Universal and Extensible Service-Oriented platform  
Feasibility and experience: The Service Abstract Machine

Jacky Estublier, Eric Simon  
Grenoble University. LIG.  
F-38041 Grenoble, France  

Abstract—Service-based technology is becoming widespread, and many service oriented platforms are available with different characteristics and constraints, making them incompatible. Consequently, designing, developing and executing an application using services running on different platforms is a challenging task if not impossible. A first challenge is to provide an ideal and virtual SOC platform; ideal because it subsumes the actual real SOA platforms, virtual because it delegates the execution to these real platforms. But since “ideal” depends on the domain; the second challenge is to provide extensibility mechanisms allowing to define what ideal means in a given domain and to provide the engineering support for the design and development of service-based application dedicated to such domains. The paper describes how we addressed these challenges, and to what extent they have been met.

Keywords—Service; service platform; extensibility; abstract service machine.

I. INTRODUCTION

Many service based technologies are currently available, and more are expected to come. The fundamental idea behind the service-oriented approach (Service-Oriented Computing or simply SOC) is that modern applications, requiring high levels of interoperability and flexibility, can be developed faster and cheaper as a network of services [1]. Indeed, in the service-oriented approach, the application components (services) have:

- A loose coupling, which improves their reuse potential;
- A platform independent specification of their interactions which improves interoperability;
- A late binding, which improves flexibility.

These properties are provided by all the technologies supporting the SOC paradigm, and can be applied in numerous domains such as business processes[2] or ubiquitous computing[3]. Unfortunately, these technologies have specific characteristics and constraints which make them complex and often incompatible. We are faced with a paradox: the SOC paradigm is meant to simplify the integration and management of applications emphasizing interoperability, but the technology is such that interoperability is problematic, and developing, executing and maintaining a service-oriented application using different service-based technologies is today a challenge if not a nightmare.

We believe that this unfortunate state of practice comes from the fact that today SOC technologies are:

- Incompatible,
- Overly complex, and
- Support only the execution.

Despite their incompatibilities, the SOC technologies implicitly adhere to the SOC paradigm, which make them fairly similar, on the principles. Therefore our goal is to define an “ideal” SOC platform providing the high level functionalities expected from a service based platform, and in which all services are homogeneous. In the ideal, this “new” machine should be virtual only, translating its activity toward real SOC platforms and toward the real involved services, hiding their heterogeneity. Unfortunately, to hide the heterogeneity, it must also emulate the features defined by the virtual machine, but not directly available in the real platform. On one such virtual machine, a service-based application can be easily built and executed.

This Service Abstract Machine (SAM) platform is also the common underlying layer on which we have built a number of tools and Software Engineering environments for the support of the life cycle phases of a software project, not only execution [4].

Section 2 is a short summary of SOC technology, section 3 describes the principles and architecture of our solution, section 4 details how discrepancies between the actual platforms can be overcome; sections 5 and 6 are comparisons and conclusion.

II. SERVICE ORIENTED COMPUTING

In the service-oriented approach, the functionalities are provided by software artefacts named service. The main contribution of the SOC is its interaction pattern; a provider publishes its services, a consumer sends a request to one or more traders to find the services it needs; if successful, the consumer establishes a connexion to the service provider. Frameworks supporting dynamicity also notify consumers of the arrival, departure and sometimes modifications of services. The technologies, which implements this
architectural pattern, is called Service-Oriented Architecture (SOA).

In practice, SOA technology requires additional features and tools. Michael P. Papazoglou defines the Extended SOA concept [5] in which SOC is made of three layers:

- **Foundation**: “basic service capabilities provided by a middleware infrastructure and conventional SOA from more advanced service functionality needed for dynamically composing services”;
- **Composition**: this layer offers composition mechanisms and non functional properties like transactions or the compliance mechanisms of a runtime composite with its definition;
- **Management & Monitoring**: the mechanisms of management and of monitoring for service-oriented applications.

