Opportunistic Computing
Experience with the SAM platform

Jacky Estublier
Grenoble University.
220 rue de la Chimie
38041 Grenoble France
+33 476 63 55 64
Jacky.Estublier@imag.fr

Idrissa A. Dieng
Grenoble University.
220 rue de la Chimie
38041 Grenoble France
+33 476 63 55 63
Idrissa.Dieng@imag.fr

Eric Simon
Grenoble University.
220 rue de la Chimie
38041 Grenoble France
+33 476 63 55 87
Eric.Simon@imag.fr

Diana Moreno-Garcia
Grenoble University.
220 rue de la Chimie
38041 Grenoble France
+33 476 63 55 87
Diana-Guadalupe.Moreno-Garcia@imag.fr

ABSTRACT
Service-Oriented Computing (SOC) technology allows one to build applications exhibiting opportunistic, non deterministic and dynamic behavior. Unfortunately in SOC technologies these features are uncontrolled and software application is not an explicit concept. The challenge therefore is to provide a well defined application definition (an application model) in which the opportunistic behavior is used only when convenient, and to provide an extended SOC platform which enforces compliant application execution. This paper discusses the issues and shows the solution proposed by the SAM platform.

Categories and Subject Descriptors
D.2.6 Programming Environments, D.2.11 Software Architectures, D.2.13 Reusable Software, H.1 MODELS AND PRINCIPLES, J.6 COMPUTER-AIDED ENGINEERING.

General Terms
Experimentation, Design, Languages.

Keywords
Service Oriented Computing, Service Selection, Service Composition, Composite services, Software engineering environments.

1. INTRODUCTION
The fundamental idea behind the service-oriented approach (Service-Oriented Computing or simply SOC) [10] is to propose an interaction protocol in which the client asks a registry which services, defined by their specifications, are available, and it selects and uses the most appropriate one of those returned by the registry. This interaction protocol gives to SOC its two fundamental properties:

- **Late binding**, since done at run time and only if needed,
- **Loose coupling**, since the client does not need to know its provider, but only the required service specification.

Late binding is nothing new, java classes or DLL are also bound at run time and only when needed, but classes and DLL are implementations, while in SOC services specifications are required, not implementations which explains why it provides loose coupling property. More important in our opinion, SOC interaction protocol also provides the opportunistic property. It is opportunistic, in the sense that the client can use one of the available service implementations; not necessarily the one planned before hand.

Not all SOC platforms provide all these features, but more important, not all clients are willing to make use of them. For example, most web service [1] clients do not use any registry and directly binds to a web service statically known by its URL, and no opportunism is provided.

In this paper we concentrate on service platforms like OSGi [8] where the registry is effectively used, where dynamism is natively supported, and where a service provider can be a client of other services. In this context, since different providers of the same service may have different service dependencies, different selections may lead to radically different executions.

SOC is based fundamentally on the concepts of service specification and service instance (and their packaging like packages and bundles in OSGi), but nowhere a concept like software application is defined. Because of SOC properties, the application “magically” emerges, if lucky, as the side effect of the interactions between unknown services randomly selected. In the general case, this is not desirable; at least some degree of control and determinism is required. This strategy...
(or better say absence of strategy) works if the execution machine(s) runs only a single and know implementations
of the services required by the application to execute. But in this case only the late binding property is preserved.

Another strategy, like in SCA [9] is to statically load and wire all the needed service implementations before to start the
application. But in this case none of the service properties are preserved.

Another possibility consists in qualifying the requested services with enough details (e.g. LDAP filter expressions in
OSGi) to be sure that only the “good” provider(s) can satisfy the constraint (for example, asking the provider by its name).
In this case loose coupling is lost if the client has to statically know its providers; and only late binding is preserved.

Process-based approaches (orchestration, choreography) [3] are deterministic in what concerns the ordering of service
invocation, but they can preserve some flexibility when it comes to select the concrete services to call. It is an interesting
compromise, but orchestration hypothesizes (or ignore) that services do not depend on other services, which is not true in
general.

Currently, we do not know systems where service-based applications can be defined making use of opportunism. This
is a great lack since service properties are mandatory for the new kinds of software applications where dynamism,
flexibility and adaptation are important characteristics.

We are therefore faced with two apparently conflicting requirements: on the one hand having an explicit and reliable
application definition, and on the other hand, taking advantage of the opportunistic property of services. Reconciling these visions requires

- An application definition leaving room for opportunism i.e. leaving well defined places where much flexibility is
  left at run time, and

- A run time platform that drives the application execution such that its definition is fully satisfied.

