencies and wrong results or errors returned to clients. Therefore, the lifecycle controller ensures that reconfiguration operations are performed safely and consistently in isolation with client requests. This service controls the lifecycle of the components by strictly delimiting the time intervals during which reconfiguration operations can be performed and those during which application level requests can be processed. Method `stopFc` brings a component to a quiescent state to enable safe reconfiguration operations, whereas method `startFc` allows application (standard, non-meta-level) requests to be processed.

Intent Controller. This controller manages the non-functional services attached to an SCA component. Section 3.4 describes in details its functionalities.

Each of these controllers implements a particular facet of the execution policy of an SCA component. They are in turn implemented as FRACTAL components. These controllers need to collaborate to provide the overall execution logic to the hosted component instance. For example, the Instance Controller needs to query the Property Controller to retrieve the property values to be injected into created instances. As an other example, the Lifecycle Controller needs to trigger instance creations when initializing eagerly a component, and for that, queries the Instance Controller. Eager initialization is an SCA specific concept which states that components should be pre-instantiated before receiving any client request. The resulting collaboration scheme between controllers is captured in a software architecture which is illustrated in Figure 4. This software architecture constitutes the backbone of the implementation of the FRASCATI component personality. Applications targeted by SCA and FRASCATI are inherently distributed and multithreaded. Even if the personality level is made thread-aware, notably with the scope policies managed by the instance controller, threads are created and managed by the implementation of the communication protocol stacks, which are mentioned in Section 3.3. In addition we can mention that the controllers implemented as FRACTAL components use a simple built-in personality, close

DOI: 10.1002/spe

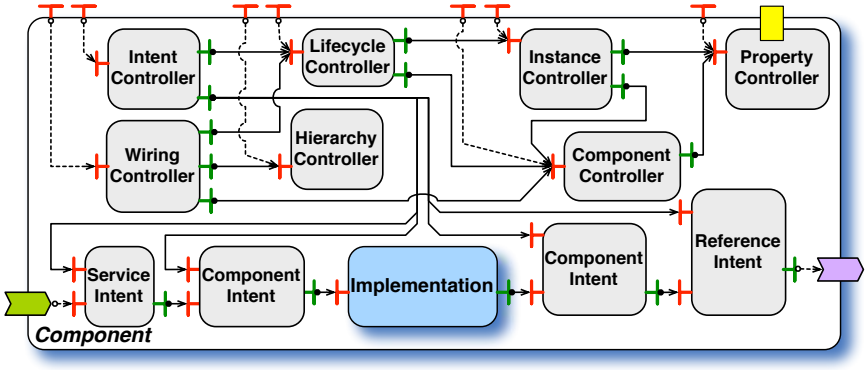

Figure 4. Personality Level.

to that the one implemented in JULIA with some basic reconfiguration capacities. The rationale for this choice lies in the fact that this is the execution policy of business components that may need adaptation, not the one of control components that implement this policy.

Reconfiguration Capabilities. Compared to the SCA assembly language that only allows the description of the initial configuration of an application, FRASCATI makes this configuration accessible and modifiable while the application is being executed. The following component elements can be changed at run-time: *wires*, *properties*, and *hierarchies*.

For example, based on the application illustrated in Figure 1, a reconfiguration scenario can consist in replacing the component `View` by a component `WebView`. This reconfiguration scenario would typically involve the following steps: *1)* stop the component (thus bringing it to a quiescent state), *2)* remove the existing wire, *3)* create a new component, *4)* wire with the new component, *5)* start the new component. Steps 1 and 5 are meant to ensure that the reconfiguration is consistent with respect to client requests. Stopping the component ensures that no incoming request is processed while the reconfiguration takes place. Note that stopping the component is not mandatory and that the corresponding steps (1 and 5) can be skipped if such a guarantee is not required. These reconfiguration steps are provided by the methods of the Lifecycle and Wiring controllers. Defining a particular reconfiguration policy is then a matter of invoking the methods defined by controllers.

Note that the personality level, being itself described as an assembly of components (see Figure 4), can be reconfigured. For example, the reconfiguration can consist in providing versions of the execution policy that check or not that the component is stopped (step 1 in the previous paragraph) before modifying the wires. In fact, by opening the personality level and making it reconfigurable, we do not impose a particular style or set of reconfiguration actions. Even though a default implementation is available with FRASCATI, it is up to the developer of the personality level to design and implement the operations s/he needs for her/his particular domain. Since the semantics for the reconfiguration actions can be changed, tasks such as verifying the consistency of a particular reconfiguration procedure are personality dependent. For example, given a particular personality, one can plan to perform some verifications with component behavioral protocols as what is done in the SOFA component model [49].

By providing a run-time 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 [33]. For example, Sicard et al. [59] show that the combination of wiring, hierarchy management, property, identity, and lifecycle are required meta-level capabilities in a component model to fully support self-repair capacities in a component-based distributed system. The same reconfiguration capabilities have been exploited to automate overload management in component-based cluster systems [9].

DOI: 10.1002/spe

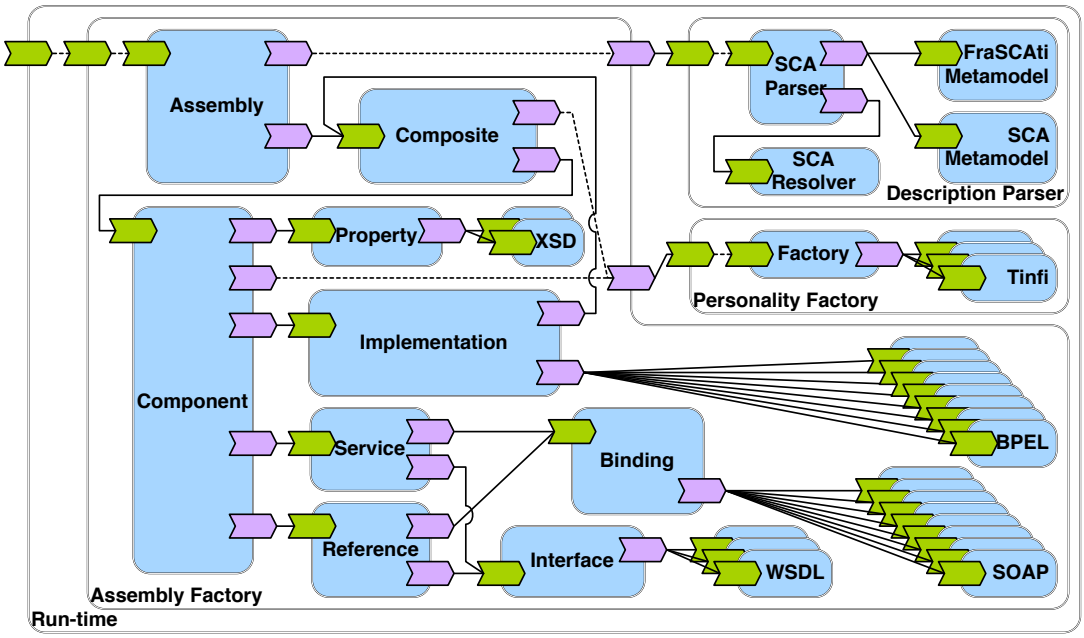

Figure 5. Run-time Level.

3.3. *Run-time Level:* Supporting the Execution of SCA Components

The run-time level of the FRASCATI platform is in charge of instantiating SCA assemblies and components. Three main components, which we present below, are defined. As illustrated by Figure 5, these are composite components implemented with the same personality as the one used for business components (see Section 3.2).

Description Parser is in charge of loading and checking the SCA assembly descriptor and creating the associated run-time model. This model conforms to a metamodel which is composed of two parts: the SCA Metamodel groups all the concepts defined by the SCA specifications, and the FraSCAti Metamodel describes some extensions, which are not included in the specifications. The isolation of metamodels in FRASCATI provides a mechanism for supporting original features, which are not defined by the SCA specification (*e.g.*, the UPnP binding or the FRACTAL implementation type) or integrating features proposed by other SCA platform vendors (*e.g.*, the REST binding defined by TUSCANY). The role of the SCA Parser is therefore to convert the XML-based description of the application into an EMF [61] model that conforms to the supported metamodels. The EMF model is then finalized in a completion step done by the SCA Resolver.

Personality Factory is in charge of generating the personality of the SCA components (described in Section 3.2). The nature of the code generated by the personality depends on the implementation type of the component (composite, Java, etc.). The FRASCATI platform supports two different generation techniques: bytecode and source code generation.