Note that the Non-functional Characteristics or the semantic aspects are crosscutting the three layers.

In their current form, SOC models are simple; for example, they usually do not provide support for containers or concerns, and they do not benefit from specific environments. On the other hand, the SOC paradigm makes that composition is usually implicit and dynamic i.e. performed at runtime; which fits pretty well the needs of a new class of software applications where dynamicity and flexibility are important characteristics. The weakness of the provided support, the demanding characteristics of the new applications, and the heterogeneity and immaturity of the actual technology explains why developing service-based applications is challenging. It requires adapting or rethinking the methods and tools needed for supporting the new service-based applications all along its life cycle, from design to maintenance.

III. SUPPORTING SERVICE-BASED APPLICATIONS

Our solution relies on a set of Software Engineering environments, each one targeting a different life cycle phase. These Domain Specific Software Engineering environments (CADSE) [4] are based on a metamodel that extends a common abstract core SOA metamodel (see Figure 3) with the concepts relevant for the life cycle phase (e.g. development, deployment). We consider the execution framework (the SAM) as one of these environments. It is because all these environments share the core SOA metamodel that they can communicate and share part of their models. It also provides extensibility: “any” other aspect found relevant can be handled by a new CADSE in which the core SOA metamodel is extended with the new concepts. In a similar way, the execution platform (SAM) can be extended in order to handle, at execution time, new aspects, or to provide new features. Therefore the system is not defined once for all but can (and do) evolve.

In particular, we currently provide a development, and a deployment CADSE and their associated run times. These environments are based on metamodels which extend only slightly the core SOA metamodel. Higher level features, like the composite CADSE and its associated SAM composite run time, define more advanced concepts and more demanding features, but are still closely related with the core SOA metamodel. Finally, we have developed business specific CADSE (like for electric application for Schneider Electric [6], or choreography CADSEs [7]) which are based on metamodels extending heavily the core SOA metamodel.

Unfortunately, the SAM is abstract only. Implementing SAM, not only would be a huge work, but it would add to the actual confusion with yet another SOC framework, which is the opposite of our goal. Therefore our approach consists in implementing SAM as a virtual SOA machine, running on top of actual SOA frameworks, and executing the services which are running on these frameworks. To do so, each function provided by the SAM API is a piece of code that calls the underlying platform to perform the SAM action. Different cases can be found:

- The underlying platform provides a similar function. SAM simply delegates or translates the function to the platform.
- The underlying platform provides only partially the function. SAM must emulate the missing part.
- The underlying platform does not provide the function at all. SAM must fully perform the function.

For lack of space, this paper focuses only to this last issue: to which extend it is possible and realistic to hide the heterogeneity between the actual SOA platforms.

A. SAM architecture

The purpose of the Service Abstract Machine is to provide a framework which, from one side, provides high level environments that helps in designing and developing service based applications (SAM CADSEs), and on the other side high level execution platform(s) that hides and subsumes existing heterogeneous SOA frameworks (SAM Machines). The SAM framework uses services which are designed for and executed by existing service oriented frameworks. The SAM machine provides a uniform and consistent API to access, query and invoke these services, irrespective of the underlying platform supporting the service.

SAM framework goal is to provide an ideal SOC environment and associated execution machine supporting the full spectrum of functionalities of the SOC technologies. Unfortunately there is no such definition of “full spectrum of functionalities”, each platform providing a different set of functionalities, and much functionality are still missing. For that reason,
SAM is designed as an extensible framework structured around a core distributed SOA metamodel.

SAM Core SOA metamodel gathers the features found in most actual SOA platforms (see Figure 3), and SAM framework includes an extension mechanism to support the many features not present in core SOA.