The paper shows the principles of the proposed approach. In section 2, we present the sate model and the
SAM machine; section 3 presents an informal global approach and an example to illustrate it. Section 4
defines the application model, section 5 presents the opportunism modelling and execution; section 6 details
the implementation of our prototype and some experimentations, and finally section 7 presents our conclusion and future work.

2. STATE MODEL. THE SAM MACHINE

Different service platforms are currently available, they all propose roughly the same concepts (service specification,
instance, registry; interaction protocol etc.) but they all differ in their API, conventions and semantics, making them
incompatible. Technically, developing an application making use of different platforms (e.g. UPnP, OSGi or Web service)
is a real challenge. The primary challenge addressed by SAM (Service Abstract Machine) [7] is to propose a unique
(abstract) platform that subsumes “all” the currently available service platforms. To do so, SAM proposes a unique SOA
metamodel with its API (see Figure 1).

Service is the central concept of this metamodel. A service
owns interfaces and properties. However a service has several aspects: specification, implementation and instance, which are
different materializations of the Service concept. Consequently, Specification, Implementation and Instance are
subclasses of the Service abstract class.

![](image)

**Figure 1. SAM metamodel (simplified)**

A Specification is an abstract definition of a service. It is
abstract because it does not contain technical concerns that are
specific to a particular service platform. The interface
(inherited from Service) defines the provided functionalities,
and the attributes define the visible properties that all its
implementations and instances must support. It indicates,
through relationships requires, its dependencies toward other
specifications. Having properties and provides and requires
relationships, it is possible to define service-based applications
only in terms of service specifications; therefore irrespective
of any implementation, platform or technical concern.

An implementation represents a code snippet which is said to
provide one specification. This code snippet can be an instance
factory or a simple class, in all cases it must be the logic code
of the service. Conversely, a specification may be provided by
a number of implementations.

Instances are run time entities corresponding to the execution
of implementations. In most systems, implicitly, the word
service refers to service instance.

We call state model, the current state of the system represented
as a graph of Specifications, Implementations and Instances
currently running in the real platforms. Using the SAM API,
users can create, instantiate and/or connect services (i.e. build
a state model) as in a concrete platform. But SAM is an
abstract machine; a service in SAM is the reification of a
service present in a concrete platform. To do so, to each
service platform supported by SAM is associated a wrapper
(SCM for Service Concrete Machine in Figure 2) which
observes the concrete platform and updates the SAM model
accordingly. Conversely, when the SAM model is changed
(using the SAM API), the “right” wrapper is called in order to
update the real machine accordingly (if possible).

For example, the OSGi wrapper listens the events sent by the
OSGi platform (Felix for example). When a service starts,
Felix sends an event caught by the OSGi wrapper. This
wrapper asks Felix in order to get as much information as
possible of the new service, and builds in SAM a specification
object with the right interface and properties, and an instance
object with the “right” properties. The OSGi wrapper cannot
infer those concepts like dependencies between specifications
because they do not exist in OSGi platform. Therefore, the state model provided by SAM is sometimes complete, but it is the most complete, faithful and consistent representation of the reality. For that reason, in the following, we will consider that the SAM model is the real state model.

Figure 2. The SAM architecture

3. APPLICATION MODEL

An application model, when convenient, should be able to define explicitly the required component implementations and their connections, as traditionally done with component-based technologies. But in this case the service properties like late binding and opportunism cannot be used. These properties of services come from the fact the interaction protocol is performed in terms of specifications (the client asks for a specification, not for an implementation). Following this approach, an application should be defined in terms of specifications only. Unfortunately, in most service platforms, specifications do not have properties or dependencies. To solve the issue, we defined the concept of group which is an extension of powertypes and materialization [2].

3.1 Groups

Let us take the example of three SpellChecker implementations: MySC, FrenchSC, TextSC each having attributes and relationships. They are all implementations of the SpellChecker service because they all provide the same SpellChecker interface; they also share some properties: they all need a dictionary and they all have a “language” attribute, but they differ in some aspects (see Figure 3).

