Flexible Composites and Automatic Component Selection for Service-Based Applications

Jacky Estublier, Idrissa A. Dieng, Eric Simon and German Vega

Grenoble University - LIG, 220 rue de la Chimie, 38041 Grenoble BP53 Cedex 9, France

Abstract. In traditional Software Engineering approaches, an application is described as a composite entity containing all its components. This approach is no longer relevant in modern Software Engineering, at least when developing service-based applications where some components (services) are selected very late during the development process or even “discovered” at execution. This new context requires describing an application in a more flexible way, leaving room for delayed selection. In turn, if component selection can be performed all along the life-cycle, an application description must explicitly include the application requirements and goals and the system must at least ensure that the selections satisfy the application description.

In this work, we propose a concept of composite addressing the needs of the advanced and flexible service-based applications, automating component selection and building composites satisfying the application description and enforcing minimality, completeness and consistency properties. We also propose tools and environment supporting these concepts and mechanisms in the different phases of the application life-cycle.

1 Introduction

Service-based technology is becoming widespread, and an increasing number of applications applying this technology are under development. Despite this success, developing service-based applications, today, is a challenging task, because the technology is somehow different, and because these applications often have unusual characteristics and constraints, which require adapting or rethinking the methods and tools needed for supporting their development.

Service Oriented Computing (SOC) [13] like its predecessor, Component Based Software Engineering (CBSE), relies on a clear separation between interface (service) and implementation (component). But SOC emphases the fact many implementations and even many running instances of a service may be available (locally, on the net or elsewhere) and that the selection of the implementation(s) and/or instances to use for a given service can be performed at any time during the life-cycle, including at execution. These characteristics make that SOC is especially well adapted to new kind of software applications like those managing captors, sensors and actuators. For example, in a house, a window shutter service may be present an undefined number of times (it is the number of windows having an automated shutter); many implementations of this service may exist and may be used simultaneously (each

Estublier J., Dieng I., Simon E. and Vega G. (2009).
Flexible Composites and Automatic Component Selection for Service-Based Applications.
In *Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering - Evaluation of Novel Approaches to Software Engineering*, pages 9-20
DOI: 10.5220/0001857500090020

shutter maker may have a different implementation of the service); new services and implementations can appear or disappear during the application execution (installing or removing a shutter, of same or different brand); and the same shutter can be used by different software applications. This context does not fit the usual component based technology, which implicitly, hypothesizes a static structure, with a single component implementation and instance per service, known before hand, not shared and so on. It does not mean that it is not possible to develop, with traditional technology, applications with more relaxed hypothesis, like the one mentioned above, but in this case the designer and developer are left alone with complex and low-level technology, without any tools and methods to help them; and development turns out to be more a hacking nightmare than Software Engineering. To a lesser degree this also applies to service-based applications due to the current lack of support tool.

Our main objective is to facilitate the realization of advanced service-based software applications which requires concepts, tools and Software Engineering environments which natively support the hypothesis required by such advanced applications. A software application is often described as a composite entity; but depending on the life-cycle phases, the composite elements and their relationships are of different nature. For example, at design time, the application can be described in term of coarse grain functional elements with constraints and characteristics; while at deployment time, it can be a set of bundles with dependencies. The "usual" composite concept fits mostly the development phase with elements being components and relationships being wires. This concept of composite is unsatisfactory for at least two reasons:

- It is too rigid to accommodate for the flexibility required by advanced applications.
- It is a low-level implementation view of the application.

Different kinds of composites have been proposed so far, adapted to different needs, different technologies and different contexts. Orchestration and choreography [14], as well as ADL (Architecture Description Language) and configurations are different kinds of composite. For example, orchestration has been proposed to solve some issues found in service-based applications, with the hypothesis that services (most often web services) can be discovered at execution, that services do not have dependencies, and that the structure of the application is statically defined (the workflow model). In this work, we propose a concept of composite that addresses the needs of the advanced service-based applications, proposing technical concepts and mechanisms allowing designing, developing and executing applications which require high levels of flexibility and dynamism.

This paper is structured as follows: In Section 2, we briefly introduce our SAM / CADSE approach. Then in Section 3, we propose a way to define and manage the concept of composite addressing several needs of service-based applications. Section 4 highlights related work. Finally, we present our conclusion and future work in Section 5.

