An Extensible Services Orchestration Framework through Concern Composition

Gabriel Pedraza, Jacky Estublier

LIG, 220 rue de la Chimie, BP53
38041 Grenoble Cedex 9, France
{Gabriel.Pedraza-Ferreira, Jacky}@imag.fr

Abstract. Service composition is one of the major ways to get new applications out of existing software components (services). The focus so far was mainly on composition formalisms, while most of the real hard issues are related to the many concerns that must be combined, and the limited help provided by the current tools. In this paper we present an approach and a platform in which a service based application is defined through different models along different concerns. The FOCAS platform includes some basic concerns (control, service, and data) and provides support for composition with any other concerns. The platform provides support for the definition of non-functional concerns in the form of annotations over the orchestration model.
The paper shows the concepts and the technology allowing to define an application as a composition of concerns, functional or not, and shows the experience with the concerns currently supported by the FOCAS platform.

1. Introduction

It is a common belief that the technology behind services has the potential to increase reuse and dynamism [16]. It can increase reuse because many services exist and can be directly used, even when operated by third parties on foreign computers (web services). Dynamism can be significantly improved since service technology relies on dynamic discovery and connection between client and service providers. Moreover some service platforms like OSGi [14] provide transparent and dynamic disconnection and reconnection to new services providers.

These properties of service based applications logically, have fostered many works, most of them focusing on the formalism in which the composition has to be described, called orchestration. The current industrial services orchestration standard is WS-BPEL [6], but many other formalisms have been proposed [2][12][18]. However, it is now clear that the formalism itself is not the main issue, and even WS-BPEL can be sufficient in a number of cases. The difficulty comes from the many concerns that must be addressed when developing industrial strength applications.

A first source of difficulty is to make interoperate services designed independently, and therefore potentially heterogeneous in technology (languages, platforms), data (representation and semantics), or in interaction protocols. Web services [1] [16], with

XML/WSDL/SOAP standards solved a number of these issues, but still many incompatibilities require mediation (data, semantics and protocol).

A second source of difficulty is related with the interoperability with other company tools. The target application may address business domains, in which large software applications and databases are already available. The service based application must be able to address these business or technological domains, and interoperate with legacy applications, which are usually not services.

A third source of difficulty is related with the so-called non functional concerns. Any real industrial service based application must support a number of these non-functional concerns like efficiency, distribution, scalability, security, transactional behavior, dynamism, etc. Of course, some work and standards addressed these issues. Web Services standards such as WS-Security [13], WS-SecurityPolicy [7], WS-AT, address some of these concerns at service level, but adding them to service compositions must be undertaken manually, which is a hard and error-prone task.

Our goal is to propose a method, a development environment and a runtime platform allowing: (1) the company method and technology team, to define which concerns, functional or not are to be addressed, and to develop the environment that support these concerns; (2) the software engineer, which is not necessarily a technology expert, to develop easily an application, clearly expressing, through models and annotations, functional as well as non functional characteristics of the target application.

The paper is structured as follows. Section 2 shows that a service orchestration is the composition of three models. Section 3 unveils how these three models are composed, and shows how composition method is generalized to any domain. Section 4 presents how abstract domains can be composed in order to add non-functional characteristics to applications. Section 5 validates our proposition. Section 6 compares it with the state of the art. Section 7 concludes the paper.

2. Services Orchestration

The first step, when building a software application, is to decompose it in parts, in order to reduce the complexity. But no decomposition can cope simultaneously with the different aspects of an application. Each concern (or aspect) is a part of software that is relevant to a particular concept, goal, or purpose [15].

The main idea behind service orchestration is that services are existing, autonomous and independent (which is often not the case in reality), and the decomposition criterion is an ordering (orchestration) of services execution.

2.1. Orchestration Core Domains

Even if it is not formally expressed in the actual formalisms, orchestration is based on three main domains, control, services, and data. Control expresses the execution

ordering, service defines the computing to be performed, and data defines the information on which the execution is to be performed. In most propositions, these three domains are closely intertwined. In contrast, we suggest defining these three aspects independently, through models. We believe this separation is a major progress since it allows expressing, in the control model, what is the business logic, irrespective of details of the underlying services and independent from the data definition, which makes sense since many services are existing, and each company has already information systems in place, used by a number of software applications.