An equivalence group (group for short) is made of an object (called the head) and a number of group members. A solved member (member for short) is the union of a group member and its head. Said differently, the head holds the common properties of the group, and the group members hold only the specific properties. Therefore, our example can be represented by the SpellChecker group. The SpellChecker group has as head the object SpellChecker (the head is named after the group) which holds the common values and the members that hold only what is specific. SpellChecker is a real object as such it has a type. In our example, SpellChecker is a Specification as presented in Figure 1 and the members are Implementations. The relationship between Specification and Implementation is both member and provides. A major contribution of the group concept is that the group head not only holds the common properties but also defines the properties by which the members can be distinguished (and selected). In our example, the object SpellChecker defines the attribute “language:String” which means that each SpellChecker member must define a value for that attribute: the group head is both an object and the type of its members.

In this example, Dictionary is also a group, and the dependency between SpellChecker and Dictionary being defined between two types is a relation type (of cardinality 1) which means that each SpellChecker member must have a dependency toward a Dictionary member. In our SOC system, we have defined two group types <Specification, Implementation> and <Implementation, Instance> (see Figure 1). In the example, MySC is both a member of the SpellChecker group and the head of group MySC with MySC_01 and MySC_02 members. As a group head, MySC defines the attribute “addrs:String” which explains why its members have values for that attribute.

3.3 General Architecture

In our approach, the software engineering activities are performed using CADSEs [6]. SAM development CADSE produces the application model (cf section 4) and a component database. The execution is supported by SAM run time which maintains the state model. A SAM manager is in charge of controlling or enforcing conformance between the state model and the application model, observing the application and state models and the information in the component database. This is a typical case of runtime models as mentioned in [12] and more precisely introduced in [13].
3.4 An example scenario

Suppose we are developing the MyTextEditor member of the TextEditor group. To test it we write the following application model:

```
Config MyEditorTest {
  Init TextEditor/MyTextEditor;
  Optional printing{type="laser"}{speed>=8ppm};
  Select SpellChecker{language="English"};
  Delay SpellChecker ;
}
```

The first line expresses that MyEditorTest application starts by executing (an instance of) MyTextEditor. This is an example of the traditional static and deterministic way of expressing architectures. The second line expresses that a laser printer with a speed greater than 8ppm should preferably be used (any other one if not available). Third line says that we need an English SpellChecker.

Suppose that the SAM run time manager is asked to execute the MyEditorTest application.

First SAM manager tries to resolve the initial configuration: TextEditor/MyTextEditor. To do so, it looks for an instance of MyTextEditor in the state model. If there are none, the manager looks at an implementation of MyTextEditor is available in the current machine; if not the manager looks into the SAM database. In our case the manager deploys MyTextEditor from the database, creates an instance and starts it.

That instance soon later calls the registry asking for a SpellChecker. The SAM manager intercepts this call and finds that the configuration requires the language to be English. Since no SpellCheckers are currently running, the manager searches the SAM database, and finds that two implementations satisfying the constraints are available. It selects one of them arbitrarily (MySC).

When MySC calls the registry for a dictionary, the same occurs. The application model says nothing about dictionary, but we suppose that the SpellChecker specification includes the following constraint:

```
Select Dictionary{language = self.language};
```

Self stands for the current SpellChecker resolution: the current MySC instance which has the language = English because it is so in its group head. SAM manager therefore will look for an English dictionary; it finds one currently running (Oxford), and one in the database (Collins). Being in opportunistic mode, it selects the one which is running (Oxford).

This toy example shows how the service properties can be used still having an explicit application model. Now we will define more precisely the mechanisms and concepts involved.

4. APPLICATION DEFINITION AND MODEL

On one extreme an application can be defined by the list of all the implementations it requires (see the first line of our example).

On the other extreme, an application description can be totally empty. The application manager uses the database and starts from the specification the application provides (its group head) and resolves it i.e. it selects one or more available members. If the selected implementations require other specifications, the process is repeated to resolve these specifications. The process stops when all the needed groups are resolved, therefore with the full list of implementations.

Both extreme are unsatisfactory; the first one because it is too static, the other one because it may lead to any inconsistent composition.

4.1 The constraint language

We have defined a selection and constraints language that may look like OCL, since it allows navigating the links and performing logical expressions on the attributes and links found in the database. Unlike OCL, our expressions are associated with objects, either types of instances, and act on types as well as on instances, since group heads are types and they are declared dynamically. But the main difference lies in the purpose of the language. Like OCL, constraints may be used to enforce internal database consistency but the main purpose of the language is to record and enforce the compatibility constraints that often exist between objects. For example the FrenchSC implementation may indicate

```
Select Dictionary{language = "French"};
```

meaning that any application containing FrenchSC must also contain a French dictionary. Constraints can be more general, like the one presented above:

```
Select Dictionary{language = self.language};
```

which factorizes these constraints at group level.