2 The SAM / CADSE Approach

2.1 The Approach

In “pure” Service Oriented Architecture (SOA) like web services [1], there are not explicit dependencies and the orchestration model is a static architectural description. Dynamic service frameworks like OSGi [11] do the opposite: only service dependencies are known, and there is no explicit architectural description. This kind of mechanism provides dynamic behavior since the framework is in charge to resolve dynamically the dependencies; but it has the drawback that the application is not explicitly defined, it has no architecture; no explicit structure. There is a conflict between making explicit the application structure and content; and providing the application a large degree of dynamism.

It is interesting to mention that this conflict also exists between the early design phase when only the purpose, constraints and gross structure are defined, and the implementation phase that (usually) requires knowing the exact structure and content of the application.

To solve this conflict, we propose to see a software project as a succession of phases which purpose is to gradually select, adapt and develop components until the structure and content of an application is fully defined and complete. These phases are either performed by humans, before execution, or by machines at execution. Or course, the machine at execution can only perform selections; but for those selections to be automated (at all phases), at least a part of the goal, purpose and constraints of the application must be made explicit, and a repository of components must be available.

Our solution relies on two systems: SAM (Service Abstract Machine) a service framework for executing the application, and a set of CADSEs (Domain Specific Software Engineering environments) [5] in which are performed the Software Engineering activities. The approach makes the hypothesis that each activity receives a composite as input, and produces a composite (the same or another one) as output. The output composite represents the same application as provided as input, but more precisely. In the same way, SAM receives a composite in input; it executes those parts that are defined, and complete those that are not fully defined, dynamically selecting the missing services.

Therefore our approach relies on a composite concept which can describe the application in abstract terms, through the properties and constraints it must satisfy, and which can describe that same application, in terms of services and connections, as understood by the underlying service platform(s). We believe that there is a continuum from these two extremes; each point being represented by a composite. All these composites and environments share the same basic SAM core metamodel, presented bellow.

2.2 SAM Core

The goal of Service Abstract Machine (SAM) is to dynamically delegate the execution performed in SAM toward real service platforms, like OSGi, J2EE or Axis

Web services. Its basic metamodel, called SAM core therefore subsumes the metamodels supported by the current service platforms.

A composite, during execution, is expressed in terms of the concepts exposed by SAM core; but composites also represent the application during the early phases of the life-cycle; SAM core is the metamodel shared by all composites, both in the Software Engineering activities and at execution; for that reason it must be abstract enough and independent from specific platforms and technologies.

The central concept is *service*. But *service*, in SAM core is an abstraction containing a Java interface along with its properties (a set of attribute/values pairs) and constraints (a set of predicates). Its real subclasses are *Specification*, *Implementation* and *Instance*, seen as different materializations of the concept of *service*. *Specifications* are services indicating, through relationships *requires*, their 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 a structural composite only in terms of service specifications, as well as semantic composites in term of constraints expressing the characteristics (functional or non functional) required from the services that will be used. Still, the system is capable to check completeness (no specification is missing), and consistency (all constraints are valid) on abstract composites, making it relevant for the early phases.

An *implementation* represents a code snippet which is said to *provide* one or more *specifications*. 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 specifications. 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 (threads) corresponding to the execution of an implementation. An instance inherits all the properties and relationships of its associated implementation. Fig. 1 illustrates the concepts of our SOA model:

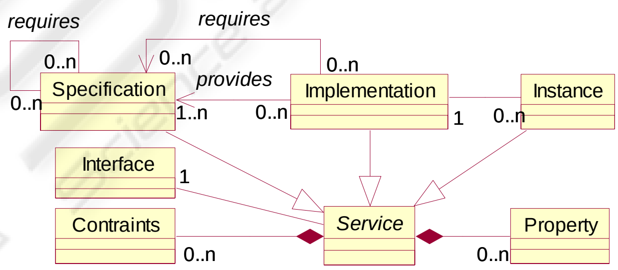

Fig. 1. SAM Core Metamodel.

