A Component-Based Reconfigurable Platform for SOA

Mohsen Saberi
Department of Engineering, Bozorgmehr University of Qaenat,
Qaen, South Khorasan, Iran

Email: saberi [AT] pbuqaen.ac.ir

Abstract— **Service Component Architecture (SCA) is a standard for developing independent technology distributed Service Oriented-Architecture (SOA). SCA standard proposes using components and also architecture descriptors. The standard also covers the life cycle stages of implementation and deployment. One of the SCA problems is that it does not address the SCA application management and support of deployed components. This article covers these subjects and defines a platform for applications that support run-time management and distribution capabilities. Evermore the component-based design of the proposed platform provides a high degree of flexibility and functionality in the platform itself. This platform can be a good context for SOA applications. The results show that in comparison with the architecture of SCA, run-time management of the platform does not have a negative impact on its performance.**

Keywords- Middleware; SOA; Component; Reconfiguration; Distribution; Load Balancing.

I. INTRODUCTION

The emergence of Service Oriented-Architecture (SOA), as an important model for online and web-based services, needs a software framework for delivery, support and management of distributed applications. Service Component Architecture (SCA) [1] has created the conditions with an extensive set of specifications and definitions on a SOA infrastructure. This set of definitions are independent of technology, programming language and protocol.

Although SCA is not the first approach that combines software components and services, and there are other approaches such as OSGi [2], but the independence of the approach from technology and its support for combining hierarchical components causes this approach to be attractive in the SOA world. Unfortunately, SCA does not address the management, configurable and scalability that are expected of a modern SOA platform. For example, SCA specification defines how the installation and configuration of the components of services are controlled, but it says nothing in the following discussion: (a) providing the functionality needed to manage the runtime configuration of components, (b) providing the facilities needed for management of the platform itself, and (c) controlling the execution of service components (for example, for online changes to the configurations). In general, it seems SOA needs a dynamic and runtime reconfigurable architecture for issuing these challenges, identified by Papazoglou et al. [3]

This paper introduces an architecture to host SCA applications. In comparison with the existing architectures, the main goal of this architecture is to address the reconfigurable issues mentioned above and to achieve the systematic management of a system. These issues must be solved in both application components and the platform (nonfunctional services, communication protocols, etc.). This is possible through the expansion of SCA component model and then using this model to implement the components of the service of programs and the platform itself.

II. SCA STANDARD AND ITS RELATED ISSUES

This section focuses on SCA and key challenges of software engineering in the context of implementing flexible component.

SCA [1,4] is a set of specifications for building distributed applications based on SOA and component-based software engineering (CBSE). This model has been constructed by a group of various companies, including BEA, IBM, IONA, Oracle, SAP, Sun and TIBCO.

While SOA offers a way to provide great services, separate from each other and accessible remotely, but it does not specify how these services should be implemented. SCA fills this gap by defining a component model. This model is useful for building service-oriented applications. The most important entities of these programs are their software components. The components can be used together to create composite components. Components need and provide the services. The required services are named reference. References and services either are connected to each other through the facilities called wires or included in their composite and promoted for external use. Figure 1 shows a symbol for each of these concepts.

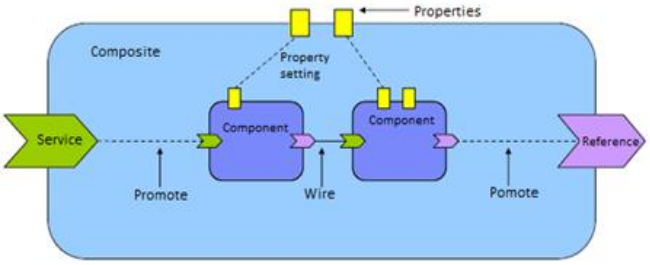

Figure 1. SCA Architechture concepts

The SCA standard [1] is organized based on four main elements: assembly language, component implementations, bindings and policies. These elements can be used to help define a service-oriented architecture that is independent of communication protocols and programming language, as much as possible.

Assembly language: this language assembles the configuration and communication components, in assistance with a grammar-based XML. For example, Figure 2 depicts the descriptor related to the communications of Figure 1. Composite MyApp (lines 1-20) covers two components View (lines 3-12) and Model (lines 13-18). In addition, MyApp service interface "run" (line 2) is located inside the component View. View and Model have been implemented in Java and in classes with names SwingGuiImpl (line 4) and ModelImpl (line 14). View provides the service interface "run" (lines 5-7) and requires the service interface Model (lines 15-17). The explicit connection between the interfaces of these two services can be seen in line 19. However, SCA, like OSGi, supports the autowire mechanism for implicit wiring of services [2].