Assembly Factory visits the run-time model created by the Description Parser and creates the corresponding component assemblies. The Assembly Factory is organized according to the key concepts of the SCA model. Interestingly, this implementation choice offers a modular implementation of the interpretation process. For example, the components Property and Interface are wired to the supported property and interface description languages, while the component Implementation is wired to the supported component implementation types. Whenever necessary, the component Binding relies on the communication protocol plugins for exporting services and references.

DOI: 10.1002/spe

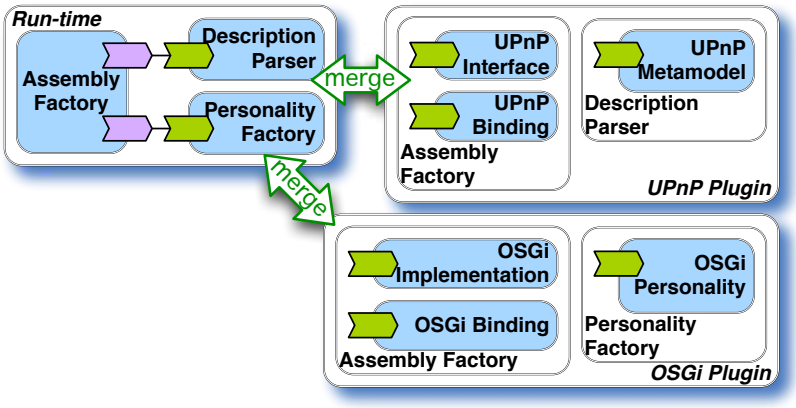

Figure 6. FRASCATI Auto-Configuration Process.

By default, the FRASCATI platform is bundled with the following plugins. This offers a broad range of possibilities for implementing distributed and heterogeneous SOA systems:

- *Interface Description Languages* (supported by the Interface component): Java, WSDL, UPNP [65] service description;
- *Property Description Languages* (supported by the Property component): Java, XSD;
- *Component Implementation Languages* (supported by the Implementation component): Java 5, Java Beans, Scala, Spring, OSGi, FRACTAL, BPEL, scripts based on the Java Scripting API;
- *Binding Technologies* (supported by the Binding component): either communication protocols, Java RMI, SOAP, HTTP, JSON-RPC, or discovery protocols, SLP [28], UPNP, or integrated technologies, OSGi, JNA.

In FRASCATI, the configuration of the run-time level is not hard-coded into the architecture description. Rather, the run-time level defines a flexible configuration process, which is inspired by the *extender* and *whiteboard* [46] design patterns of OSGi. FRASCATI therefore defines the concept of *platform plugins* as architecture fragments that are dynamically composed at runtime (cf. Figure 6). In particular, both the core platform and its various extensions are described as partial SCA architectures (so called *architecture fragments*), which are packaged as *platform plugins* composed upon application requirements. The platform configuration process therefore operates in two steps:

1. The FRASCATI bootstrap is executed with a minimal configuration including support for Java. The bootstrap looks for architecture fragments in the classpath. Whenever an architecture descriptor `frascati.composite` is found among the loaded bundles, the bootstrap merges the content of the descriptor with the core configuration in order to build the final configuration of the run-time platform.
2. Once all the architecture fragments have been merged, the bootstrap creates a new instance of the run-time platform based on the merged descriptor. This version of the platform is then used to instantiate and manage the business application.

Figure 6 depicts an example of configuration of the FRASCATI platform including the OSGi and UPNP plugins. The OSGi Plugin architecture fragment enables the interoperability between SCA and OSGi [47] technologies. The OSGi Implementation supports the implementation of an SCA component as an OSGi bundle, while the OSGi Binding retrieves the reference of an OSGi service from a bundle repository and binds it to an SCA application. Furthermore, the integration of the OSGi programming model is leveraged by the definition of a dedicated OSGi Personality, which handles the automatic discovery and binding of services. Similarly, the UPnP Plugin architecture

DOI: 10.1002/spe

fragment includes support in the Assembly Factory for UPnP service description, as well as UPnP communication protocols (including service discovery). As UPnP is not one of the standard technology promoted by the SCA specifications, the Description Parser requires to be extended with the meta-model dedicated to UPnP.

Reconfiguration Capabilities. In terms of reconfiguration, three main capabilities are brought by the run-time level: *binding management*, *dynamic instantiation* and *platform extension* with new plugins.

Binding between components is fully dynamic in FraSCAti: communication protocols are encapsulated as binding components, also known as *stubs* and *skeletons*, which encapsulate protocol specificities, such as message marshaling. Given that stubs and skeletons are themselves SCA components, the URI of a Web service (available as a property of these component) can be changed at run-time, thus reconfiguring the architecture of the distributed application.

Dynamic component instantiation is another original features of the FraSCAti platform for reconfiguring SCA systems. The Assembly Factory can be invoked at run-time to create new instances of components.

The fact that the run-time level is implemented as an assembly of SCA components brings to the platform all the reconfiguration properties emphasized in Section 3.2. For example, this enables the hot deployment of new plugins for the Personality Factory in order to tailor the platform to new usage conditions, unforeseen at startup.

3.4. *Non-Functional Level:* Injecting Middleware Services

The SCA Policy Framework specification [44] provides a mechanism for attaching metadata to component assemblies (Java 5 annotations in the case of the Java language mapping). These metadata elements influence the application behavior by triggering the execution of non-functional services. For example, the `@Confidentiality`, `@Integrity`, and `@Authentication` metadata ensure confidentiality, integrity, and authentication of service invocations, respectively. Some general purpose metadata like `@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 injecting and managing the non-functional services. This is left as a platform-specific issue.

FraSCAti proposes two innovative means of integrating non-functional services into SCA applications: *i) to implement non-functional services as regular SCA components* (note that these components may be composite and be the result of the assembling of several other components), and *ii) to provide an interception mechanism for connecting these non-functional services to application services*.

By using SCA components to implement non-functional services, we provide an integrated solution where there is only one paradigm for implementing both business and technical concerns.

By using interception to inject technical services within the business code, we keep these concerns clearly separated not only at design-time, but also at run-time. Figure 7 illustrates the mechanism. Each component implementing a non-functional service (*e.g.*, component `Security Service` in Figure 7) registers with a particular policy metadata. When an SCA assembly descriptor is parsed by the FraSCAti platform, the non-functional components are added on the services and/or references annotated with the registered metadata. When a client request is served, the request is first trapped and handled by the non-functional component that applies its logic. Note that, although the figure does not illustrate it, several non-functional components can be attached to the same service or reference. In this case, they are executed in the order in which they were attached. When no more non-functional component are available, the request is delegated to the business component. Non-functional components therefore act as filters on the business logic control flow.

Listing 2 illustrates the SCA assembly language descriptor corresponding to the example of Figure 7. The application level composite component `MyApp` is the same as the one of Figure 1. The line numbering has been preserved, but elements irrelevant for this current example have been omitted for clarity sake. The parameter `require` (see lines 2, 5, and 8) allows the developer

Prepared using speauth.cls
DOI: 10.1002/spe

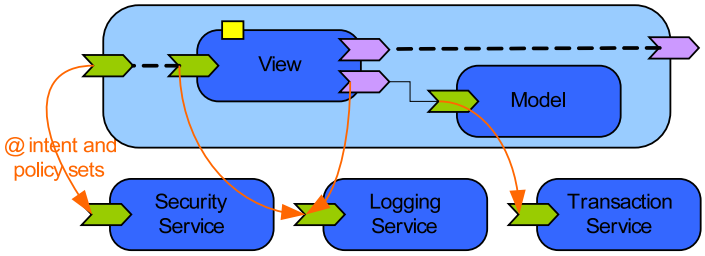

Figure 7. FRASCATI Interception Mechanism.

to specify the requirements in terms of non-functional properties for the associated service or reference.

```
<composite name="MyApp">                                                    1
    <service name="run" promote="View/run" require="SecurityService"/>      2
    <component name="View">                                                 3
        <service name="run" require="LoggingService">                       4
        <reference name="model" require="LoggingService">                   5
        <!-- ... -->                                                        6
    </component>                                                            7
    <component name="Model">                                                8
        <service name="model" require="TransacService">                     9
        <!-- ... -->                                                        10
    </component>                                                            11
</composite>                                                                12

<composite name="SecurityService">                                          14
    <service name="serviceitf">                                             15
    <!-- ... -->                                                           16
</composite>                                                                17

<!-- LoggingIntent and TransacIntent -->                                    19
```

Listing 2: Component-Based Implementation of Non-Functional Services.