SAM framework is made of two parts: engineering environments (CADSEs) and execution environments (Run Time Machines). CADSEs are defined to support a specific life cycle phases of a software project, in a specific business or technological domain. A CADSE is an Eclipse extension (plug-in) specialized in the support of engineers when performing a specific kind of activity. A CADSE is driven by a metamodel that defines the concepts and the operations used by engineers when performing the activity supported by that CADSE. CADSE A extends CADSE B if the A metamodel extends B metamodel. All SAM CADSEs extend at least the core SOA metamodel. The activity performed when using a CADSE produces a model (conforming to the CADSE metamodel), and its associated engineering artefacts; typically programs and their associated metadata.

![Figure 1. SAM framework](image)

A SAM machine is a set of run time (RT) machines; each one specialized in the support of specific SOA related features, including at least the core-SOA runtime. A RT machine is the interpreter of models conforming to its metamodel.

A SAM extension is a couple: CADSE/RT sharing the same metamodel; the CADSE being the editor (it produces models and engineering artefacts), and the RT being the associated interpreter (it interprets the models and executes actions on the artefacts). Figure 1 shows the core SOA and its Development, Deployment and Dynamic composite extensions.

The execution environment is made of three layers: a logical layer containing interpreters, a physical layer containing wrappers and real platforms, and a dispatcher in between.

![Figure 2. SAM Run-Time examples](image)

Each RT is a couple <Interpreter, {wrapper}>. The logical layer of each RT is the model interpreter. The model produced by the corresponding CADSE is reified at execution in terms of objects which are instances of metamodel classes. The interpreter contains the code associated with the operations defined on the class; it allows navigating and querying the model (introspection), changing the model (reflexion) and executing the operations defined on the instances (interpretation). This explains also why the model and its metamodel are the only visible part of a SAM machine.

The physical layer of a RT is made of a number of wrappers. Each wrapper is associated with a real service platform. A SAM wrapper works both in the “top-down” and “bottom-up” ways. In the top-down way a wrapper provides an API which is implemented by delegating, translating or emulating the function toward the real platform. In the bottom-up way, a wrapper observes the platform and notifies the logical layer when something relevant occurred; the logical layer can update its model accordingly. In the Figure 2 example, the Web-Service platform (WS) does not provide the functionalities required for deployment; it means that SAM will not be able to deploy services managed by the Web-Service platform, but yes to deploy OSGi [8] or JEE services. Conversely, the composite logical extension only requires the functionalities defined by the core SOA; all real platforms will therefore support SAM composites.

The dispatcher simply selects or looks for the right wrapper that has to execute the SAM action. For example, if a SAM client asks for a service, the dispatcher will ask the different platforms, in a predefined priority order, to find those providing this service”.

B. The Core SOA

The core SOA metamodel contains the concepts and the functionalities provided by any SOA framework
In this metamodel, a service is an abstraction described by properties (attributes), and additional information (documentation, semantic description etc.). Its real subclasses are Specification, Implementation and Instance, seen as different materializations of the concept of service.

A Specification is the abstract definition of a service; it contains at least an interface and it indicates, through relationships requires, its dependencies toward other specifications. Despite being a rather abstract concept (it does not include any implementation, platform or technical concern), it is possible to define applications and structural composites only in terms of service specifications.

An implementation represents a code snippet which is said to provide a specification. Conversely, a specification may be provided by a number of implementations. The provides relationship has a strong semantics: the implementation object inherits all the properties values, relationships and interfaces of its specifications and it must implement (in the Java sense) the interfaces associated with its specification. In particular, if specification “A” requires specification “B”, all A’s implementations will require B. An implementation can add dependencies, through relationship requires, toward other specifications. It is important to mention that, in contrast to most systems, an implementation cannot express dependencies toward other implementations.

Instances are run-time entities corresponding to the execution of an implementation. An instance inherits all the properties and relationships of its associated implementation; and for that reason it provides and requires specifications.

Brokers give access to each category of service; a machine is a singleton that is related to other remote SAM machines, allowing remote access to services hosted by other SAM machines (see section III.F).