2.3 An Example

Let us introduce our Media Player Application (MPA) example that we use thereafter in this paper. In our system, each MPA to be built is a composite which consists of a media renderer (which consumes a flux, for example a video stream), a media server (which provides flux found in a storage), and a controller that interacts with the customer and connects servers and renderers. Each MPA must fulfill a set of characteristics (properties and constraints) to be consistent. Its components (services) should be chosen among those available in the SAM repository if available, if not they are to be developed, but in any case these services must be compatible and deliver the desired characteristics of the composite.

Suppose the SAM repository has the content shown in Fig. 2. In this repository, *MediaPlayerImpl* is an implementation which integrates the functionalities of both a media renderer and a controller; therefore it *provides MediaPlayer* specification. The *Log* specification has two implementations namely *LogImpl_1* and *LogImpl_2*. *LogImpl_1 requires* a *DB* (a database) and *Security* (for secure logging). *DivX* is an implementation that provides *Codec* specification, and that has a property "*Quality = loss*". *Codec* has property *Unique=false* which means that an application may use more than one *Codec* implementation simultaneously. *Unique* is a predefined attribute whose semantic is known by the system, "*Unique=true*" is the default value. *Shared=true|false* (e.g. property of *LogImpl_2*) is another predefined attribute expressing the fact that an implementation or an instance can be shared by different applications (*false* is the default value).

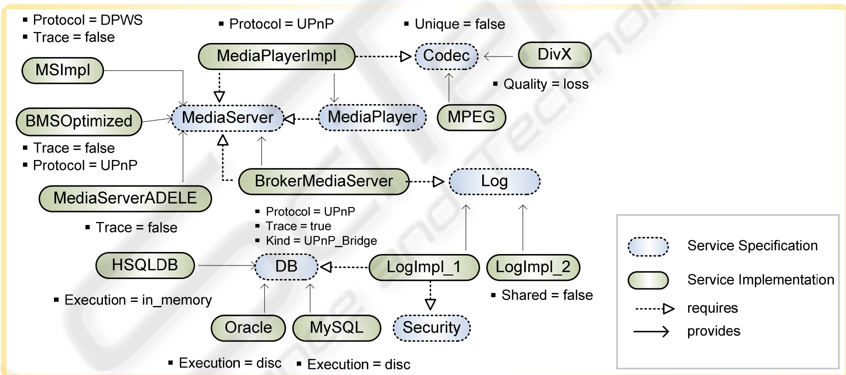

Fig. 2. A SAM Repository.

The *MediaPlayer* specification *requires MediaServer* specification which means that all its implementations (e.g. *MediaPlayerImpl*) also require *MediaServer* specification. From a service (e.g. *BrokerMediaServer*) we can navigate its relationships to obtain for example the service(s) it requires (*Log*) or its available implementations (*LogImpl_1* and *LogImpl_2*) following the *provides* relationship.

Services may have constraints which, like in OCL, are predicates. In contrast with OCL, these constraints can be associated both on types and on service objects (i.e. Specifications, Implementations and Instances). The language allows both navigating over relationships and LDAP search filters as defined in [6]. For example, *LogImpl_1* implementation may declare that it *requires* an *in_memory DB.* This constraint can be expressed as follows:

```
Self.requires(name=DB)..provides (execution = in_memory);
```

Self denotes the entity on which the constraint is associated (*LogImpl_1)*. *Self.requires* denotes the set of entities required by Self (*{DB, Security}*); *(name = DB)* select the elements of the set satisfying the expression (*DB*), *..provides* is a reverse navigation which returns all elements that provide *DB* (*{Oracle, MySQL, HSQLDB}*); finally the expression returns the set {HSQLDB} since it is the only *DB* implementation with property (*execution= in_memory*). An expression returning at least an element is considered true. The constraint means that from the point of view of the object origin of the constraint (*LogImpl_1*), *DB* has a single valid implementation: *HSQLDB.*

If the constraint follows a single relationship type, it expresses which relationships are valid. When the relationship is created, the constraint is evaluated for that relationship, if false the relationship cannot be created. For example, if we want that *Codec* implementations cannot have more dependencies than *Codec* itself, we can set the constraint:

```
equals(Self.requires, Self..provides.requires);
```

In this example, *Self* denotes the *Codec* specification that defines the constraint. *Self..provides* denotes the set of implementations which provide it (*MPEG* and *DivX*). This constraint is relevant for **all** implementations providing *Codec*. Therefore, such constraints enforce some repository integrity.