Reconfiguration Capabilities. SCA components implementing non-functional services can be wired and unwired at run-time to the business component of the application using the API of the `Intent` controller presented in Figure 3. This mechanism is similar to the one available in component and *Aspect-Oriented Programming* (AOP) [34] frameworks, such as FAC [50] or JBoss AOP [13], where aspects can be woven and removed dynamically.

4. USE CASE: AUTONOMOUS HOME CONTROL SYSTEM

In this section, we illustrate the use of the FRASCATI platform in a smart home environment. In particular, we introduce an application for the dynamic management of multiple appliances (cf. Section 4.1) and we present how such an application can be realized as a distributed feedback control loop with the FRASCATI platform (cf. Section 4.2). We show that the definition of distributed feedback control loops provides a flexible architecture for dynamic adaptation. More precisely, Section 4.3 illustrates the reconfiguration capabilities brought by FRASCATI to SCA systems when new devices or modules appear, and Section 4.4 illustrates the reconfiguration of the FRASCATI platform itself with an energy consumption management scenario.

4.1. Scenario

A smart home generally refers to a house environment equipped with various types of *sensor nodes*, which collect information about the current temperature, occupancy (movement detection),

DOI: 10.1002/spe

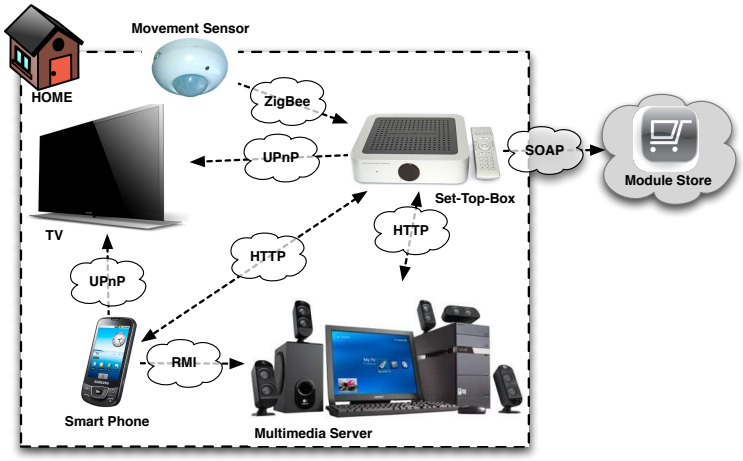

Figure 8. Overview of the Autonomous Home Control System.

noise level, and light states. In addition to that, *actuators* are also deployed within rooms to physically control appliances, such as lights, air conditioning, blinds, television, and stereo. In this environment, both sensors and actuators can be accessed from mobile devices owned by family members. The *Home Control System* (cf. Figure 8) deployed in such a smart home is able to detect changes in the surrounding environment and to reconfigure the mobile devices seamlessly. For example, the installation of a new television in the living room can automatically trigger the deployment of a remote controller facility on authorized mobile devices. Additionally, when the family members leave the room, the appliances can be automatically turned off in order to save energy.

4.2. Distributed Architecture

We design the above described *Home Control System* with the FRASCATI platform. The control system is an SCA application deployed on a *Set-Top Box* (STB) and several *mobile devices* (cf. Figure 9). Furthermore, the control system can provide an access to the *multimedia server*, which exposes various types of entertainment content (audio files, videos, photos, etc.) as services. Finally, the *Home Control System* can also connect to third party services, such as UPNP [65] devices and application stores available on the Internet.

We have chosen the autonomic computing paradigm, because the systems built using this approach have the capacity to manage and dynamically adapt themselves according to business policies and objectives [29, 48, 62]. In particular, the deployed systems exhibit properties, such as *self-configuration*, *self-optimization*, *self-healing*, and *self-protecting* [29, 33]. These properties are obtained by relying on the MAPE-K control loop principle, which has been defined by IBM and is composed of 4 phases [15, 33, 62]: *i) Monitoring* phase to collect, aggregate, and filter the events produced by the sensors and the system itself, *ii) Analysis* phase that consists in the processing of the information collected during the previous step, *iii) Planning* phase to determine the actions for executing the changes identified in the analysis, and *iv) Execution* phase to apply the plan determined in the previous phase using the adaptation capabilities of the system. These different steps share a *Knowledge base* that includes historical logs, configuration information, metrics, and policies [31].

In our design (cf. Figure 9), the STB device supports the analysis and planning parts of the feedback control loop, while the mobile devices enclose the monitoring and execution parts. This means that the collected contextual data are sent to the STB device, which decides upon necessary reconfigurations to apply. Additional contextual data can also be collected by the STB in order to assist the decision-making process. Once the STB has planned the adaptation to perform, it sends a reconfiguration script to the mobile device that provides detailed instructions for the reconfiguration.

DOI: 10.1002/spe

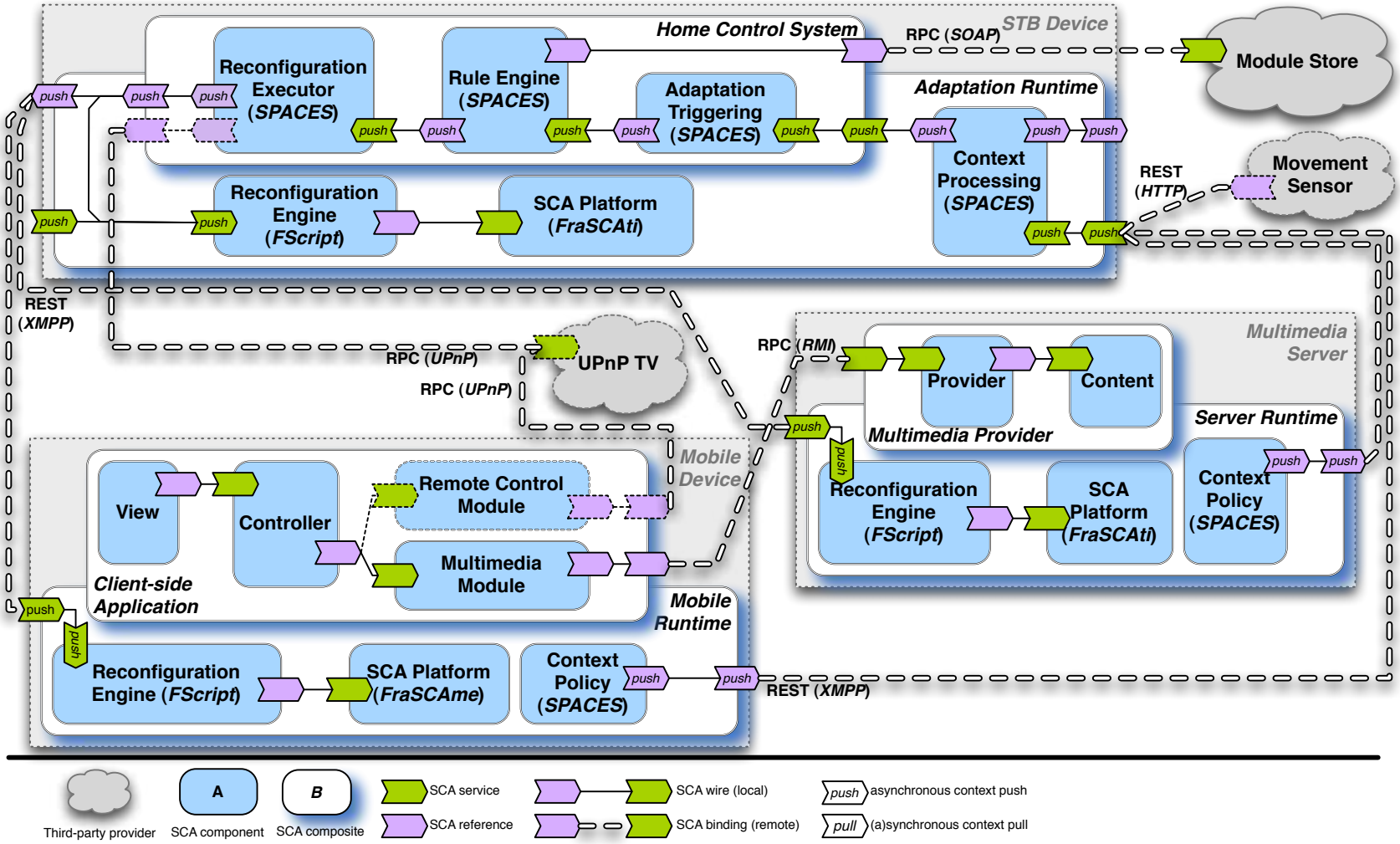

Figure 9. SCA Implementation of the Autonomous Home Control System.