C. CADSE developments

Using the canonical HelloWord example, we may have a client, calling method Hello() found in the HelloSpec service specification, and implemented by the HelloImpl service implementation.

![Figure 3. Core SOA metamodel](image)

![Figure 4. HelloWorld Model](image)

![Figure 5. HelloWorld client Code](image)

In the simplified client code shown in Figure 5, the variable samlocal is the current SAM machine, it is always available. We get the machine specification broker and we ask it to provide the HelloSpec specification, then we ask for all its implementations. Finally, for each implementation, we call the hello() method on all the existing instances. The template is used to format your paper and style the text. All
management and administration parts are SAM heterogeneous service based applications; only the administration and management of legacy applications would like to make use of them, without redeveloping, or even declaring them, as seen by a SAM client. SAM core RT infers, from the information provided by the current designations. The above approach makes the hypothesis that services currently supported by the current platforms. Second, the development RT starts the client: our HelloWorld application starts executing. Each time a client asks for an instance (calling method getInstanceObject), SAM RT realizes the implementation is not loaded in any platform, and calls back the RT that declared that implementation (development RT in our case). Development RT installs the bundle in the underlying platform, and calls back the RT that declared that implementation. SAM core RT builds incrementally a model representing the services really executing in the underlying platforms.

E. SAM core RT

The above approach makes the hypothesis that services, the managed application being fully legacy and unchanged.

Take again our HelloWorld example, where only the client has been developed using usual tools (Figure 5). When the client executes getSpecification(condition), the core RT interpreter asks all its underlying platforms to return the specifications satisfying the condition; it creates and returns the corresponding Specification model elements; and similarly for getImplementations() and getInstances(). When the HelloWorld executes, core RT builds incrementally a model representing the services really executing in the underlying platforms.

SAM core RT is designed to be extended by other RT extensions. In practice a SAM service-based application is a mixture of SAM services (developed under a SAM CADSE), and of legacy services (“discovered” by core RT at run-time). In our example as described in D, the client will find both the implementations and instances declared in the model, and those already present in an platform, and discovered dynamically by core RT, including non-iPOJO ones. The client is not aware where these entity come from on which platform they execute, and when and who installed and started them.

F. Distributed SAM

SAM is a distributed machine, even if the services it manages and executes are local services. For a SAM machine, are local the services managed by one of its platforms. For example, the Web-Service platform reifies, when required, some of the Web-Services available on the network. The WS platform owns a proxy factory for which the specification is the proxy interface, the implementation is the proxy factory and the proxy is the instance; from SAM point of view, a Web Service is a local service.

Each SAM machine uses a mechanism allowing dynamically discovering each other, forming networks of SAM machines. A SAM client can ask its “local SAM machine” the list of known remote SAM machines. From a remote machine (a proxy of that machine dynamically generated), a client can navigate freely through that remote machine brokers, discovering the services available on that machine. From the remote broker, a client can ask creating instances of service implementations, of references to existing instances, exactly in the same way as on the local machine; SAM Dynamically generates the proxies giving remote access to these “local” services. Note that the platform and the service itself ignore that it is accessed remotely.

Therefore, a given service (say a web service) can be both known by different SAM machines (through its web service platform), and each one accessed through the remote SAM machine mechanism: a given service can be accessed through multiple paths. For that reason, a service has an unique identifier: <serviceId, machineId>, with machineId a global unique identifier.
on the network, serviceId an identifier unique in the machine (cf. [10] [11]).

G. Life cycle

For SAM core RT, the services specifications present in the model are “defined”, the implementations are “installed”, and the instances are “active”. This life cycle may be different from the underlying platform life cycle. For example, an implementation (therefore installed in SAM) will be really installed in the OSGi platform only at its first use in SAM. Conversely, SAM core RT creates instances and implementations (and therefore “active” and “install” them) only after they are discovered by a platform wrapper. For example, a Web-Service appears in SAM when its proxy and the corresponding service instance are created.