The *BrokerMediaServer* implementation may declare that it requires *MediaServer* implementations that are *UPnP*, but not a bridge:

```
Self.requires(name=MediaServer)..provides
             (&(kind!=UPnP_Bridge)(protocol=UPnP));
```

3 Composite

Suppose that we want to build an UPnP-based home appliance MPA such that, when running in a particular house, it is capable of discovering the media servers available in that house and to provide *MediaPlayer* functionalities. Defining that MPA in the traditional way, as a composite which gives the full list of components, is inconvenient, or even impossible for a number of reasons:

- Some components may not exist (yet);
- Some components may be known (yet);
- There is no guaranty of completeness;
- There is no guaranty of consistency;
- There is no guaranty of optimality.

Creating a complex composite on real cases, is not only a time consuming and error prone task; but it may be simply impossible when components are missing (they must be developed like the *Security* service in Fig.2) or when components are selected in a later phase, or even discovered during execution (like *MediaServers* and *Codecs*).

Our goal is to propose a way to build and manage composites that avoid the above pitfalls. Such composites are rather demanding; indeed they require the following properties:

- ***Completeness control***. A composite must be capable of being explicitly incomplete; this is the case when components will be developed, when design choices have not made or when the components cannot be known before execution time. The composite must tell what is missing and why.
- ***Consistency control***. The composite must be able to detect and report inconsistencies and constraints violation.
- ***Automation and optimality***. The system should be capable to compute an optimal and consistent list of components, satisfying the composite requirements and constraints, but also the degree of completeness required.
- ***Evolution***. Incomplete composite must be such that they can be incrementally completed.

The following sections present the concepts and mechanisms for composite management.

3.1 Static Composite Definition

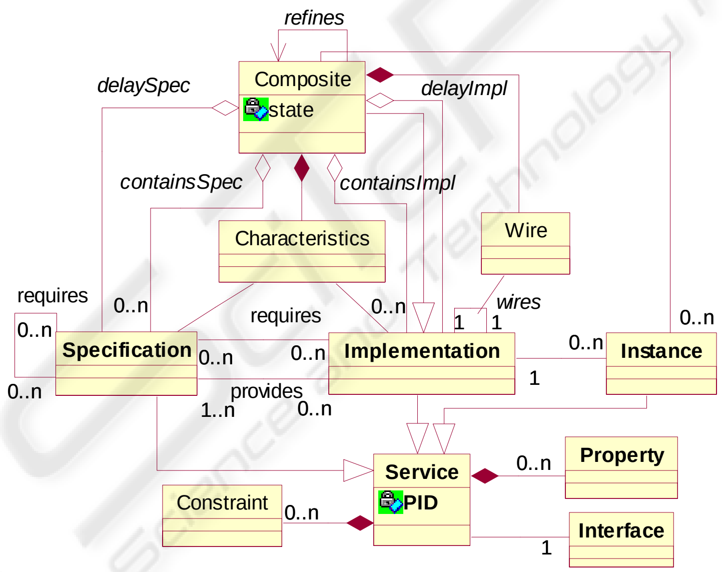

Fig. 3. Composite Metamodel.

In our system, we define a composite as an extension of the SAM core presented above. The extension presented in Fig. 3 is only one of the different SAM Core extensions; indeed other simpler composite concepts have also been defined and implemented.

As shown in Fig. 1 a SAM composite is a service implementation that can contains specifications (*containsSpec* relationship), implementations (atomics or composites;

containsImpl relationship) and instances. Classically, a SAM composite can be defined by the list of its service components (specifications and/or implementations) setting explicitly the *containsSpec* and *containsImpl* relationships.

Being an implementation, a composite is not necessarily self-contained; it may have *requires* relationships toward other services. Similarly, it is not necessarily complete. As explained above, a composite must be capable of being incomplete by giving explicitly the delayed choices of components. Thus, a *delaySpec* or *delayImpl* relationships toward an entity "X" express the fact that the selection process should not follow the "X" dependencies. Delayed service selections can be carried out at any later time for example during development, at deployment or at execution; the strategy is up to the user.