In our approach, a basic orchestration domain is made of the composition of the control, service and data atomic domains. Hence, formal and explicit definition of metamodels in each domain proved to be of great help, both conceptually, to understand clearly what the domain is about, and technically as the main support for tools which have to process models.

2.1.1. Control Domain

In our approach, the Abstract Process Engine Language APEL [8] is used to express the control model. APEL is a high level process language containing a minimal set of concepts that are sufficient to understand the purpose of a process model.

In APEL an activity is a step in the process that results in an action being performed. The actual action to be executed is not defined into the process model, and it can be a service invocation (a Web service, DPWS service, an OSGi service), any kind of program execution (legacy, COTS) or even an human action. Ports are the activities communication interface, each port specifies a list of expected products.

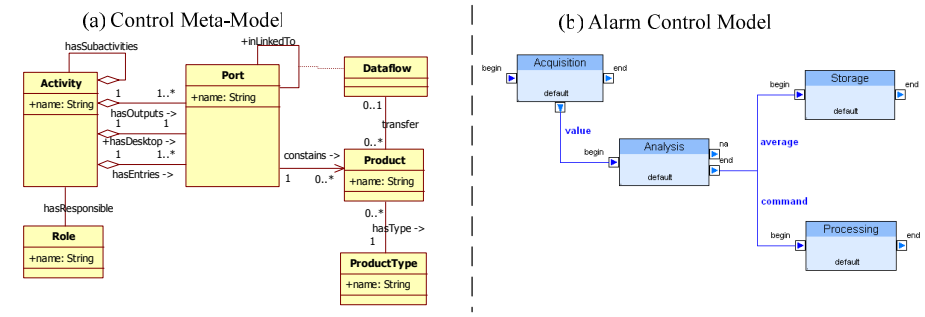

Fig. 1 Control Domain Components

A Product is an object (records, data, files, documents, etc) that flows between activities. Products are represented by variables having only a symbolic name and a type which is also a symbolic name (e.g. *"client"* is a product of type *"Customer")*. This property does not preclude of the actual nature, structure and content of the real data that will circulate in process. Dataflows connect output ports to input ports, specifying which product variables are being transferred between activities.

Graphically an activity is represented as a rectangle with tiny squares on sides which denotes its ports, a dataflow by one line that connects ports and products as labels on dataflows.

Left side of Fig. 1 presents the APEL metamodel, and right side presents an APEL model created with our editor. The model example is an alarm system, where the *Acquisition* activity is in charge of collecting environment measures, then the *Analysis* activity performs a computation over the gathered data, finally two parallel activities are executed, the first-one *Storage* activity makes the data persistent and *Processing* activity triggers an action in case of abnormal data values.

2.1.2. Data Domain

The data domain manages the information used in the company; it specifies the data types (structure and content) and manages the access to the information. This data abstraction makes it possible to hide the heterogeneity of the data used by both, the concrete services and other company applications. Data are represented in two concrete syntaxes: a graphical representation (UML class style) and as Java classes to facilitate programmatically data handling.

Left side of Fig. 2 presents the data metamodel, and the right side presents a possible data model for the alarm system, in which the gathered measure is a room temperature. Data types declared in this model are Temperature, Average and Action.

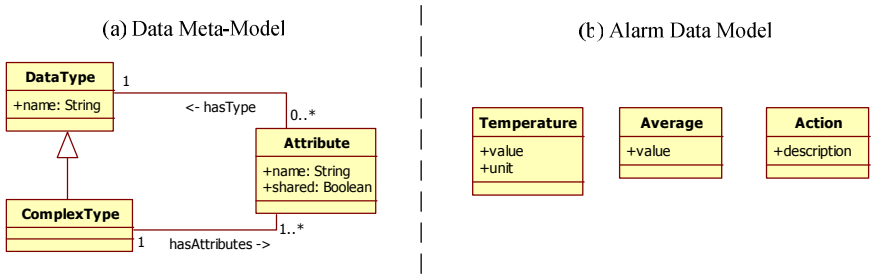

Fig. 2 Data Domain Components

2.1.3. Services Domain

To facilitate deployment and portability among different execution platforms we express our services model at two abstraction levels. Abstract services are used by the orchestration at an abstract level, and concrete services are the actual services implemented in a given technology. Abstraction level separation allows describing a service independently of its implementation technology. Java interfaces are used as formalism to define abstract services. These interfaces use as types of their method parameters simple Java types or the types defined in the Data domain presented above. In Fig. 3 a possible service model for alarm system is presented.

```
public interface TemperatureService {
   public Temperature getTemperature();
   public Average average(Temperature[] temperatures);
   public void save(Average average);
   public void doAction(Action action);
}
```