Prepared using speauth.cls
DOI: 10.1002/spe

The monitoring and execution parts of the feedback control loop deployed on the mobile device are supported by the FRASCATI platform. In a similar way, we designed the content provider as an SCA application in order to reconfigure it when needed.

The distribution of the feedback control loop is based on the SPACES dissemination framework [55, 54], which applies the *REpresentational State Transfer* (REST) architectural style [23] to distribute the context information across the physical devices involved in the scenario. SPACES therefore reflects context information as representation-agnostic documents, which can be aggregated, combined, and transformed in order to infer adaptation situations. SPACES is integrated in the feedback control loop and the FRASCATI platform through a REST binding, which is an example of extension of the standard bindings defined by the SCA model. The use of SPACES for distributing the feedback control loop offers FRASCATI more flexibility in terms of communication protocols (*e.g.*, HTTP, FTP, XMPP) and resource representations (*e.g.*, XML, JSON) used to transfer the context information [54]. By integrating SPACES, the reconfiguration scripts executed to reconfiguration the SCA architecture are therefore automatically inferred from the context information collected by the home appliances.

Distributed Architecture Design and Implementation. When faced with a distributed architecture such as the one presented previously, several software lifecycle phases are to be addressed. This paragraph reports on the way design and implementation are conducted with SCA and FRASCATI, while the next one illustrates deployment. Two principles can be followed for design and implementation, either top-down, by starting with the architecture and then refining to obtain the components, or bottom-up, starting by the components and assembling them to obtain an architecture.

The SCA Assembly Language is the cornerstone of the approach and the architecture description language shared by all components whatever their implementation language is. This role puts a strong emphasis on the software architecture description and suggests that the assembly description should be put first in the design. Yet, the architecture may also be distributed, as this is the case in the previously described scenario, and the weak coupling principle of SOA guarantees that services should be kept as independent as possible from each others. Therefore, architects should start by designing services on each node of the distributed architecture. These services are designed with the SCA Assembly Language. If necessary the system may be decomposed in finer grained subsystems. Finally, leaf elements of the hierarchy are implemented as components.

The developer has to provide, first an architectural description in the SCA Assembly Language for each node of the distributed architecture and next, component implementations in a programming language such as Java. The reconfiguration capabilities are provided by the FRASCATI platform, as described in Section 3. In most cases, reconfiguration policies are composed of several steps (see for example the five steps scenario described in Section 3.2). These steps can be specified as services, and provided in the same way as regular business services.

Distributed Architecture Deployment. Deployment and runtime management are two important lifecycle phases, which need to be dealt with when setting up an SOA application. Yet, these issues are out of the scope of the SCA specifications and are left as platform specific issues. The SCA specifications envision cases where a composite component can be distributed across several nodes. However, no solution is specified to define how such a deployment should be performed. FRASCATI provides two original contributions for these topics. First, [21] defined a deployment metamodel and an Eclipse plugin with a graphical editor for defining deployment plans. This plugin generates deployment scripts for the DeployWare framework [25], which is a general purpose middleware framework for deploying and managing the lifecycle (downloading, installing, configuring, starting, stopping, etc.) of distributed applications. In the scenario previously described, the script specifies that the composite components on the STB and the multimedia server should be both deployed and that the composite on the STB should be started first. When started, the services provided by these components are advertised in the repositories of the protocol they are bound to (*e.g.*, UPnP, XMPP, RMI). Composite components on the mobile devices can be deployed independently. When

DOI: 10.1002/spe

started, they retrieve from the corresponding repositories the references to the services they require. More information about these steps can be found in [55]. The second contribution concerns runtime management. In particular, FRASCATI provides an interface that can be accessed remotely for performing basic operations, such as requesting the instantiation of a component on another node. This feature is illustrated in [39].

The distributed infrastructure described in Figure 9 is able to evolve over time to reflect changes in the surrounding environment. The following sections therefore provide some examples of reconfigurations which can be triggered and executed at run-time.

4.3. Dynamic System Evolution

In this situation, the family installs a new UPNP TV in the living room. The STB device detects this new appliance and queries the MODULE STORE available on Internet to check if any *Home Control* modules are available for this TV. Two modules are available from the application store for this UPNP TV: a *remote controller module* and an *energy saving module*. Therefore, the STB device downloads both modules from the Internet and generates two reconfiguration scripts.

The first one is sent to the reconfiguration engine of the mobile device. This script installs the remote controller module by downloading it from the STB device and reconfigures the client-side application to add and wire a Remote Controller Module component as described in Listing 3.

```
function installModule(app, description) {                                    1
    stop($app);                                                                2
    module = adl-new($description);                                            3
    add($app, $module);                                                        4
    controller = $app/child::controller;                                       5
    wire($controller/interface::module-tv, $module/interface::module);         6
    promote($controller/interface::tv, $app);                                  7
    bind($controller/interface::tv, "upnp");                                   8
    start($app);                                                               9
    return $app;                                                               10
}                                                                              11
```

Listing 3: Reconfiguration Script for Installing a New TV Module.

As described in Section 3.2, this script first brings the application in a quiescent state (line 2), and then creates an instance of the Remote Controller Module component (line 3 - the description of the module is stored in `description`). This new component is included in the application architecture (line 4) and wired to the Controller component (line 6). Finally, the reference tv is promoted to the enclosing component (line 7) and exposed as a UPnP binding in order to discover and connect to the UPnP TV (line 8). When these steps are completed, the execution of the application can be resumed (line 9).

The reconfiguration engine used in this scenario encloses the FSCRIPT interpreter [19] and exploits the reconfiguration controllers provided by the SCA personality level (cf. Section 3.2). FSCRIPT is a *Domain-Specific Language* (DSL) to program dynamic reconfigurations of FRACTAL architectures. FSCRIPT includes the FPATH notation to navigate inside FRACTAL architectures. Furthermore, the use of the FSCRIPT interpreter provides transactional guarantees during the execution of the reconfiguration.

The second script is executed by the reconfiguration engine available in the STB device to deploy additional reconfiguration rules for reducing the energy consumption of the TV. This script connects the Movement Sensor available in the environment to the STB device and deploys a new management rule within the Rule Engine component.

4.4. Energy Consumption Management

In this situation, the Movement Sensor detects that the living room is empty and triggers the feedback control loop for taking appropriate actions. The STB device therefore receives this contextual information and checks the status of the UPNP TV. If the TV is turned on, while nobody

DOI: 10.1002/spe

is in the room, the Home Control System will stop it automatically by invoking the UPnP TV service as described in the Groovy (`http://groovy.codehaus.org`) script of Listing 4.

```
class ShutdownTV implements Runnable {
    def setTv(ref) {
        tv = ref
    }

    def run() {
        if (tv.isOn()) {
            tv.turnOff()
}   }   }
```

Listing 4: Implementation in Groovy of the Reconfiguration Policy.

In practice, this script is an SCA component connected to the UPnP TV service. This means that the SCA component descriptor can enclose the behavior of the component as a script and can therefore be easily serialized over the network.

```
<component name="script">
    <service name="execute">
        <interface.java interface="java.lang.Runnable" />
    </service>
    <reference name="tv">
        <interface.upnp profile="upnp.tv.RemoteControlTV" />
        <binding.upnp />
    </reference>
    <implementation.script language="groovy">
        class ShutdownTV implements Runnable {
            def setTv(ref) {
                tv = ref
            }

            def run() {
                if (tv.isOn()) {
                    tv.turnOff()
        }   }   }
    </implementation.script>
</component>
```

Listing 5: Description of the Reconfiguration Policy Implemented in Groovy.

During the deployment of this script, a dependency towards the Script interpreter requires to be fulfilled. If the Component Factory of FraSCAti platform does not support this language (cf. Figure 2), the SCA component associated to the interpreter (so called Script Plugin) is dynamically deployed within the platform as a platform-level reconfiguration of the architecture (similar to the one described in Section 4.3).

Figure 10 illustrates the reconfiguration capabilities of FraSCAti in terms of SCA systems and platform. Both the architecture of the system and the platform are fully introspectable and reconfigurable. The figure provides a snapshot of the GUI tool, which assists administrators for that. The node `Multimedia Server` in the tree view on the left pane is the root composite of the SCA system, while the FraSCAti node is the root composite of the platform. The contextual menu displays some of the reconfiguration operations which are available: in this case the component can be stopped, renamed, and different types of binding (REST, Web Services, JSON-RPC, and RMI) can be added. This illustrates that both the system and the platform are fully introspectable and reconfigurable using the same artifacts and API.