3.2 Automatic Composite Building

Selecting manually the components of a composite, and creating explicitly the associated relationships is tedious and error prone since this manual process does not guaranty *minimality* (all components are useful), *completeness* (all required components are present) nor *consistency* (constraints are all valid). To simplify the process of defining a composite and to enforce the properties of *completeness*, *consistency* and *minimality*, we need an automatic composite construction mechanism. Thus, we need a language in which it is possible to specify the required characteristics of the composite to build, and an interpreter, which analyses the composite description and selects, in the database, the components that together constitute a composite satisfying the description, complete and consistent.

Therefore, a SAM composite can also be defined by its goal i.e. by its characteristics, properties and constraints. We use a language to describe the intended properties and constraints of composites. To create a composite, the designer first defines the Specification(s) it provides, and then, optionally, imposes some choices explicitly creating relationships indicating the Specifications it *requires*, the Specifications or Implementations it *contains* and those it *delays*. Then the designer expresses the expected composite properties; our system performs the rest of the job. For example, to build our UPnP-based home appliance MPA, one could first create a *provides* relationship to *MediaPlayer*, a *delaySpec* relationship to *Codec*, and a *containsImpl* relationship to *MPEG* as in Fig. 4:

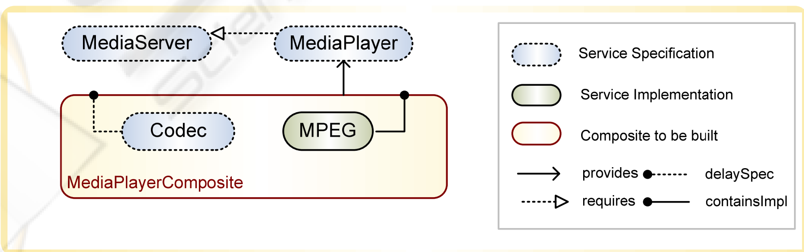

Fig. 4. Composite initial MPA definition.

Then we can declare the intended characteristics of the MPA as follows:

- We want to create an UPnP MPA:
  `Select Implementation (Protocol=UPnP);`
- The MPA to build must provides a trace of executing actions:
  `Optional Implementation (Trace=true);`
- The MPA should foster service sharing whenever possible.
  `Select Implementation (Shared=true);`

This language is an extension of the constraint language, in which *Self* can be replaced by any set, including a complete type extension, like *Implementation* meaning all actual implementations found during the selection process. In our example, traces are preferred but not required. Since the system is weakly typed, an expression is ignored if no element (in the selection set) defines the attribute; if only one element defines the attribute with the good value, it is selected. The first sentence means that we must select an implementation with *protocol=UPnP* for those Specifications for which at least one Implementation defines the *protocol* attribute; in our example, this selection applies only to *MediaServer*.

These expressions are interpreted when computing which are the required services and selecting the implementations and instances that fulfil (1) the intended composite constraints, and (2) all the constraints expressed by the already selected components. For instance, if we select the *LogImpl_1* implementation then we will necessarily select *HSQLDB* since it is the only DB that satisfies the *LogImpl_1* constraints.

Based on the description of a composite i.e. its initial relationships and its constraints, an interpreter computes and selects the required services that satisfy the composite characteristics. The interpreter "simply" starts from the specification provided by the composite (e.g. *MediaPlayer* in our MPA), and follows the *requires* relationships to obtain all required specifications. It also follows the *requires* relationships of its contained services (*containsSpec* and *containsImpl* relationships). For each *specification* found (except those delayed and those explicitly required by the composite itself), it tries to select one or more implementations satisfying all the constraints associated with the services already selected and the selection expressions defined by the composite itself. For each selected implementation the interpreter iterates the above steps to found other required services. For instance, from the repository in Fig. 2 and the selection expressions declared by the MPA, the interpreter builds the following composite Fig. 5:

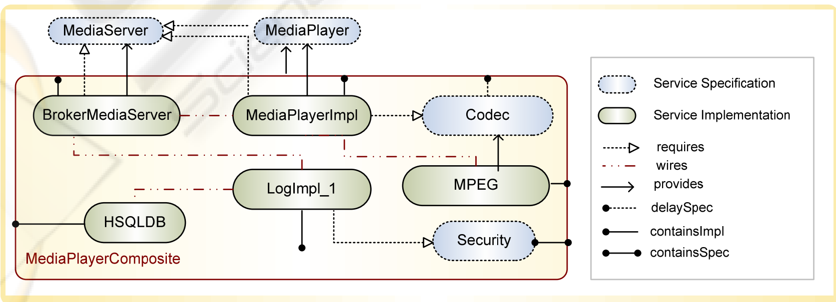

Fig. 5. A configuration of the MPA composite.