Fig. 3 Abstract Service Definition

2.2. Orchestration Core Model Composition

We present now how domain models are composed to build an orchestration model.

2.2.1. Composing Control and Data Models

Composing control and data, in our case, means associating instances of the *ProductType* concept defined in the control model with instances of the *DataType* defined in the data model. By example the *Measure* product type defined in the control model corresponds to the *Temperature* type defined in data model. The Fig. 4 presents our generic model composition tool (Codele) used in this case to create associations between control and data models.

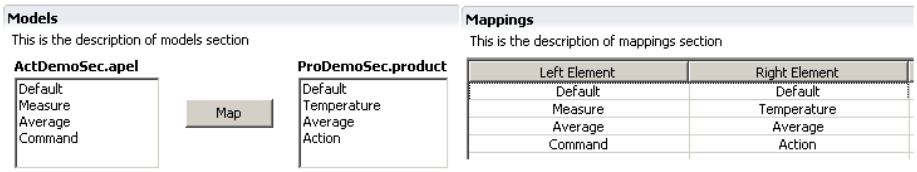

Fig. 4 Composing control and data (Codele)

2.2.2. Composing Control and Service Models

Composing control and services consists in establishing a relationship between an activity in control model and an abstract service operation in the service model. In our alarm example we establish the following links: *Acquisition-getTemperature*, *Analysis-average*, *Storage-save*, and *Processing-doAction*.

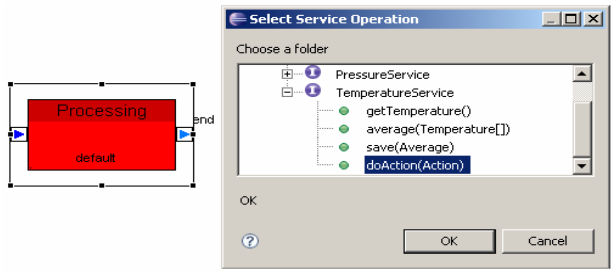

Fig. 5 Activity-Abstract Service Link

In Fig. 5, selecting the *Processing* displays the abstract services whose signature is compatible with the data "owned" by the activity. Most often, only one is possible.

3. Domain Composition : Links and Metalinks

The composition presented above is powerful enough to create an application using service orchestration technology. But other concerns corresponding to specific application requirements are occurring in specific business domains. These concerns must be composed with those presented above to meet the application requirements.

Composing two domains requires specifying relationships between concepts present in their metamodels, called metalinks, and relationship between elements pertaining to both models, called links. Metalinks describe the semantic relation between concepts of the two domains.

In a domain, the metamodel describes the abstract syntax in which models are expressed. Following most meta-environment strategy, at execution time, the model is transformed (reified) into instances of the abstract syntax classes. The metamodel also describes the domain operational semantics through the abstract syntax class, plus some other classes. These classes constitute an interpreter of the models once reified.

A strong requirement of our approach is that the domains to be composed must not be modified at all, i.e. models and interpreters must be kept unchanged. In our approach [9][10], the composition is performed defining metalinks between metamodel concepts, along with the code that specifies the semantics of the metalink. But if a metalink is defined between concept X and Y, then links must be defined between some instances x of X and some instances y of Y, i.e. links must be defined between model elements. Fig. 6 shows how two domains can be composed.

Models and links are reified thus it is possible, at execution time, to navigate the links and the models, and to create/modify/delete links and model instances, allowing meta-level introspection and complete reflection (i.e. dynamic model evolution).