Note that since group heads are types, the constraints interpreter knows that language is an attribute declared for the Dictionary group. Constraints are strongly typed despite the fact that types can be created dynamically.

An implementation that executes only on Linux should not be associated with an implementation that does not run on Linux. It can be expressed as follows:

```
Select Implementation {! ((Self.attributeDef.name = "OS") (Self.OS != "Linux"))};
```

Which means that, it attribute OS is defined in a group, then must be selected only those implementations that run on Linux.

An automatic composition may fail because it has not been possible to resolve a group, either because no member satisfies the constraints (the application will be incomplete) or because some constraints are in conflict (the application will be inconsistent). For that reason, the automatic selection manager can be executed in “backtrack” mode, which means that if a resolution fails, the manager undoes previous selections and tries another composition. This ensures that if a solution exists, it will be found, but it may be very expensive or impossible in practice for large databases.

4.3 Universal versus contextual constraints

The above constraints are called universal constraints because they must be satisfied by any application making use of the associated object. It guarantees that any application will be made of components for which no incompatibility is known.
An application description can add other constraints, which purpose is to indicate the intention, properties and features the application must show. The language used is the same, but the constraints and properties are interpreted only in the context of the current application.

The application defined above (MyEditorTest 3.4) simply indicates that an English spell checker is required of course much more complex definitions are possible.

4.4 Delayed selection

The application manager (tries to) resolve all groups found in the transitive closure of the dependency relationship, ending if successful in a full list of components. This does not fit our requirement of late binding and dynamic execution. For that purpose an application definition can indicate which resolutions should be left to a later phase in the application life cycle. This is indicated by the expressions like Delay SpellChecker; which means that group SpellChecker must not be resolved now. It is the user, and ultimately the execution machine, that turns off the delay property.

The SAM manager therefore contains the exact same application manager as used during development; it tries to resolve at run time the groups not yet resolved with the same algorithm (the very same code runs).

4.5 Application model

The application manager, after completing its selection process, generates an application model. This application model is a graph whose nodes are the selected objects and arcs the dependency relationships, as found in the database. The root node is the application object. Nodes are (clone of) the databases objects, and therefore include their attributes and constraints. An application model is the minimum subset of the SAM database needed for execution.

5. OPPORTUNISM MODELING AND EXECUTION

An application model is a model conforming to the SAM metamodel (figure 1), and extended with the concept of application (contextual constraints, composition relationships etc.). It can be seen as a sub-set of the database from which are removed the unused groups, in which resolved groups only contain the selected member(s), and in which unresolved groups are empty (for space and performance reasons).

The state model is also a (degenerated) model conforming to the SAM metamodel. It is degenerated because, in the general case, the underlying service platforms do not support all the SAM concepts. Most often only interfaces and instances are available in a state model, and most attributes and relationships are not present.

However, even if in theory the objects found in the state and application models are the same, their meaning is different. The fact an object pertains to an application model means that this object (or its resolution, or a similar object) may be used during execution. An instance in the state model means that this instance is currently running. Nevertheless, an imageOf relationship can be established between an object in the application model and the “same” object in the state model.

By language abuse we will say that an instance in the application model pertains to the state model (and vice-versa) if there is a relationship imageOf between these two instances (seen as the “same” instance).

Given a group O, we call resolution of O the fact to select, initialise or generate a member of O. In SAM metamodel, O can be a Specification (which resolution provides an implementation), or an Implementation (which resolution provides an Instance).

We call complete resolution of an object O the transitive closure of the resolution operation. The complete resolution of a Specification is one (or more) instance(s) that implements that specification.

We call resolution set of an object O the set of all its possible resolutions; and complete resolution set the transitive closure of the resolution set operation. The complete resolution set of a Specification is the (possibly infinite) set of instances that implement that specification.

We say that a state model instance pertains to an application model object O if it pertains to O’s complete resolution set.

Conversely, an object O in the application model is active if there exists in the state model an instance pertaining to O.