For the *MediaPlayer* specification, the interpreter selected its unique implementation *MediaPlayerImpl* but *MediaPlayerImpl* requires a *MediaServer,* therefore *BrokerMediaServer* is selected since it is the only *MediaServer* implementation satisfying the composite selection *(Protocol=UPnP)*. In turn *BrokerMediaServer* being selected, a *Log* service is required, and *LogImpl_1* is selected, because *LogImpl_2* does not satisfy the composite constraint *(Shared=true)*; and consequently *HSQLDB* (because of the *LogImpl_1* constraint) and *security* because of the *requires* relationship. Unfortunately S*ecurity* has no available (or no convenient) implementation; the state of the composite is "incomplete" and this specification is added in the MPA composite through the *containsSpec* relationship. *Codec* being delayed, the system does not try to select any of its implementations; at run-time, depending on the discovered *MediaServers*, the required *Codecs* will be installed. Since the *Codec* implementations cannot have other dependencies (because of the *Codec* constraint), there is no risk, at execution, to depend on an unexpected service. Composite dynamic execution behaviour will not be discussed in this paper.

Our system guarantees minimality (nothing is useless), completeness (except for explicit delays) and consistency since all constraints and selection expressions are satisfied. But we no not guaranty optimality because the system may fail to find the "best" solution or even a solution when one does exist. Indeed, during the selection process, if a specification with *(unique= true)* has more than one satisfactory implementations, the system selects one of them arbitrarily. It may turn out to be a bad choice if the selected one sets a constraint that will later conflict with another component constraint. The solution consists in backtracking and trying all the possibilities, which turns out to be too expensive in real cases.

3.3 Composite Contextual Characteristics

SAM composite extends SAM Core defining new concepts (e.g. composite), new relationships (e.g. contains, delay); but any individual composite can also extend the existing services with properties that are only relevant for the composite at hand. These relationships and properties are called contextual characteristics (see Fig. 3).

For example, *wire* is a contextual relationship; it means that two implementations are directly linked but for a given composite point of view only; this is not true in general. We may wish to add property to some implementations such that constant values, parameters or configuration information which are those required in a given composite only, like bufferSize, localPath and so on. Implementations, along with their contextual properties are called components in SCA [12]. Contextual characteristics may apply to implementations, in order to create instances with the right initial values, as in SCA, but also to specifications, when specific implementation can be generated out of some parameters. More generally, contextual characteristics, including constraints, apply to any delayed service, when the selection must be performed in the scope of the current composite, and with the properties only relevant in that scope. This is fundamental when selections are delayed until execution.

4 Related Work

We can classify the approaches and languages for service composition as orchestration, structural and semantic [8], [3], [10].

Orchestration [14] is a recent trend fuelled by web services which de facto standard is Business Process Execution Language for Web Services (BPEL4WS) [9]. Structural composition defines the application in term of service component linked by dependency relationships. Service Component Architecture (SCA) specification [12] is a structural SOA composition model [14]. Most efforts to automate service composition are performed in the web semantic community. The hypothesis here is that services do not have dependencies, and that specifications include a semantic description using ontology languages such as OWL (Ontology Web Language) or WSML (Web Service Modeling Language). The goal is to find an orchestration that satisfies the composite semantic description (OWL-S [16] or WSMO [17]).

Automating service selection has been addressed in many research works focusing on quality-of-service (QoS) criteria like reliability or response time. [7] propose a QoS based for service selection and discuss about the optimisation of this selection using heuristic approaches. [2] propose an approach for dynamic service composite and it reduce the dynamic composition to a constraints (ontology based) satisfaction problem. In most of these systems, QoS requirements are specified at the overall application level. Therefore, it becomes unclear how to derive the QoS goals from participating services [19]. In our approach, service properties and requirements can be specified both on the individual services that will participate in a given composition and on the composites themselves.