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

Figure 2. A sample descriptor

Component Implementations: This element defines how to implement SCA services. SCA assumes nothing about technologies used for component implementations, but also supports programming languages, like Java, C ++, COBOL, C, as well as scripting languages, such as PHP and advanced web oriented technologies, like Spring beans, EJB stateless beans or BPEL orchestrations. The choice between wide ranges of solutions promotes for the integration and implementation business services and thus the independence of programming languages.

Binding Specifications: This element specifies how the SCA services should be available. This includes access to other programs based on SCA or any other kind of service-oriented technologies such as EJB [5] or OSGi [2]. Although Web services are the preferred option for SCA communication technology, but this option may not meet all the needs of the system. In some cases, technologies with different properties (e.g. in terms of reliability or performance) may be required. As a result, SCA defines the concept of connection: 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 not focus on a special interface description language (IDL), and instead, support different languages, such as Web Services Description Language (WSDL) and Java Interface. The independence from communication protocols and interface description languages creates corporations with another middleware SOA technologies.

Policy Frameworks: nonfunctional properties can be added to an SCA component by the concept of policy set (or intent), so that it can declare the set of nonfunctional parameters that the service depends on. After that, the SCA platform should ensure that these policies are implemented. Security and transactions [6] are two policies that are in the SCA specification. However, developers may require to have other types of nonfunctional properties (e.g. persistence or logging). Therefore, the set of policy set may be extended by the user-specified values.

In general, these principles offer a wide range of different solutions for the implementation of SCA-based applications. Developers can think about the combination of new forms of mapping programming languages (e.g. the components of the SCA written by Scala [7] or XQuery [8]), language interface definition (such as CORBA IDL [9]), communication bindings (e.g., JBI [10] and REST [11]) and nonfunctional properties (e.g. timing and authentication). So, supporting this diversity of technologies needs to define a modular infrastructure for the deployment of heterogeneous application configurations.

The SCA has two important challenges that must be met by SCA platform providers. First, although SCA specification has all the mechanisms required to declare a wide range of variation points in the given application, but it says nothing about the architecture of the platform that implements these variation points. So the first challenge of designing a SCA platform is to be flexible and extensible enough for integrating these variation points.

The second challenge is that the SCA specification focuses on the description of assembling and configuration of components that are used to write a SOA program. The assembly is used as input to initialize and start the program. However, the SCA specification does not talk about run-time management of the program. This management includes monitoring and reconfiguration of the program. In addition,

SCA specification does not include the run-time management of the platform itself. But these properties are necessary for an SOA platform to change the operating conditions, to support online evolution, and to deploy the program in a dynamic environment (such as cloud computing or ubiquitous environment). These problems have been resolved in the proposed platform.

III. HISTORY

Several implementations of the SCA specification are available. These implementations can be divided into two categories: commercial (e.g. HYDRASCA, IBM WebSphere Application Server Feature Pack for SOA, Oracle Event-driven Architecture Suite) and open source (such as TUSCANY, NEWTON, and FABRIC3).

While TUSCANY covers different standards defined by the SCA Open SOA group, the proposed platform focuses on the main features of SCA in Java in order to gain a run-time core with a lighter and faster footprint. In comparison with TUSCANY, NEWTON and FABRIC3, the proposed platform is based on the SCA reflecting functionality programming model to provide a dynamic reconfiguration of SCA applications and the platform itself. Using the proposed platform, SCA components can change their structure at runtime. Also, by using this method, assemblies can be reconfigured to address the new requirements. Finally, by using this method new components can be constructed. These features open a new perspective for agility of SOA, SCA run-time management of applications and the platform itself.

Compared to known component models, such as EJB, COM/.NET [12] and CCM [13], SCA provides a software architecture concept and also the Architecture Description Language (ADL) to give a correct vision on assembling components. The proposed platform expands SCA model with reflection inherited from FRACTAL [14-16] and FAC [17] models.

The proposed platform shares several features such as introspection and reconfigurability with component platforms, such as OPENCOM [18], HADAS [19], PRISM [20], LEGORB [21], K-COMPONENT [22] and JBOSS [23]. However, these models have smaller components in comparison with the SCA components of the proposed platform. These components are comparable with FRACTAL components. The purpose of these models is middleware platforms, such as OpENORB [18]. The purpose of the proposed platform is distributed SOA applications. These programs are essentially heterogeneous in communication protocols and implementation languages. So this platform should be able to integrate several different technologies.