5. IMPLEMENTATION AND EVALUATION

This section describes the implementation of the FraSCAti platform and reports on some performance measurements. These measurements show that the extra capabilities in terms of

DOI: 10.1002/spe

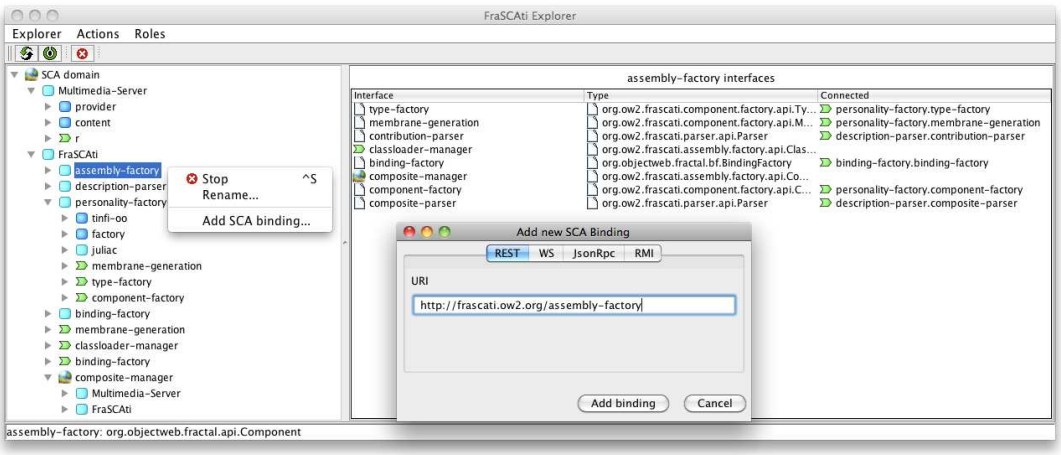

Figure 10. SCA Systems and Platform Reconfiguration Features with FRASCATI Explorer.

Table I. Code Footprint for Different Configurations of the FRASCATI Platform.

Configuration	Plugins	Code Footprint
Minimal		256KB
Core	Minimal + EMF parser	2.4MB
Dyn	Core + dynamic code generation & compilation	6.9MB
Full	Dyn + all plugins referenced in Section 3.3	25MB

extensibility and reconfigurability introduced in the FRASCATI platform do not penalize its performance when compared to the *de facto* current reference implementation of SCA. We also evaluate the discovery mechanism and the exchange of context data in the smart home scenario by using the SLP, UPNP, and REST bindings.

5.1. *Implementation Details*

The FRASCATI platform is implemented in Java and can be freely downloaded from `http://frascati.ow2.org`. Although the SCA specification is independent from programming languages, a platform that implements the specification has to be implemented in a particular programming language and is thus preferentially tied to a programming language. In our case, this is the Java language for its wide acceptance and its portability across various operating systems.

As presented in Section 3.3, FRASCATI is designed as a plugin-oriented architecture, which enables finely selecting the needed functionalities in terms of middleware features. This approach offers many different configurations of the platform in order to fit a broad range of user needs. Table I summarizes the code footprint of four commonly used configurations of the platform. The *minimal* configuration reflects the code footprint of the kernel level (described in Section 3.1), while the *core* configuration represents the footprint of the FRASCATI platform including support for Java technologies—*i.e.*, the configuration used by the FRASCATI bootstrap described in Section 3.3. By comparison, the code footprint of Apache TUSCANY Java SCA version 1.3.2, which is the *de facto* reference implementation of SCA is 55MB.

FRASCATI can run in two modes: as a *standalone application server* or *embedded as a service engine* in the OW2 PETALS (`http://petals.ow2.org`) JBI Enterprise Service Bus.

DOI: 10.1002/spe

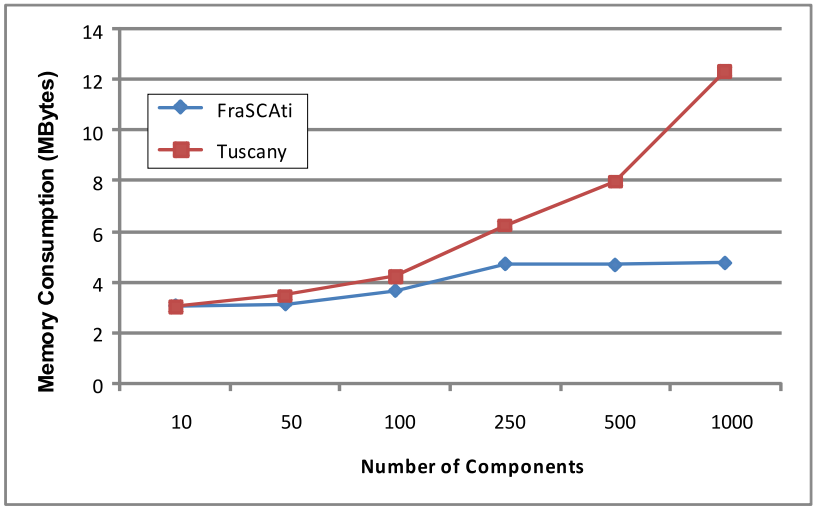

Figure 11. Memory Consumption (MBytes) per Number of Instantiated Components.

FRASCATI has been used to implement demonstrators in several application domains: service-oriented scientific computing [2], a new generation of collaborative development forge, a business-to-business platform, and system monitoring.

5.2. *Platform Performance Evaluation*

To evaluate the performances of our platform, we compare it to Apache TUSCANY Java SCA version, which is the *de facto* reference implementation of SCA. We devised a simple micro-benchmark to compare the memory consumption and the execution time of FRASCATI 1.3 in configuration `Full` with TUSCANY 1.6. The measurements have been conducted on an Intel Core Duo T2300 1.66GHz PC with 2GB of RAM running Windows XP and JDK 1.6.0_07. 64MB are allocated to the JVM.

The first series of measurements evaluate the cost of the FRASCATI infrastructure. Figure 11 compares the evolution of memory usage depending on the number of instantiated components. The assembly takes the form of a tree of components where each instantiated component shares the same implementation.

The second series of measurements concern the cost induced by FRASCATI when invoking a service. Figure 12 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, which are empty components, are reached and the invocations return. The purpose of this experiment is thus to measure the cost of invoking an SCA component over a local wire.

The third series of measurements concern reconfiguration. We measured the time taken by the reconfiguration scenario presented in Section 3.2. This scenario replaces a component by a peer in an existing architecture. This scenario contains the five following steps: *1)* stop the component, *2)* remove the existing wire, *3)* create a new component, *4)* wire with the new component, *5)* start the new component. We ran this scenario on the assembly used for the previous two series of measurements. In such a reconfiguration, replacing one component 1,000 times takes 0.35s. Given that reconfiguration is a feature which is available in FRASCATI only, we have not been able to perform some comparison with other platforms. Yet this measurement illustrates that the cost for reconfiguration is sufficiently low to be considered as usable in many application domains.

Analysis. These performance measurements show that the design of the FRASCATI platform provides comparable or better performance than the reference platform of the domain, while

DOI: 10.1002/spe

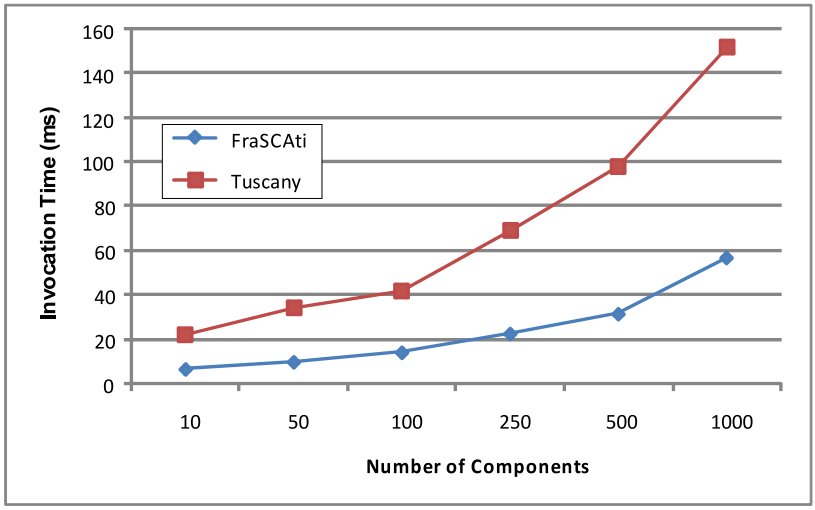

Figure 12. Invocation Time (ms) per Number of Instantiated Components.

providing introspection and reconfiguration capabilities. The relevance of these micro-benchmarks lies in the fact that they reveal the memory and the CPU consumed by the infrastructure. One of reasons that explain the difference in performance can be found in [30] where the authors compares the FRACTAL component model and TUSCANY. In particular, the authors observe that "*there are more internal calls using the* TUSCANY *framework*" than in the JULIA framework which is the reference implementation of FRACTAL. Notably, a single invocation requires "*In* TUSCANY *SCA, [...] a total of 12 additional internal calls [...] one of them is using Java reflection*" whereas "*in the* JULIA *implementation there are two objects*" between the caller component and the callee. Since FRASCATI is built on top of the JULIA personality of FRACTAL, these figures explain why the performance measured with our framework are better.