5 Conclusions

SOC represents the logical evolution we are witnessing in Software Engineering: increasing the decoupling between specifications (interfaces) and implementations, increasing the flexibility in the selection of implementations fitting specifications, delaying the selection even until execution time, allowing multiples implementations and instances of the same service to pertain to the same application, and finally allowing some services to be shared between different applications during execution. Building a complex (service-based) application in this context is very challenging.

The traditional way to define an application is, on the one hand, through a number of documents and models in which the purpose, constraint and architecture of the application are defined, and on the other hand, through the full list of its components (being services or not), often called a composite (or configuration). Building the composite manually is tedious, error prone when components developed independently have conflicting requirements, and even impossible when some selections are to be done very late in the life-cycle.

We propose to extend the concept of composite in order to represent faithfully the application along the different life-cycle phases, from design to execution. To that end, the composite must contain a high level description of the application, in term of properties and constraints it must satisfy, and on the other hand in term of components, bundles and run-time properties.

In this work, we show how it is possible to go seemingly from the high level description to the execution one, and how the system, all along this long process, is able to compute and enforce the conformity and compatibility of the different descriptions, while enforcing minimality, completeness and consistency properties. Our work is a step toward the above goal, but even in its current form, it provides a large fraction of the properties discussed above and show the feasibility of the approach. We expect future work to present an implementation of the full picture.

SAM is available at http://sam.ligforge.imag.fr and CADSE at http://cadse.imag.fr

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, H.: Web Services – Concepts, Architectures and Applications. Springer, Verlag (2003).
2. Channa, N., Li, S., Shaikh, A. W., Fu, X.: Constraint Satisfaction in Dynamic Web Service Composition. In 6th International Workshop on Database and Expert Systems Applications (2005) 658– 664.
3. Dustdar, S., Schreiner, W.: A survey on web services composition. In International Journal of Web and Grid Services (IJWGS) 1 (2005), 1-30.
4. Escoffier, C., Hall, R. S., Lalanda, P.: iPOJO: an Extensible Service-Oriented Component Framework. In SCC'08 International Conference on Services Computing, IEEE Computer Society (2007) 474-481.
5. Estublier, J., Vega, G., Lalanda, P., Leveque, T.: Domain Specific Engineering Environments. In APSEC'08 Asian Pacific Software Engineering Conference (2008).
6. Howes, T.: RFC 1960: a String Representation of LDAP Search Filters. Web site: http://www.ietf.org/rfc/rfc1960.txt (1996)
7. Jaeger, M. C., Mühl,G.: QoS-based Selection of Services: The implementation of a Genetic Algorithm. In KiVS Workshop: Service-Oriented Architectures and Service Oriented Computing (2007) 359-370.
8. Milanovic, N., Malek, M.: Current solutions for web service composition. Internet Computing, IEEE 8 (2004), 51–59.
9. OASIS (2007). Web Service Business Process Execution Language Version 2.0. Web site: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf
10. Orriens, B., Yang, J., Papazoglou, M. P.: Model Driven Service Composition. In ICSOC'03 International Conference on Service Oriented Computing, Springer 2910 (2003), 75-90.
11. OSGi Release 4. Web site: http://www.osgi.org/Specifications/HomePage
12. OSOA (2007):.Service Component Architecture: Assembly Model Specification Version 1.0.: http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications
13. Papazoglou, M. P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges, IEEE 40 (2007), 38–45.
14. Papazoglou, M. P., Van den Heuvel, W. J.: Service oriented architectures: approaches, technologies and research issues. VLDB Journal 16 (2007), 389–415.
15. Peltz, C.: Web services orchestration: a review of emerging technologies, tools, and standards. Hewlett-Packard Company (2003).
16. W3C (2004). Semantic Markup for Web Services. http://www.w3.org/Submission/OWL-S/
17. WSML: Web Service Modeling Language. Web site: http://www.wsmo.org/wsml/
18. W3C (2005). Web Service Modeling Language.: http://www.w3.org/Submission/WSMO/
19. Yen, I-L., Ma, H., Bastani, F. B., Mei, H.: QoS-Reconfigurable Web Services and Composition for High-Assurance Systems. IEEE Computer Society Press 41 (2008), 48-55.