OSGi Declarative Services [2] is another service-oriented component model for SOA. Several platforms, such as Eclipse EQUINOX, Apache FELIX and KNOPFLERFISH have implemented this component model. OSGi Declarative Services is expanded with tools, such as iPOJO [24] to support the overlooked features, like composite components. OSGi and iPOJO focus on Java, while SCA support from multi mapping languages. In addition, OSGi focuses on the life cycle and component identification, while SCA emphasizes on an architecture-centric approach for deploying services. The proposed platform brings the possibility of reconfiguration for SCA, which is beyond those available with OSGi and iPOJO. In addition, since the proposed platform supports the implementation of components with OSGi, a program can be fully implemented with OSGi, while takes advantages of software architecture explained with SCA assembly language. This allows features like OSGi versioning of components to be used.

MADAM [25] and MUSIC [26] are the middleware framework that support dynamic run-time reconfiguration and mobile applications. In particular, the methods take advantages of component paradigm for automatically changing the structure of a program when changing its context. While MADAM defines its own component model, MUSIC uses the OSGi Declarative Services to implement middleware softwares and services. In particular, MUSIC compensates weaknesses of OSGi by defining a component-based architecture for applications and its supporting platform. This configuration of this architecture is continuously optimized by an adaptation middleware.

IV. THE PROPOSED PLATFORM

All components of the proposed platform have been designed and implemented based on the SCA component model paradigm. Figure 3 shows an overview of the proposed platform architecture. Application level is related to end users SCA applications and is designed and implemented by the development team. Other levels are based on the SCA infrastructure and are used to deploy and host applications. In the following paragraphs, we will describe each of these areas. In each level, we will emphasize reconfigurability that has been added to the SCA.

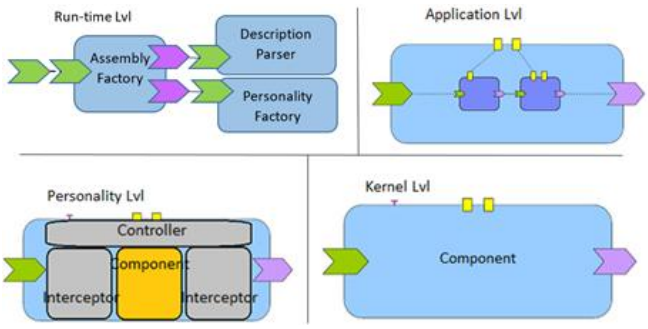

Figure 3. The proposed platform

Kernel level: From a technical perspective, the proposed platform is built on the FRACTAL component model [14]. FRACTAL is a component model independent from the programming languages and is used to create software systems with high configurability. FRACTAL software architecture model combined the ideas from software architecture and distributed configurable systems sources. This model inherits

the main concepts of software architecture [27] to build modular software systems, encapsulated components and explicit communications between them. FRACTAL model is used as a basic model for the development of a variety of configurable middleware platforms. In addition, the model is used for the construction of automated, architecture based and distributed system management capabilities. Some of these capabilities are deployment and reconfiguration management [28,29,30,31], self-repair [32,33], overhead management [34] and self-protection capabilities [35]. FRACTAL model is specified by the formal description [36] based on Alloy language [37].

One of the advantages of FRACTAL is that it enables customizing execution policy associated with a component. FRACTAL implements a particular policy, named component personality (or for short personality) for executing the implemented components. We are using this policy in our platform. Bruneton et al. [14] implemented two personalities named JULIA and DREAM. JULIA is a reference personality for component with a reconfiguration feature. DREAM also is a personality for implementing a message-oriented middleware.

The personalities of each component are implemented by controllers and interceptors. Each controller implements a particular aspect of a personality, such as life cycle management or binding management. The controllers expose their services through their interfaces. Similarly, interceptors change the behavior of components when receiving requests or sending responses.

All FRACTAL components include a control interface named Component. The goal this interface is the same as IUnknown interfaces of COM components [12] that allow the capabilities and needs of a component to be determined dynamically. In other words, the control interface Component defines the identity of a component and plays a role similar Object in object-oriented languages, such as Java or C#.

The API of Component interface is depicted in the relevant section of Figure 3. The API has methods for getting information and type of the interface. The API has separated the service interface from the core component of FRACTAL. On the core, FRACTAL can define components' personalities in modular mode to improve the implementation of policies related to a component and provide several sets of control interfaces.

Personality level: the personality of a component is a structural and run-time feature of the component. Among these features are cases such: how a component should be instantiated, started, wired with peers, activated, reconfigured, how requests should be processed, how properties should be managed, and so on. These features can vary greatly in order to accommodate different operating environments and contexts, such as grid computing, Internet applications, embedded systems and wireless sensor networks.