IV. Supporting real SOA platforms

So far, SAM services appear as if supporting all the features defined in the SAM core metamodel, while the real services are supported by real platforms (and not SAM). Of course, no real platform directly supports our SAM metamodel; therefore this ideal view is not the reality. For example, the concepts of specification and implementation as well as relationships requires and provides are not available in most platforms.

The issue has to be addressed in the two cases: (1) the model is provided by SAM CADSE, as managed by SAM Development RT, and (2) the model is deducted from the underlying SOA platforms, as managed by SAM core RT.

The model provided by a CADSE contains all the meta-information defined in core SOA metamodel, in particular the attributes (properties) and the relationships. SAM clients can therefore navigate and query freely the model during execution. For SAM core RT, the platform wrappers are doing their best to infer (reverse engineer) the SAM model from the information provided by the real platforms; unfortunately most of the meta-information is not available and cannot be inferred (relationship requires for example).

In both cases, the operations defined in the core metamodel may not be supported by the real service or the real platform. For example, instantiating a service makes no sense for UPhP or Web-Services.

The challenge is to provide clients with a SAM model as complete and faithful as possible which indicates what can and what cannot be done on a service, without having to know the real platform and its technical details. We can distinguish different cases:

- The RT logical layer is sufficient to provide the feature. It is the case with the distribution; the proxy generator relies only on the service interface; it does not need to know the real platform. It is a feature always provided.
- The feature is available on the real platform. The wrapper translates and delegates to the platform. It is the case when querying the instance broker, or asking for getting an instance.
- The feature is only partially available on the platform; the wrapper emulates the missing part. For example, the web service wrapper must extract the concept of specification from the WSDL information.
- The feature cannot be provided at all. For example instantiation on a web service or a DPWS platform.

Independently of the platform, services have characteristics that should be known by its clients, in order to use them consistently. For example the fact a service can be shared by multiple clients, or the fact an implementation can have multiples instances.

SAM RT goal being to hide the real platforms heterogeneity and complexity, the solution we propose is to characterize the services such that a SAM client can use consistently a service irrespective of the real platform and its technical details.

The core SOA metamodel predefines properties indicating the features the associated entity does not support. The characteristics associated with the general concept of service are the following:

- Sharable: The same service can be used by different clients.
- Stateless: The implementation or instance does not have state.
- Remotable: can be accessed from another SAM machine.
- Dynamic: The dynamic properties are a set of three orthogonal characteristics: DynamicAppear, DynamicDisappear and DynamicSubstitute if respectively the associated entity can dynamically appear, disappear or be substituted by another one.

Some characteristics are associated to specifications and implementations indicating how are to be managed their associated implementations or instance. They are:

- Multiple: More than one implementation of a specification or many instances of an implementation can be present in a given SAM machine.
- Instanciable: The specification or the implementation is associated with a factory allowing creating dynamically the associated implementation or instance.

By default these properties are supposed to be true. If a service sets one of these properties to false, it means that the service cannot fulfill the associated feature. Conversely, an administrator or a composite service can turn some properties to false, meaning that the associated feature will not be allowed for the associated
entity respectively in the current SAM machine or in the current composite service.

The development CADSE automates or assists users in setting these properties; some platform wrappers are such that they set systematically some characteristic to false; as for instance for the web service platform (dynamic, remotable, importable, …). Unfortunately, some of these properties cannot be inferred (statefull, sharable …); it is the platform wrapper responsibility to take an optimistic approach (true by default), or pessimistic one (false by default).

Note that in most SOA technologies, the dynamic characteristics are false because the late binding is not seen as a dynamic feature for SAM. Conversely, OSGi is dynamic; but the substitute property assumes a stateless service which is not necessarily the case.

Other characteristics can be defined and managed by the extensions; for example, the deployment extension can define a migrable property. By lack of space, the details about the dynamic properties, the extensions properties and the characteristic scope are not discussed in this paper.