Activating an application model object O consists in establishing an imageOf relationship between an instance Oi in the state model and one instance Oi of O’s complete resolution. The issue is to identify or create the “best” instance Oi in the state model that will pertain to O. The activation of an object O can be:

- **Strict.** From O’s complete resolution set, is selected an instance Oi that is already existing in the state model.
- **Simple.** Is returned an instance Oi pertaining to the state model that can be a resolution of O. Oi may not pertain to O’s complete resolution set.
- **Extended.** A strict activation is tried: if it fails (no Oi instance is available), Oi must be created in the state model, which may require deploying its implementation from the database and creating an instance Oi in SAM.
- **Opportunistic.** It is an activation where are tried successively the strict, simple then extended modes.
- **Strongly Opportunistic** It is an opportunistic activation of O’ where O’ is O group head. A strongly opportunistic activation of an implementation is the opportunistic activation of its specification.

Usual service platforms only provide the simple resolution mode, and usual component platforms only provide the strict resolution mode. In our example above (section 3.4) in strict mode resolving Dictionary produces an error; in simple mode it returns Collins, in extended mode it returns Oxford, in opportunistic or strongly opportunistic modes it returns Collins.

Now suppose that the application model explicitly includes Oxford as Dictionary resolution. Activating Oxford in strict, simple, extended or opportunistic modes produces an error; in strongly opportunistic modes it returns Collins.

Executing an application A means first to activate its initial object(s). It is A’s initial state s0, s0 conforms to A by
construction. A’s state changes if an instance pertaining to that state asks for another service, if a service pertaining to A disappear or if a property of a service pertaining to A changes.

A state \( S_{i+1} \) is conforming to an application A if \( S_i \), conforms to A and \( S_{i+1} \) pertains to the complete resolution set of A. An execution of A is said to be conforming if the succession of states starting from \( S_0 \) are all conforming.

SAM manager is in charge to enforce a conforming execution of application models. It first creates the initial state, and each time A’s state is about to change, SAM manager takes the control. If an A’s state instance xi asks for a service Y, SAM manager looks in A’s model if it is true that xi depends on a Y resolution. If true Y is activated (see above) and yi (the instance Y was resolved to) is returned to xi. We got a new state still conforming to A’s model.

6. EXPERIMENTATION

The SAM machine currently supports OSGi [8], iPOJO [4], DPWS, UPnP and web services. It has been used in various European projects, and in various research projects. It is an open source project that can be found at http://sam.imag.fr.

CADSE is also an open source project available at http://cadse.imag.fr. SAM-Dev is one of the most popular CADSE, but iPOJO, orchestration (FOCAS) [11], CADSEg [6], and a few dozen CADSEs are in current use. In the scope of this work, we have developed SAM-Composite [5], that extends SAM-core with the concept of application, constraints (universal and contextual) and selection language; SAM-Deploy that extends with the capability to deploy and execute a service from a repository; and SAM-Dynamic, not presented here, that supports fully dynamic applications.

A CADSE manages a workspace which always includes software artifacts (source code, files and so on) and a model that represents the same in a high level metamodel. All SAM CADSEs extend, in different ways, the same SAM-core metamodel (see Figure 1). For that reason, in SAM Runtime, we are using the same code for the management of the model, and the same resolution engine is executed. It ensures complete consistency between the resolution performed during development and resolution performed at run time. It is also the same code that runs when it comes to select something in the database.

7. CONCLUSION

This paper presents a work in progress. The work already performed demonstrated the validity of the fundamental ideas: an application can be defined in a very flexible way using a mixture of a classic deterministic architecture (explicit components and connectors defined before execution) and an abstract description in intention. Such an application model allows designers to take advantage of the opportunistic property of services but only when convenient.

The second fundamental idea is that it is possible to execute such a loose application model, and to define what an execution conforming to that model is. Indeed, an application model describes in intention a very large and potential infinite number of different but conforming executions. It is also shown how the SAM manager can enforce a conforming execution while taking as much advantage as possible of the current state of its underlying heterogeneous service frameworks.

The third fundamental idea addressed here is that execution is nothing else than the last resolution step in the process from an application design to its execution. Each step resolves some elements and leave others intentionally undefined, either because some component (service) are still to be created, or because the choice must be left to later phases, including execution where properties like opportunism can be used.

This paper did not discuss a number of associated issues like the evolution model, conformance vs. validity, dynamic behaviour and extensions; they are the topic of forthcoming papers. Many issues are still to be addressed. In this work, intentionally, a single application model is executed; in real life many applications are running, the legacy ones do not have any model, each one of the others having its own application model, but potentially they all share various services and resources.

We believe that this work contributes in different areas; models at run time, dynamic architecture, opportunistic behavior, and the seemingless transition between different phases of the software engineering.

8. REFERENCES