So the personality design of a component includes the definition of controllers that are needed to implement this meta-level activities. Six of these controllers are included in the personality components of the platform. The API of the six cases is:

- WiringController
  - bindFc(in cltItfName: String, in srvItf: Object): void
  - listFc(): String[]
  - lookupFc(in cltItfName: String) : Object
  - unbindFc(in cltItfName: String): void
- InstanceController
  - getFcInstance(): Object
- Property Controller:
  - getFcValue(in name: String): Object
  - putFcValue(in name: String, in value: Object): void
- HierarchyController
  - addFcSubComponent(in comp : Component): void
  - getFcSubComponents() : Component[]
  - removeFcSubComponent(in comp : Component): void
- LifeCycleController
  - startFc(): void
  - stopFc(): void
- IntentController
  - addFcIntentHandler(in intent: Object): void
  - listFcIntentHandler(): Object[]
  - removeFcIntentHandler(in intent: Object): void

Wiring Controller: The controller allows you to query among the list of wires for each component (lookupFc), creates new wires (bindFc), remove existing wires (unbindFc) and retrieves the list of current wires (listFc). The operations can be done at the run-time.

Instance Controller: SCA specification defines four states for instantiation of a component: STATELESS (all instances of a component are equal), REQUEST (an instance is created from the component for each request), CONVERSATION (an instance is created for each user's session) and COMPOSITE (one instance of the component for each composite). So Instance Controller creates created the instance of each component, based on these four states. Method GetFcInstance, prepared by the controller, returns the component instance associated with the running thread.

Property Controller: The controller can add a property, a key-value pair, to a component (putFcValue) and recover its value (getFcValue).

Hierarchy Controller: The SCA Component model is a hierarchical model. Each component in this model is primitive or composite. The components of a composite are sub-components that, in turn, can be primitive or composite. The management of the hierarchy is done by the Hierarchy Controller. The controller has provided methods to add/query/delete sub-components to a composite.

Lifecycle Controller: when working with multithreaded applications (the general state of distributed applications considered by SCA specification), reconfiguration cannot be done in an uncontrollable from. For example, while a customer request is being processed, any change in wiring may lead to inconsistencies and erroneous results or cause errors to be customers. So lifecycle controller ensures that the reconfiguration is done in safely and consistently manner. Method StopFc turns off a component for performing the reconfiguration. Method StartFc allows the software start processing normal requests.

Intent Controller: The controller is responsible for wiring non-functional service to an SCA component.

Each of these controllers implements a specific aspect of the execution policy of an SCA component. The controllers, in turn, are implemented as FRACTAL components. These controllers need to work together to present an overall execution logic for a component instance. For example, Instance Controller needs to query the Property Controller in order to retrieve the properties values and then inject them in the instances. As another example, Lifecycle Controller needs to create instances of a component at eager initialization, and it must query to Instance Controller. Eager initialization is a particular SCA concept that says SCA component must be instantiated before receiving any client requests. The proposed scheme of cooperation between the controllers is shown in Figure 4. The architecture is used as the backbone of the implementation of component property in the platform. SCA software and platform are inherently distributed and multithreaded. Even if the level of personality is made thread-aware, especially with scope management policy by instance controller, threads are created and managed by the implemented stack protocol. In addition, it is notable that these controllers which have been implemented as FRACTAL components, use a simple built-in personality, similar to implemented personality in JULIA. The rationale of this choice lies in the fact that it is the execution policy of business component, that probably needs adaptability, not one of controller components that implements this policy.

Reconfiguration Capabilities: compared with the SCA assembly language that has only the basic tools for describing the configuration of a program, the platform can get access and modify the configuration at the run-time. The following component elements can be changed at runtime: wires, properties and hierarchies.

For example, based on the program shown in Figure 1, in a reconfiguration scenario, View component can be replaced with a WebView component. Usually, these reconfiguration scenarios include the following steps: (1) stopping the component (and thus getting that component unavailable), (2) unwiring, (3) creating a new component, (4) make new wires with the new component (5) starting the new component. Steps 1 and 5 ensure that reconfiguration is synchronized with the customer's request. Stop a component ensures that no incoming request be processed in the reconfiguration step. Note that the stop element is not mandatory and if the guarantee is not required, then the relevant steps (1 and 5) would be deleted. This process of reconfiguration is provided by the methods of Lifecycle and Wiring controllers. So the definition of a particular reconfiguration policy includes calling the defined controllers methods. Note that the personality level which has been described as a component assembly (Figure 4), can be reconfigurable. As an example, this reconfiguration can provide different versions of execution policies, which determine before modifying wires to the new component, whether the old on should be stopped or not. In fact, we do not define new reconfiguration or a particular style by opening the personality level and reconfiguring it [38].