V. RELATED WORK

SAM can be compared with other service platforms from different perspectives: distribution, interoperability, extensibility and integration with engineering environments.

Web service SOC platforms, like AXIS2, as well as Jini and Corba are natively distributed and most often hide the communication protocol used. But major platforms, like OSGi, are centralised. A major feature of SAM core is to turn all services, irrespective of the platform, into distributed services.

R-OSGi [12] also transforms OSGi into a distributed platform. R-OSGi discovers remote services and provides a communication protocol based on proxy generation. R-OSGi hides the distribution by reifying in each OSGi registry all the OSGi services available in the network. In contrast, SAM is not limited to OSGi and does not hide the fact a service is remote at least for parameter passing semantics, efficiency, availability, reliability and scalability reasons. SAM reifies the remote machines allowing navigating the network, and proxies are generated only on the first access to a remote service. SAM does not hide the distribution but the communication technology.

The iPOJO technology [9][13] leverages OSGi, by an extension mechanism (by code injection) and a high level metamodel close to SAM’s core. It is no surprise since both works are developed in the same team. iPOJO focuses on dynamic aspects and addresses extensibility through “handlers”. Indeed SAM core RT is implemented in iPOJO using specific handlers.

Service Component Architecture (SCA) [15] addresses both interoperability and distribution issues and proposes a metamodel similar to core SOA (designed before SCA specification were available). For example, specifications are close to component-types, and instances to components. Unfortunately, SCA does not provide an extensible SOC metamodel or interoperability with engineering environments. In SCA, interoperability between heterogeneous services requires using remote protocols, even on the same machine.

SAM approach to extensibility is based on model and metamodel composition both for the engineering environment and in the run-time platform, providing platform extensibility and generic interoperability with life cycle phase support. No platform we know offer comparable features.

VI. CONCLUSION

We presented in this paper the SAM framework and one of its components: SAM core SOA. SAM core tries to reach utopias which are (1) to provide a unique consistent and ideal SOC platform, (2) still using legacy and heterogeneous services running on existing platforms.

SAM core SOA relies on a uniform and consistent core SOA metamodel containing the best of the actual SOC platforms (point 1). To reach points 2, for each legacy platform we defined “wrappers” in charge of delegating, translating or emulating the core SOA functionalities on top of the real platform. Point 2 is a utopia since it is not always possible to emulate the missing features. For that reason, we introduced characteristics expressing for each service what features are not supported.

Clearly point 1, an “ideal SOC platform” does not make sense; it depends on the needs, the domains, the stakeholders and the technology. Further, we believe that currently, the main difficulty is not running but designing and developing service-based applications. It requires supporting different life cycle phases in which high level specific concepts are used and it requires the run time platform that recognizes and interprets these concepts at execution. For all these reasons we designed the SAM framework.

SAM framework challenges are to be (a) highly extensible (b) to consistently support both the associated engineering activities and the execution; and (c) to be capable of supporting both model driven execution, and the traditional SOC non-deterministic execution philosophy. Point a) has been addressed using model and metamodel extensions developed in other context ([4]); point b) by the architecture presented in Figure 1. Experimentation is under way to assess the limits of this approach, but so far, it gave us satisfactory results.

As presented, SAM is a model-based engineering and execution framework for service-based applications, since applications execute following models developed in the associated CADSEs; it is a
deterministic and closed view. This view contradicts the very fundamentals of the SOC paradigm which emphasizes instead a non-deterministic and open world in which services are “naturally” available. Challenge c) consists in managing and controlling the conflict arising when both views are used simultaneously.

We believe that SAM framework allows each company using SAM to build and use its own “ideal SOC platform”.

We believe that SAM contributes to SOA paradigm at least because it proposes a high level and consistent core SOA platform, hiding the many current platforms, and because it provides mechanisms for extending and adapting the SOA paradigm to specific domains, both from an engineering and execution platform points of view.

REFERENCES