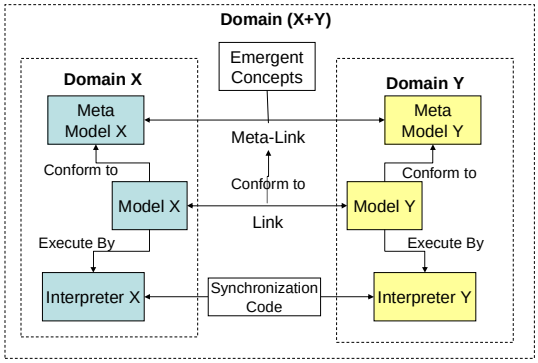

Fig. 6 Domain Composition Schema

For example, Fig. 4 shows how the generic model composition tool (Codele) has been customized to compose Control and Data models. Customization is possible because a metalink has been defined between the *ProductType* concept in the Control metamodel and the *DataType* concept in the Data metamodel. In the Control metamodel a *ProductType* has only a symbolic name, nothing is known at about the nature of products that are flowing between activities. Conversely, in the Data metamodel we define the structure of entities, but nothing is known about the actions performed on them. In this example, the metalink expresses a specialization relation allowing making precise what is really circulating between activities.

At model level, links between instances of two concepts must be defined. The composition defines that a *Measure* (in control model) will be a *Temperature* (in Data model), in the same way links are created between *Average-Average* and *Command-Action*, in Fig. 4 model composition for the alarm system is presented. Model composition is realized without modification of the original models which significantly increases reuse possibilities. One can imagine using the same control model applied on another Data model, by example, to use *Pressure* instead of *Temperature* type.

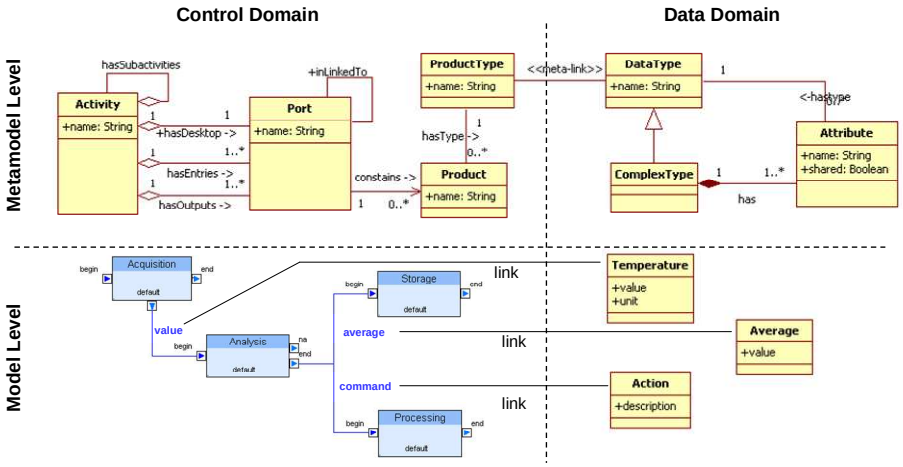

Fig. 7 Control and Data Domain Composition

Interpreters have also been composed to provide execution capacity to composed models. Interpreters are composed based in metalinks definitions, all public methods on related concepts are intercepted, and when methods are invoked, actions in the opposite domain can be performed. Additional code must be written to define the composition semantics as is presented in Fig. 6. In the particular case of *Control* and *Data* domains, when a *Product* is created in the control interpreter, a corresponding *Data* is created in the data interpreter. Fig. 7 summarizes how the control and data domains have been composed, and how models of Alarm application are composed.

3.1. Workflow Example

We have illustrated how domains can be composed with the *Control* and *Data* domains; indeed, it is how the core orchestration domain has been developed. But the same technique can be applied to compose the core orchestration domain with any other domain of interest, to fit specific business needs.

We have defined a workflow domain as the composition of the core orchestration domain, with document and resource domains. The reason is that the data circulating in a traditional workflow or an office automation system are documents (files). The document domain is in charge of managing and copying files, composition is defined by a metalink between *DataType* and *Document* concepts. In a traditional workflow, most activities are performed by humans. The resource domain expresses how resources (company structure, divisions, humans), are organized, the corresponding instances being stored in an LDAP repository for example. Hence this relation is defined by a metalink between *Activity* and *Role* concepts.