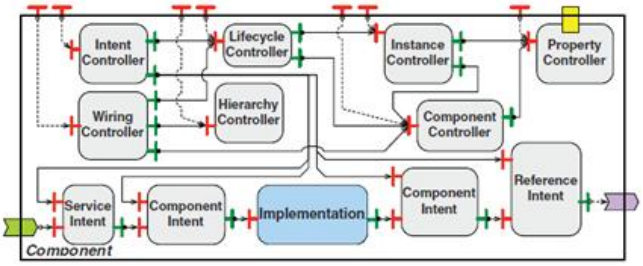

Figure 4. Personality level

By providing a runtime API, the suggested platform enables changing of a SCA program. This feature is of particular importance for designing and implementing agile SCA applications, such as context-aware applications and autonomic applications [39]. For example, Sicard et al. [33] show that in order to support from the complete self-repair feature of a distribution system, the combination of wiring, hierarchy management, property, identity and lifecycle is mandatory at the meta-level of a component model. Same reconfiguration capability is seen in automatic overhead management in the component based cluster systems. [34,40,41]

Run-time level: the duty of run-time level of the platform is to create instances from of SCA components and assemblies. There are three major components at this level that are defined below. As shown in Figure 5, these components are composite and have been implemented with the same personality of business components.

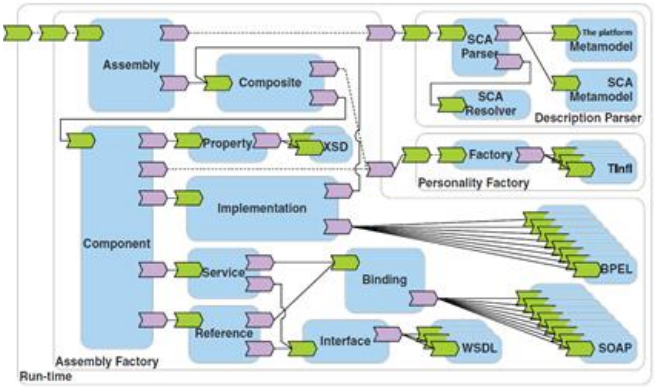

Figure 5. Run-time level

Description Parser: the task of this component is loading and checking the SCA assembly descriptors and constructing the runtime model associated with it. This model is consistent with the meta-model that consists of two parts: SCA Metamodel that categorizes all concepts defined by the SCA specification and Arch Metamodel that is used for describing some extensions that are not in the specification. The meta-model isolation in the platform provides a mechanism to support the main features that are not defined by the SCA specification (e.g. UPnP binding or the FRACTAL implementation type). So the role of SCA parser is to pars the XML-based descriptor to an EMF [42] consistent with the supported meta-model. The EMF then will be finalized by the SCA Resolver.

Personality Factory: The task of this component is creating the personality of SCA components. The nature of the code generated by personality depends on the type of the component implementation (composite, Java, etc.). The platform supports two different production techniques: bytecode and source-code.

Assembly Factory: this component receives the run-time model created by Description Parser and constructs its related component assemblies. Assembly Factory is organized according to the key concepts of SCA model. It is notable that the choice in implementation provides a modular implementation from the implementation process. For example, the Property and Interface components are related to the supported property and interface description languages, while the Implementation component will be wired to a variety of supported implementations. Whenever necessary, Binding component relies on the communication protocol to provide services.

By default, the following plugins are available in the platform. The plugins offer a wide range of features for implementation of distributed and heterogeneous SOA:

- Interface Description Languages (supported by the Interface component): Java, WSDL, UPNP [43] service description,
- Property Description Languages (supported by the Property component): Java, XSD,
- Component Implementation Languages (supported by the Implementation component): Java 8, 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 [44], UPNP, or integrated technologies, OSGi, JNA.

In the platform, the run-time level configuration has not been hard coded into the architecture description. Instead, the runtime level defines a flexible configuration process inspired from the extender and whiteboard [45] design patterns of OSGi. Therefore, the platform defines the platform plugins as architecture fragments that are created dynamically and at the run-time (Figure 6). The platform kernel and its various extensions are defined as partial SCA architectures (the so-called architecture fragments) that are added as plugins to the program or the platform as needed. So the platform configuration process is done in two stages:

1) The bootstrap of the platform runs with minimal configuration. The bootstrap looks for the architecture fragments in ClassPath. Whenever a composite is seen in the loaded fragments, the bootstrap would merge its descriptor content with the main configuration to reaching the final configuration of the platform.