The size of assemblies studied in these benchmarks is meaningful for CPU and memory intensive applications such as those of the domain of distributed event-based simulation. For example, the OSA platform [52] uses components to simulate large scale peer-to-peer networks. Grid environments are used with nodes hosting simulations for up to 50,000 peers. Each peer is implemented as an assembly of 14 FRACTAL components. Recently, discussions have been conducted to move to service-oriented simulations. Since our SCA components are based on FRACTAL, FRASCATI is a good candidate for this move. In more traditional SOA applications, even though such large assemblies are not needed, we can witness that in the range of 10 to 100 components, our platform performs better. Once again, the main lesson learned from these experiments is that reconfiguration does not hinder performances, and that we can design a platform with similar or better performances, even with this extra functionality.

Optimizations. The above performance measurements show that the cost of invoking 1,000 components is of $56ms$ with FRASCATI. Most of this cost is due to the infrastructure that is set up to provide reconfiguration capabilities to the application. Nevertheless in some cases, such as performance-constrained applications, this overhead can be considered as too costly. We have then developed a series of optimizations aiming at skipping the part of the infrastructure which is in charge of reconfiguration and inlining the business code of the application in a single class. In this case, the application is no longer reconfigurable, but performances increase drastically. Measurements show that the cost of invoking 2,000,000 components drops to 1.2s. The technique, which is applied here, is the same than the one we have previously applied for embedded Java applications [51]. As demonstrated, the performances are even better than those of an object-oriented application. This is due to the fact that the merge algorithm we have developed inlines business code into one single class and removes all object instances but one. The JVM no longer

DOI: 10.1002/spe

Table II. Performance of SLP, UPnP and REST bindings in *ms*

Providers Configuration	Information Provider	Discovery Latency		Retrieval Latency		
		SLP	UPnP	Object	JSON	XML
a) 1 Local Provider	N/A	68	73	244	304	315
b) 1 External Provider	Laptop	91	111	292	252	261
c) 1 External Provider	N800	216	284	513	817	818
d) 2 External Providers	Laptop&N800	507	547	576	839	845
e) 2 External Providers	N800 A&B	736	769	641	989	1046

has to deal with instance creation and garbage collection. This merge is made possible by the fact that the application structure is a priori known with the assembly descriptor, which enumerates all the instances contained in the application.

5.3. *Use Case Performance Evaluation*

In addition to the performance evaluation of the platform, we have also measured the cost associated with the discovery and the exchange of contextual data in our smart home environment. We have tested the scenario using two Intel Core 2 Duo U7700 1.33 GHz with 2GB of RAM and Intel Pro wireless 3945ABG card running Windows XP and JDK 1.6.0_14. The mobile clients are two Nokia N800 Internet Table with 400 MHz, 128 MB of RAM, interface WLAN 802.11 b/e/g, Linux Maemo (kernel 2.6.21) and CACAOVM Java Virtual Machine 0.99.4. The `Minimal` configuration (see Table I on page 18) of the FraSCAti platform has been used. We have evaluated the RESTful bindings using XML, JSON, and the Java Object Serialization for context representations. We have also measured the UPnP and SLP bindings latency, which are based on the CyberLink for Java (`http://cgupnpjava.sourceforge.net`) and jSLP (`http://jslp.sourceforge.net`) libraries, respectively.

Table II summarizes the overhead observed for discovery via the UPnP and SLP bindings. These values include the discovery, instantiation, and configuration of the SCA wires. The given measures are the average of 10,000 successive tests, of which the first 100 were considered as part of a warm-up phase. The purpose of this warm-up phase is to avoid measuring the non deterministic cost introduced by the Java VM for loading and processing code. Regarding the discovery cost, we observe that it is possible to use this kind of bindings with a reasonable overhead ($68ms$ per message) in SCA applications. We also notice that the discovery latency due to the network is of approximately 25% when compared to the tests with a local provider (configuration *a*) and the laptop as a provider (configuration *b*). Although the measures with mobile devices (configuration *c*, *d* and *e*) demonstrate that we can discover services in a reasonable time, their use as providers considerably increases the discovery latency. This additional cost is mainly due to the limited processing capacity of these devices. As expected, SLP is more efficient than UPnP, which is a protocol with an higher-level complexity.

Table II reports on the costs of interactions once the data providers are discovered (last three columns in Table II about retrieval latency) in the feedback control loop. In these tests, we use RESTful bindings (cf. Section 3.3) and three different representations for information retrieval (Java Object Serialization, JSON, and XML) for the communication. As it can be seen, the exchange the information costs $244ms$ per message (configuration *a*). In the case of configurations including the mobile device (*c*, *d* and *e*), we again observe some additional overhead due to the constraints imposed by embedded devices.

DOI: 10.1002/spe

6. RELATED WORK

This section compares FRASCATI with other approaches in terms of SCA platforms, component models, and component adaptation techniques.

6.1. SCA Platforms

Several other 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 (http://tuscany.apache.org), NEWTON (http://newton.codecauldron.org), FABRIC3 (http://fabric3.codehaus.org). The Open SOA web site (http://www.osoa.org) 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 run-time kernel, which is lighter in terms of code footprint 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 FRACTAL [11]. Compared to TUSCANY, NEWTON, and FABRIC3, the novelty of FRASCATI lies in the introduction of 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 run-time, 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 run-time management of SCA applications and of their supporting platform.

6.2. Component Models

Compared to well-known component models, such as EJB [7], COM/.NET [10], and CCM [42], SCA brings the notion of a software architecture and provides an *Architecture Description Language* (ADL) for supporting this vision of a disciplined way of assembling components. FRASCATI extends the SCA model with reflective capabilities inherited from the FRACTAL [11] and FAC [50] models.

FRASCATI shares with component platforms, such as OPENCOM [17], HADAS [5], PRISM [38], LEGORB [35], K-COMPONENT [20], and the microkernel of JBOSS [24], several characteristics like introspection and 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 [17], which is designed and implemented with OPENCOM. FRASCATI targets distributed SOA applications. These applications are inherently heterogeneous in terms of communication protocols and implementation languages. The platform must then be able to integrate many different technologies. The architecture of the run-time level presented in Section 3.3 provides a solution for this.

SOFA [12] is another component model which shares several characteristics with FRACTAL and FRASCATI. Among other things, SOFA provides an aspect model for defining the control level of components. FRASCATI also deals with component and aspects, yet at a different level. As illustrated in Section 3.4, aspects with FRASCATI implement non-functional middleware services, which aim at enhancing the business functionalities of the applications. Aspects and micro-components with SOFA target the execution semantics of the components. In this respect, they play a role similar to that of the personality level presented in Section 3.2.

OSGi Declarative Services [47] is another service-oriented component model for SOA. Various platforms, such as EQUINOX from Eclipse, FELIX from Apache, and KNOPFLERFISH implement this component model. OSGi Declarative Services has been extended, *e.g.* with the iPOJO [22] framework, to support missing features, such as composite components. However, both OSGi and iPOJO are centered on Java, whereas SCA supports several language mappings. Furthermore, OSGi

DOI: 10.1002/spe

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. In addition, since FRASCATI supports component implementations with OSGi, an application can be entirely implemented with OSGi while benefiting from a software architecture described with the SCA assembly language. This allows benefiting from OSGi facilities such as component versioning.