Fig. 8 shows how we have defined and implemented our process support system. It is important to notice that this composition process is structured, and fully reuses the previous orchestration domain composition.

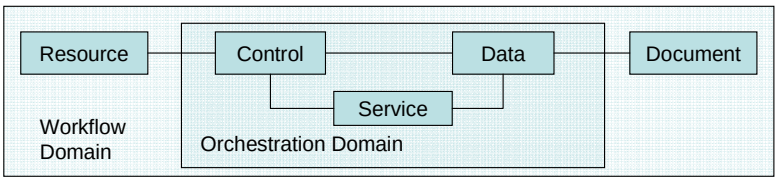

Fig. 8 Workflow Domain

4. Abstract Domain Composition: Annotations

The above composition technique can only be applied on domains in which it is possible to define an interpreter, and in which can be defined models, independent from the other domain models. Unfortunately, many domains, and most notably the non functional domains, do not satisfy this requirement. For example, we can define a security domain which metamodel contains concepts like authentication, confidentiality, integrity, etc. But it is not possible to define a security model independently from the associated application: confidentiality makes sense only when applied to a particular data for example. A domain in which independent models cannot be created is said to be abstract; it can be “instantiated” only when applied to a concrete domain that “inherits” the abstract domain characteristics.

The composition of an abstract domain with a concrete domain is based on an annotation technique, the composition schema is presented in Fig. 9. First, metalinks express the relationship between both domains, in the form of syntactic restrictions used to constrain annotation declarations. These restrictions indicate which concepts of X abstract domain can be applied on which concepts of Y concrete domain.

At model-level, elements of a concrete model of Y domain are annotated with concepts of the X domain metamodel respecting the restrictions imposed in the metalinks definition. Annotations can have attributes indicating how the annotation will be processed by the code generator. Annotation tools use metalinks definition to permits only correct annotations on concrete domain models.

The composition is created at an abstract level, hence several code generators can be developed to adapt orchestration to different technical constrains. Finally the code generator, receiving as input the model and its annotations, and having knowledge of X and Y domain metamodels and of Y interpreter implementation conventions, generates code which is weaved with the interpreter code to produce a customized execution of composite X+Y domain.

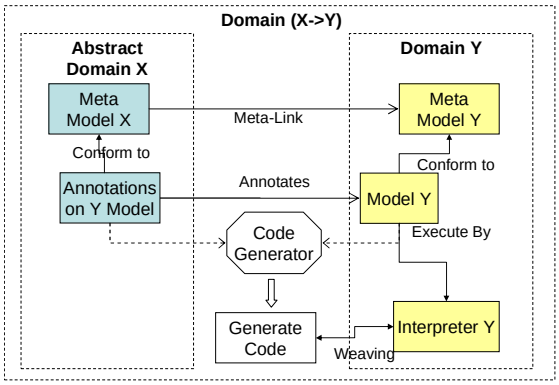

Fig. 9 Abstract Domain Composition Schema

There are two open issues in abstract domain composition approach: (1) validation of the annotations semantic, and (2) code generation ordering. Former issue is about the correctness of a set of annotations, metalinks only give a syntactical validation but nothing is said about the validity of overall annotations on model. Validation can be implemented at two levels in our approach, at edition time when annotations are declared or at generation time..

Order of code weaving is an existing problem in AOD approaches, the order in which aspects are weaved with the base code can change the semantics of the resultant code. In the same way the order of generation code of different concerns in our approach can change (or eliminate) the execution semantic of previous abstract domain compositions.

In comparison with AOP, our approach does not have an open joint point model, only state changes in Activities can be used as joint points. Moreover the additional behavior i.e. advices is embedded into the code generator in the form of code templates, and pointcuts are defined by quantification i.e. using the annotations declarations. Finally the weaving process is realized in generation phase. At technical level our approach uses AOP, but at user level, domain composition is performed, and AOP technology is not visible at all.

4.1. Security Example

Let us illustrate our annotation technique showing how the security concern was composed with the base orchestration domain. In our alarm system example, one can imagine that the alarm application is executed in a plant floor. Three security requirements are necessary: data circulating in the system must be confidential, data must not be modified by third parties, and more important, malicious actions performed on plant equipments must be avoided.