2) When all the fragments of architecture were merged, the bootstrap sets up a new instance of the run-time platform based on the merged descriptor. This version of the platform is then used for creating and managing the business software.

Figure 6 shows an example of the configuration of the platform with OSGi plug-ins and UPnP. OSGi Plugin architecture provides interoperability between SCA and OSGi [2] technologies. OSGi Implementation supports from the implementation of a SCA component as an OSGi package. In addition, OSGi Binding retrieves a reference to an OSGi service from bundel repository and wires it to a SCA program. Assembly Factory supports UPnP Plugin architecture fragments for UpnP service description as well as communication protocols, such as service discovery protocol. Since UPNP is not a member of the standard technologies of SCA specification, the Description Parser must be extended by the proprietary UPnP model.

Reconfiguration Capabilities: The main three terms which are important at the run-time reconfiguration are: binding management, dynamic instantiation and platform extension with new plugins.

In the platform, the binding between components is fully dynamic. This means that the communication protocols are defined in wiring components and named stubs and skeletons. The protocol features, such as message marshaling, are located in these components. Since the stubs and skeletons are SCA components, the URI of a web service (which is available as an option of the component) can be modified at the run-time and reconfigured in the distributed software architecture.

The other main feature of the platform for reconfiguration of SCA system, is dynamic components instantiation. Assembly Factory is invoked at the run-time to create new instances of components. For example, to be able to wire the platform with new extension not seen at bootstrap, the feature enables deployment of the new plugins for Personality Factory.

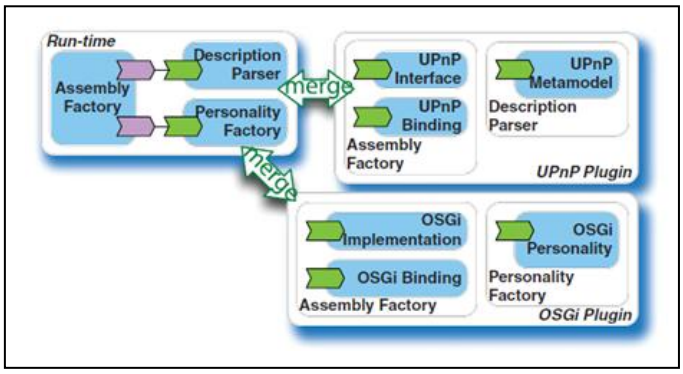

Figure 6. The platform auto-configuration process

V. PLATFORM PERFORMANCE EVALUATION

To evaluate the performance of the platform, we have compared it with Apache TUSCANY Java SCA version. In practice, this version is the de-facto reference of SCA implementation. We have used a simple micro benchmark to compare the memory consumption and execution time of the platform in full configuration with TUSCANY 1.6.

The first measurements evaluate the cost of platform infrastructure. Figure 7 shows the memory usage comparison based on the number of components used.

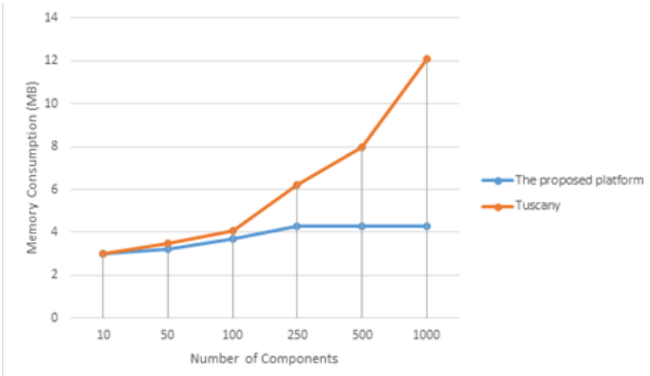

Figure 7. Memory consumption

The second measurement is the cost of a service call. Figure 8 shows the execution time on a local network. In this scenario, the root assembly is called which, in turn, it calls its two children components. The invocation on each component node of the tree is repeated until it reaches the calls to a leaf. The leaves are empty components.

The third measurement focuses on reconfiguration. We measured the time taken by the reconfiguration scenario. In this scenario, a component is replaced by its equivalent in an existing architecture. This scenario involves the following five steps: (1) stopping the component, (2) unwiring, (3) creating a new component, (4) make new wires with the new component (5) starting the new component. The scenario is on assembling two previous scenarios. In this reconfiguration, 1,000-times repetition of a component replacement took 0.35 seconds. Since reconfiguration is a feature available only on the proposed platform, thus comparison with other platforms is impossible. However, this measure shows that reconfiguration cost is low enough to be used in many application areas.