6.3. Component Adaptation

MADAM [26] and MUSIC [56] are middleware frameworks supporting the dynamic reconfiguration of mobile applications at run-time. In particular, these approaches exploit the component paradigm to change automatically the structure of an application whenever its surrounding context change. While MADAM defined its own component model, MUSIC exploits OSGi Declarative Services to implement both application and middleware services. In particular, MUSIC compensates on the weaknesses of OSGi by defining a UML2-based component architecture of the application and the supporting platform, whose configuration is continuously optimized by an adaptation middleware. Although FRASCATI does not address the automatic adaptation of components depending on execution context, the run-time level of FRASCATI has been extended with an adaptation manager, which is responsible for resolving the components required for implementing the application. Such an issue has been addressed by the CAPPUCCINO project by implementing ubiquitous feedback control loops on top of FRASCATI (similar to the one described in Section 4). This experience has demonstrated the capability of extending FRASCATI to the domain of autonomous management of SCA applications [55]. This solution, compared to MUSIC, offers an explicit end-to-end view of the distributed architecture of the system, while the OSGi approach only focus on local architectures. The distributed nature of FRASCATI therefore opens up for more advanced reconfiguration scenarios where the platform can be itself reconfigured at run-time.

The work of Calton Pu et al. on the SWIFT [27] framework for building feedback control loops and addressing system reconfiguration share also some objectives with our work. As illustrated in Section 4, feedback control loops can be implemented on top of FRASCATI. Yet, our solution does not impose a particular model of loops, nor a particular model of reconfiguration. The reconfiguration capabilities offered by FRASCATI components are fine grained operations which can be composed to provide higher levels solutions.

DYNAMICTAO [36] is a middleware platform which shares some objectives with FRASCATI regarding reconfiguration. DYNAMICTAO is a CORBA compliant reflective ORB that supports dynamic reconfiguration and allows inspection and reconfiguration of its engine. Reification is achieved with the notion of a *component configurator*, which is an entity that holds dependencies between a certain component and other system components. Hooks are defined on configurators and allow attaching different strategies. One of the main differences is that reconfiguration capabilities with FRASCATI are not localised on a entity, but are spread over all the components of the architecture: each component provides its own reconfiguration capabilities and hooks are defined for all components with the interception mechanisms presented in Section 3.4. LEGORB [35] builds on DYNAMICTAO and provides a component-based ORB with the ability to load just enough components to provide the middleware services required by the application. The idea is similar to the one used with FRASCATI for configuring the run-time level (see Section 3.3). GAIA [53] is another project exploiting DYNAMICTAO to provide a component-based operating system for assisting in the development of ubiquitous computing applications based on the notion of an active space. In both cases of LEGORB and GAIA, FRASCATI differs by the fact that flexibility is applied for configuring an SOA environment and not just addressing a particular ORB technology, but federating a set of heterogeneous communication and middleware technologies.

DOI: 10.1002/spe

7. CONCLUSION

We have presented the FRASCATI platform for developing *Service Component Architecture* (SCA) [4] based distributed systems. SCA is a standard for distributed *Service-Oriented Architectures* (SOA). The novelty of FRASCATI is to bring run-time adaptation and manageability properties to SCA applications and their supporting platform. As stated by [37], this is a key challenge in SOA research. With FRASCATI, an SCA application can be introspected to discover at run-time its structure, modified dynamically to add new services, reconfigured to take into account new operating conditions. Both the component-based architecture of the system and the binding of this system to external services can be reconfigured. This flexibility and openness at the application level is also offered at the platform level.

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 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.

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 extend the FRASCATI platform towards three domains: *aspect-oriented programming*, *cloud computing*, and *embedded systems*. Concerning *Aspect-Oriented Programming* (AOP) [34], we already have, in relation with the interception mechanism (see Section 3.4), some notions like *join points* and *advice code* (components implementing non-functional services). In order to obtain a fully-fledged AOP development technique for SCA, we need to extend the grammar of the SCA assembly language with additional concepts, such as a pointcut language, weaving directives like ordering, and code injection techniques (also know as inter-type declaration in the AOP domain). This will provide the ability to experiment the tight integration of components, aspects, and services. The second domain of extension for FRASCATI is cloud computing. SCA provides a good starting point for providing a programming and computing model for *Software as a Services* (SAAS) and middleware *Platform as a Services* (PAAS). The notions of components and services are general enough to accommodate a broad range of cases for the applications, which will be hosted on these clouds. The reconfigurability features of FRASCATI open up interesting paths for experimenting with elastic middleware environments where one needs to manage services deployed on several thousands of nodes and which must be able to cope with changes in the topology due to node failures or activity peaks. For that, we plan to investigate the coupling of FRASCATI with existing cloud platforms, such as *Google Application Engine* (GAE) and with middleware services, such as deployment and load balancing to provide a fully-fledged *Platform as a Service* (PAAS) of our own based on FRASCATI. Finally, in addition to large-scale applications, such as the ones for clouds, we believe that software components and services are also relevant for tiny applications for embedded systems. As demonstrated by [64], SCA is also suitable for developing services within *Wireless Sensor Networks* (WSN). Many differences exist in terms of programming languages, protocols, and bindings between applications for these environments and the ones we are currently addressing with FRASCATI. Yet, we believe that the openness of FRASCATI and its plugin-oriented architecture are good candidates for experimenting with extensions specific to the WSN domain. In the end, we believe that, much like what is being done in software engineering with *Software Product Lines* (SPL), we can define a product line for middleware platforms that will be able to address a broad range of application domains from clouds, to Internet applications, to mobile applications, and to embedded applications.

REFERENCES

1. T. Abdellatif, J. Kornas, and J. Stefani. J2EE Packaging, Deployment and Reconfiguration Using a General Component Model. In *Proceedings of the 3rd International Working Conference on Component Deployment (CD'05)*, volume 3798 of *LNCS*. Springer, 2005.