The identified concepts in security abstract domain are: authentication, meaning that a service invocation must be performed by a trusted resource; confidentiality, to indicate that the data exchanged must not read; and integrity to indicate that the data exchanged between services must no be modified by third parties.

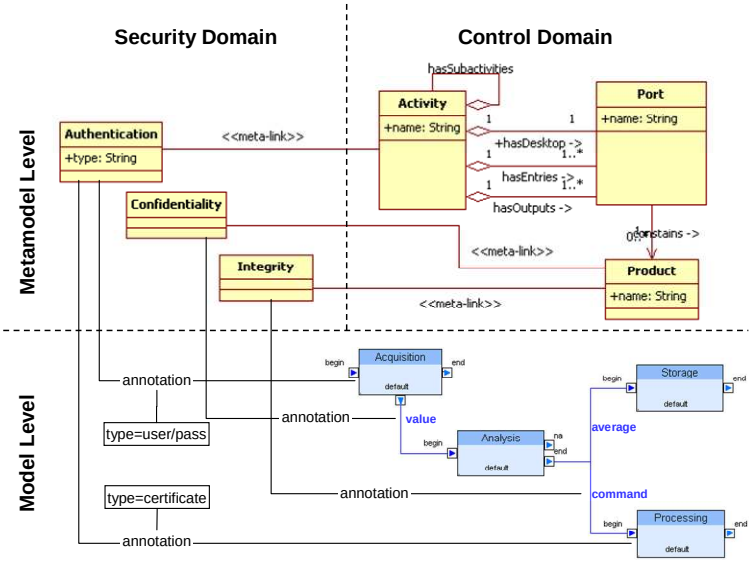

Fig. 10 Security and Control: Abstract Domain Composition

Metalinks specify the following syntactical restrictions: *Authentication* annotation can only be declared on *Activities* while *Confidentiality* and *Integrity* concepts can only annotate *products* when they flow between activities. Fig. 11 shows how the security metamodel and metalinks definitions have been used to partially generate an extension over the control model editor. This extension permits only to annotate concepts with appropriate annotations.

In our alarm example, *Acquisition* and *Processing* activities are both annotated with *Authentication*, but with different values for the type attribute, *value* product is annotated to be confidential and *command* product to maintain integrity. In contrast with domain composition, in abstract domain composition, abstract models do not exist per se, instead abstract metamodel concepts annotate model level elements of the concrete metamodel. In our security example, security metamodel concepts annotate the orchestration model as presented in Fig. 10.

In our security/orchestration example, for user convenience, only the control model is annotated. This is possible due to the following domain composition properties:

- Asymmetric nature of orchestration composition where control is considered as a central domain.

- Introspection property of domain composition which allows the code generator as well as the generated code to navigate into composed orchestration models.

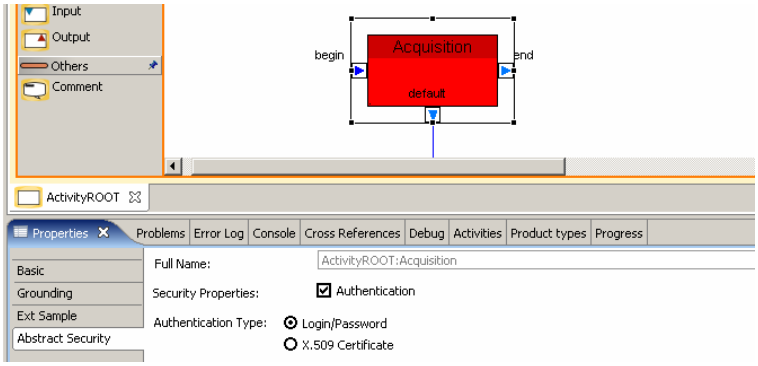

Fig. 11 Security Annotation Extension: Alarm System Security Properties

We have developed a code generator to guarantee security properties in web services orchestration. It first validates annotations and then generates code using the Apache WSS4J library which is an implementation of WS-Security standard. The code generator processes the control model annotations and changes the Web service invocation semantic accordingly. If an activity is annotated with *Authentication*, an authentication header is added to SOAP messages in the invocation of associated service, in the same way data encryption is used when products are annotated as *confidential* or to maintain integrity.