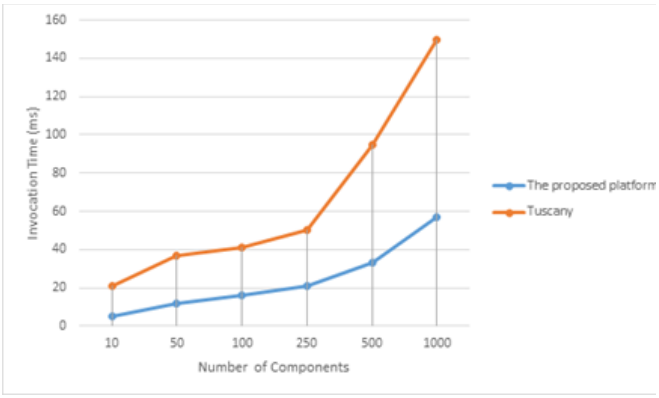

Figure 8. Invocation time

VI. CONCLUSION

This paper presented a platform for extending Service Component Architecture (SCA) [1] based on distributed systems for applications. SCA is a standard for Service-oriented Architectures (SOA). The idea of this platform is to enable consistency and run-time reconfiguration for SCA applications and the platform itself. Furthermore, the system administrator will be able to distribute the components of applications at the run-time, and as s/he prefers. As [3] has said, these issues are a subset of key challenges for researching on SOA. With this platform, the structure of a SCA application can be dynamically modified at the run-time to be able to add new services to the applications, to reconfigure the application according to the new situation and to move running services to new systems. With this platform, you can reconfigure both the system components and the wiring of the system with external services. The flexibility and openness of the platform are also provided in the platform itself.

The proposed platform, like SCA component model applications, uses a component-based structure for its applications. As shown in evaluation of comparison between the platform and Tuscany SCA, the flexibility of the platform is not detrimental to performance.