DOI: 10.1002/spe

2. S. Bagnier and J. Forrest. SCOrWare Service Component Architecture (SCA) to make SOA a reality. In *Proceedings of the 20th International Conference on Software and Systems Engineering and their Applications (ICSSEA'07)*, Dec. 2007.
3. Beisiegel, M. et al. Service Component Architecture: Building Systems using a Service Oriented Architecture. white paper 0.9, BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, Sybase, Nov. 2005. http://www.iona.com/devcenter/sca/SCA_White_Paper1_09.pdf.
4. Beisiegel, M. et al. Service Component Architecture, Nov. 2007. `http://www.osoa.org`.
5. I. Ben-Shaul, O. Holder, and B. Lavva. Dynamic Adaptation and Deployment of Distributed Components in Hadas. *IEEE Transaction on Software Engineering*, 27(9), 2001.
6. S. Boag, D. Chamberlin, M. F. Fernndez, D. Florescu, J. Robie, and J. Simon. *XQuery 1.0: An XML Query Language*. W3C Recommendation, Jan. 2007. `http://www.w3.org/TR/xquery`.
7. S. Bodoff, E. Armstrong, J. Ball, and D. Carson. *The J2EE Tutorial*. Addison-Wesley, 2nd edition, June 2004.
8. S. Bouchenak, F. Boyer, S. Krakowiak, D. Hagimont, A. Mos, N. D. Palma, V. Quéma, and J. Stefani. Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters. In *Proceedings of the the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. IEEE Computer Society, 2005.
9. S. Bouchenak, N. D. Palma, D. Hagimont, and C. Taton. Autonomic Management of Clustered Applications. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'06)*, 2006.
10. D. Box. *Essential COM*. Addison-Wesley, 1998.
11. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL Component Model and its Support in Java. *Software Practice and Experience (SPE)*, 36(11-12):1257–1284, 2006.
12. T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Proceedings of the 4th International Conference on Software Engineering, Research, Management and Applications (SERA'06)*, pages 40–48, Dec. 2006.
13. B. Burke. It's the Aspects. Java's Developer's Journal, Dec. 2003.
14. D. Caromel, A. di Costanzo, and C. Delbé. Peer-to-Peer and Fault-Tolerance: Towards Deployment-Based Technical Services. *Future Generation Computer Systems*, 23(7), 2007.
15. N. Chase. An Autonomic Computing Roadmap, 2004.
16. B. Claudel, N. D. Palma, R. Lachaize, and D. Hagimont. Self-protection for Distributed Component-Based Applications. In *Proceedings of the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'06)*, number 4280 in LNCS. Springer, 2006.
17. G. Coulson, G. Blair, P. Grace, F. Taiani, A. Jolla, K. Lee, J. Ueyama, and T. Silvaharan. A Generic Component Model for Building Systems Software. *ACM Transactions on Computer Systems*, 26(1):1–42, Feb. 2008.
18. P. David, M. Léger, H. Grall, T. Ledoux, and T. Coupaye. A Multi-stage Approach for Reliable Dynamic Reconfigurations of Component-Based Systems. In *Proceedings of the 8th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'08)*, volume 5053 of *LNCS*, 2008.
19. P.-C. David, T. Ledoux, M. Léger, and T. Coupaye. FPath and FScript: Language support for navigation and reliable reconfiguration of FRACTAL architectures. *Annales des Télécommunications*, 64(1-2):45–63, Jan. 2009.
20. J. Dowling and V. Cahill. The K-Component Architecture Meta-Model for Self-Adaptative Software. In *Proceedings of Reflection'01*, volume 2192 of *LNCS*, pages 81–88. Springer, Sept. 2001.
21. S. Drapeau, V. Zurczak, D. Fournier, P. Merle, A. Hurault, D. Belaid, S. Tata, and M. Dutoo. ANR SCOrWare Project: WP2 - Conception and Development Tools Specifications, Mar. 2009. `http://www.scorware.org`.
22. C. Escoffier and R. Hall. Dynamically Adaptable Applications with iPOJO Service Components. In *Proceedings of the 6th International Symposium on Software Composition (SC'07)*, volume 4829 of *LNCS*, pages 113–128. Springer, Mar. 2007.
23. R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
24. M. Fleury and F. Reverbel. The JBoss Extensible Server. In *Proceedings of the 4th ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'03)*, volume 2672 of *LNCS*, pages 344–373. Springer, June 2003.
25. A. Flissi and P. Merle. A Generic Deployment Framework for Grid Computing and Distributed Applications. In *OTM Confederated International Conferences, Grid computing, High Performance and Distributed Applications (GADA 2006)*, volume 4276 of *LNCS*. Springer, 2006.
26. K. Geihs, P. Barone, F. Eliassen, J. Floch, R. Fricke, E. Gjørven, S. Hallsteinsen, G. Horn, M. U. Khan, A. Mamelli, G. A. Papadopoulos, N. Paspallis, R. Reichle, and E. Stav. A comprehensive solution for application-level adaptation. *Software Practice and Experience (SPE)*, 39(4):385–422, 2009.
27. A. Goel, D. S. C. Pu, and J. Walpole. Swift: A feedback control and dynamic reconfiguration toolkit. In *Proceedings of the 2nd USENIX Windows NT Symposium*, Aug. 1998.
28. E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. RFC 2608 (Proposed Standard), June 1999.
29. S. Hariri, B. Khargharia, H. Chen, J. Yang, Y. Zhang, M. Parashar, and H. Liu. The Autonomic Computing Paradigm. *Cluster Computing*, 9(1):5–17, 2006.
30. P. Hnetynka, L. Murphy, and J. Murphy. Comparing the service component architecture and fractal component model. *The Computer Journal*, 2010.
31. IBM. An Architectural Blueprint for Autonomic Computing, 2004. `http://www-03.ibm.com/autonomic/pdfs/ACBP2_2004-10-04.pdf`.
32. D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, Apr. 2002.
33. J. Kephart and D. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, Jan. 2003.
34. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, pages 220–242. Springer, June 1997.

DOI: 10.1002/spe

35. F. Kon, J. Marques, T. Yamane, R. Campbell, and M. Mickunas. Design, Implementation, and Performance of an Automatic Configuration Service for Distributed Component Systems. *Software Practice and Experience (SPE)*, 35(7):667–703, June 2005.
36. F. Kon, M. Romain, P. Liu, J. Mao, T. Yamane, L. Magalhaes, and R. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'00)*, Apr. 2000.
37. M. Papazoglou and P. Traverso and S. Dustdar and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer*, 40(11):64–71, Nov. 2007.
38. S. Malek, M. Mikic-Rakic, and N. Medvidovic. A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems. *IEEE Transaction on Software Engineering*, 31(3), 2005.
39. R. Melisson, P. Merle, D. Romero, R. Rouvoy, and L. Seinturier. Reconfigurable run-time support for distributed service component architectures. In *Tool Demo at the 25th ACM/IEEE International Conference on Automated Software Engineering (ASE'10)*, pages 171–172, Sept. 2010.
40. P. Merle and J.-B. Stefani. A Formal Specification of the Fractal Component Model in Alloy. Technical Report RR-6721, INRIA, Nov. 2008.
41. M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language (2. Edition). Technical Report LAMP-REPORT-2006-001, EPFL, 2006.
42. OMG. *CORBA Component Model*, Feb. 1999. Document ad/99-02-05.
43. OMG. *Common Object Request Broker Architecture (CORBA/IIOP)*, Jan. 2008. Document formal/2008-01-04.
44. Open SOA. *SCA Policy Framework*, Mar. 2007. Version 1.0.
45. Open SOA. *SCA Transaction Policy*, Dec. 2007. Version 1.0.
46. OSGi Alliance. *Listeners Considered Harmful: The Whiteboard Pattern*, Aug. 2004.
47. OSGi Alliance. *OSGi Service Platform Core Specification Release 4*, Aug. 2005.
48. M. Parashar and S. Hariri. Autonomic Computing: An Overview. In *International Workshop on Unconventional Programming Paradigms (UPP'04)*, volume 3566 of *LNCS*, pages 257–269, 2005.
49. P. Parizek, F. Plasil, and J. Kofron. Model checking of software components: Combining java pathfinder and behavior protocol model checker. In *Proceedings of the 30th IEEE/NASA Software Engineering Workshop (SEW'30)*, pages 133–141, Jan. 2007.
50. N. Pessemier, L. Seinturier, L. Duchien, and T. Coupaye. A Model for Developing Component-Based and Aspect-Oriented Systems. In *Proceedings of the 5th International Symposium on Software Composition (SC'06)*, volume 4089 of *LNCS*, pages 259–274. Springer, Mar. 2006.
51. A. Plsek, F. Loiret, P. Merle, and L. Seinturier. A Component Framework for Java-based Real-Time Embedded Systems. In *9th ACM/IFIP/USENIX International Middleware Conference (Middleware'08)*, volume 5346 of *LNCS*, pages 124–143. Springer, Dec. 2008.
52. K. Ribault, F. Peix, J. Monteiro, and O. Dalle. Osa: an integration platform for component-based simulation. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques (SIMUTools'09)*, Mar. 2009.
53. M. Roman, F. Kon, and R. Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online*, 2(5), 2001.
54. D. Romero, R. Rouvoy, L. Seinturier, and P. Carton. Service Discovery in Ubiquitous Feedback Control Loops. In *Proceedings of the 10th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'10)*, volume 6115 of *LNCS*, pages 113–126. Springer, June 2010.
55. D. Romero, R. Rouvoy, L. Seinturier, S. Chabridon, D. Conan, and N. Pessemier. *Enabling Context-Aware Web Services: Methods, Architectures, and Technologies*, chapter Enabling Context-Aware Web Services: A Middleware Approach for Ubiquitous Environments, pages 113–135. Chapman and Hall/CRC, July 2009.
56. R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz. *Software Engineering for Self-Adaptive Systems (SEfSAS)*, volume 5525 of *LNCS*, chapter MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments, pages 164–182. Springer, 2009.
57. L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *Proceedings of the 6th IEEE International Conference on Service Computing (SCC'09)*, pages 268–275, Sept. 2009.
58. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
59. S. Sicard, F. Boyer, and N. D. Palma. Using Components for Architecture-based Management: The Self-Repair Case. In *Proceedings of 30th International Conference on Software Engineering (ICSE'08)*, pages 101–110. ACM, 2008.
60. B. Smith. Reflection and Semantics in Lisp. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'84)*, pages 23–35, 1984.
61. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework (2nd Edition) (Eclipse)*. Addison-Wesley Longman, Amsterdam, 2nd revised edition, January 2009.
62. R. Sterritt, M. Parashar, H. Tianfield, and R. Unland. A Concise Introduction to Autonomic Computing. *Advanced Engineering Informatics*, 19(3):181–187, July 2005.
63. Sun Microsystems. *Java Business Integration (JBI) 2.0*, 2002. JSR 312.
64. A. Taherkordi, F. Loiret, A. Abdolrazaghi, R. Rouvoy, Q. Le-Trung, and F. Eliassen. Programming Sensor Networks Using REMORA Component Model. In *Procedings of the 6th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS'10)*, volume 6131 of *LNCS*, pages 45–62. Springer, June 2010.
65. UPnP Forum. *UPnP Device Architecture*, Oct. 2008. Version 1.1. `http://www.upnp.org`.

DOI: 10.1002/spe