REFERENCES

[1] B. M., "Service component architecture," 2011. [Online]. Available: http://www.oasis-opencsa.org/sca.
[2] OSGi Alliance. OSGi Service Platform Core Specification Release 4, 2005.
[3] P. M, T. P, D. S and L. F., "Service-oriented computing: State of the art and research challenges," *IEEE Computer*, vol. 40, no. 11, p. 64–71, 2007.
[4] B. M, B. H, B. D, E. M, H. O, I. S, M. A, K. A, M. A, M. J, N. M, N. E, P. S, P. G, R. M, R. M, T. K, V. S, W. P and W. L., "Service component architecture: Building systems using a service oriented architecture," 2011. [Online]. Available: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-sca/SCA White Paper1 09.pdf.
[5] B. S, A. E, B. J and C. D., The J2EE Tutorial (2nd edn)., Addison-Wesley, 2004.
[6] Open SOA. SCA Transaction Policy, 2007.
[7] O. M, A. P, C. V, D. I, D. G, E. B, M. S, M. S, M. N, S. M, S. L, S. E and Z. M., "An overview of the scala programming language (2nd edn)," 2006.
[8] B. S, C. D, F. MF, F. D, R. J and S. J., "XQuery 1.0: An XML Query Language. W3C Recommendation," 2011. [Online]. Available: http://www.w3.org/TR/xquery.
[9] "OMG. Common Object Request Broker Architecture (CORBA/IIOP)," 2008.
[10] "Sun Microsystems. Java Business Integration (JBI) 2.0," 2002.
[11] "Fielding RT. Architectural styles and the design of network-based software architectures," University of California, 2000.
[12] B. D, Essential COM, Addison-Wesley, 1998.
[13] OMG. CORBA Component Model, 1999.
[14] B. E, C. T, L. M, Q. V and S. J-B., "The FRACTAL component model and its support in Java. Software Practice and Experience (SPE)," vol. 36, no. 11-12, p. 1257–1284, 2006.
[15] R. D. Nicola, M. Loreti, R. Pugliese and F. Tiezzi, "A Formal Approach to Autonomic Systems Programming: The SCEL Language," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2014.
[16] "Designing Autonomic Management Systems by Using Reactive Control Techniques," *IEEE Transactions on Software Engineering*, pp. 640 - 657, 2016.
[17] P. N, S. L, D. L and C. T, "A model for developing component-based and aspect-oriented systems," in *Proceedings of the 5th International Symposium on Software Composition (SC'06) (Lecture Notes in Computer Science; vol. 4089)*, 2006.
[18] C. G, B. G, G. P, T. F, J. A, L. K, U. J and S. T, "A generic component model for building systems software," *ACM Transactions on Computer Systems*, vol. 26, no. 1, pp. 1-42, 2008.
[19] B.-S. I, H. O and L. B, "Dynamic adaptation and deployment of distributed components in Hadas," *IEEE Transaction on Software Engineering*, vol. 27, no. 9, 2001.
[20] M. S, M.-R. M and M. N, "A style-aware architectural middleware for resource-constrained; distributed systems," *IEEE Transaction on Software Engineering*, vol. 31, no. 3, 2005.
[21] K. F, M. J, Y. T, C. R and M. M, "Design, implementation, and performance of an automatic configuration service for distributed component systems," *Software Practice and Experience (SPE)*, p. 667–703, 2005.
[22] D. J and C. V, "The K-component architecture meta-model for self-adaptative software," in *Proceedings of Reflection'01*, Berlin, 2001.
[23] F. M and R. F, "The JBoss Extensible Server," *Proceedings of the 4th ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'03),* p. 344–373, 2003.
[24] E. C and H. R, "Dynamically adaptable applications with iPOJO service components," *Proceedings of the 6th International Symposium on Software Composition (SC'07) (Lecture Notes in Computer Science; volume 4829)*, p. 113–128, 2007.
[25] G. K, B. P, E. F, F. J, F. R, G. E, H. S, H. G, K. MU, M. A, P. GA, P. N, R. R and S. E, "A comprehensive solution for application-level adaptation," *Software Practice and Experience (SPE)*, vol. 39, no. 4, p. 385–422, 2009.
[26] R. R, B. P, D. Y, E. F, H. S, L. J, M. A and S. U, "MUSIC: Middleware support for self-adaptation in ubiquitous and service-oriented environments," *Software Engineering for Self-Adaptive Systems (SEfSAS)*, p. 164–182, 2009.
[27] S. M and G. D., Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, 1996.
[28] T, K. J and S. J., "J2EE packaging; deployment and reconfiguration using a general component model," in *Proceedings of the 3rd International Working Conference on Component Deployment (CD'05)*, 2005.
[29] D, d. C. A and D. C., "Peer-to-Peer and Fault-tolerance: Towards deployment-based technical services," *Future Generation Computer Systems*, vol. 23, no. 7, 2007.
[30] P, L. M, G. H, L. T and C. T., "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)*, 2008.
[31] F. A and M. P., "A generic deployment framework for grid computing and distributed applications," in *OTM Confederated International Conferences, Grid computing, High Performance and Distributed Applications (GADA 2006)*, 2006.
[32] B. S, B. F, K. S, H. D, M. A, P. ND, Q. V and S. J., "Architecture-based autonomous repair management: An application toJ2EE clusters," *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05),* 2005.
[33] S. S, B. F and P. ND, "Using components for architecture-based management: The self-repair case," in *Proceedings of 30th International Conference on Software Engineering (ICSE'08)*, Liepzig, Germany, 2008.
[34] B. S, P. ND, H. D and T. C, "Autonomic management of clustered applications," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'06)*, Barcelona, Spain, 2006.
[35] C. B, P. ND, L. R and H. D, "Self-protection for distributed component-based applications," in *Proceedings of the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'06)*, 2006.
[36] M. P and S. J-B, "A formal specification of the fractal component model in alloy," INRIA, 2008.

[37] J. D., "Alloy: A lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 2, p. 256–290, 2002.

[38] P. P, P. F and K. J, "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)*, 2007.

[39] K. J and C. D, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, p. 41–50, 2003.

[40] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and QoS-aware cluster management," in *ASPLOS '14 Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014.

[41] Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems*, Bordeaux, France, 2015.

[42] S. D, B. F, P. M and M. E, EMF Eclipse Modeling Framework (2nd edn) (Eclipse), Addison-Wesley, 2009.

[43] "UPnP Forum. UPnP Device Architecture, Version 1.1," 2011. [Online]. Available: http://www.upnp.org.

[44] G. E, P. C, V. J and D. M, Service Location Protocol, Version 2, 1999.

[45] "OSGi Alliance. Listeners Considered Harmful: The Whiteboard Pattern," 2004.

[46] P. A, L. F, M. P and S. L, "A component framework for Java-based real-time embedded systems," in *The 9th ACM/IFIP/USENIX International Middleware Conference (Middleware'08) (Lecture Notes in Computer Science; vol. 5346)*, 2008.