The same generator is used whatever the models in the orchestration domain. The generator also extends the model elements definition with attributes such that the annotation information will be found, at execution, on the reified model elements, extending introspection to the abstract domain composition. However, due to the generative approach, reflection is not possible; removing authentication at run-time for example is not possible.

5. Evaluation

Our approach has been validated first with the construction of FOCAS, which is the extensible service orchestration framework designed to support our approach, and second with the development of real applications in collaboration with our industrial partners.

FOCAS environment is built on top of Eclipse IDE and EMF. The metamodel of each concrete core domain (control, data and services) is defined by an Ecore model, and the associated editor is either hand coded (APEL) or generated by EMF framework. FOCAS has a high level view of domains and their relations, using its interface, developers hardly ever have to deal directly with Eclipse artifacts.

We realized very soon that our domain composition technology is too complex to be undertaken manually even by domains experts. A primary goal of FOCAS was to fully support domain composition, hiding its underlying complexity (use of Aspect J, metamodel manipulation, relationship semantics, etc). With FOCAS, adding a new concrete domain (concern) to the environment consists in specifying the metalinks between the new domain and the core metamodel using Codele metalinks editor, and in providing (in Java) their semantics. Adding a new abstract domain consists in specifying the metalinks between the new domain and the control metamodel, from which an extension of the control editor is generated (see Fig. 11). Then, the code generator (optionally the annotation validator) has to be written, it must implement some FOCAS interfaces and use the JET technology to be fully integrated into the environment.

It is much simpler to develop a service based application satisfying different concerns using model composition than in the traditional way; nevertheless, composing model by hand is not so easy. The other goal of FOCAS is to provide a framework which fully hides the many artifacts, code, and activities required by such applications. FOCAS automates the model composition task (see Fig. 4, Fig. 5), and generates most (and often all) the code required to make the application executable.

We have used our tool and approach to build real applications in association with some of our industrial partners. The process support system presented in section 3.1, which is a domain composition that has been used by years by different partners.

The alarm system presented along this paper is a (simplified) implementation of real system for a plant floor production that we have developed for our industrial partner Thales. In that application, we used DPWS and security properties implemented using WSS4J framework; but our new code generator use the Rampart library (security framework in Axis 2).

Other abstract domains are under way most notably the distribution (choreography), dynamic and deployment abstract domains. All these experiences have shown that FOCAS dramatically simplifies service based application design and development.

6. Related work

Domain composition has been addressed in two major ways: model and metamodel merging on the one side and models/metamodels synchronization and weaving on the other side. The first camp focuses on the semantics of operators like match, merge, weave applied to two models, and several platforms have been developed, like AMMA [19], Rondo [20], EOL [21], MOMENT [22].

Metamodeling environments have been developed to support model composition. GME (Generic Modeling Environment) builds new metamodels using union and inheritance, while XMF (eXecutable Metamodeling Facility) uses a specific language XSync for model synchronization.

All these approaches produce new models and metamodels which most often they do not support domain execution, and that impose to rebuild all the models and the associated tools. A company asset is made (among other things) of the models developed and tuned along the years, and of the people expertise in using the domain languages. For that reason we strongly emphasize domain composition that fully respects the models and the tools developed for a given domain.

In [4], composition is split into a core process and a set of business rules. Orchestration models are described in BPEL4WS language and business rules in a rules language. Composition of rules can be implemented of two ways, the first uses an aspect oriented variation of BPEL4WS or using a business rules engine synchronized with the execution of process engine. The principal advantage presented in [4] is the capability to reuse the business rules and a certain degree of flexibility because rules can evolve independently of the orchestration model. In our approach we consider business rules as particular case of composition with abstract domain, i.e., annotations.

In [4] and [5] a Web service composition is split into a core process and a set of rules. Business rules express company policies, which provides some flexibility because rules can evolve independently of the orchestration core model. In [5] rules are used to guide the execution environment in binding, re-binding and self-reconfiguring services when their properties do not match the environment requirements or when execution context changes. Decoupling between orchestration specification and rules allows for separation of concerns. In [4] and [5] composition of arbitrary domains is not addressed.

In [17] at high level tool is built to add security policies to services orchestration. A GUI is used with a set of patterns to help users in defining policies, which are then translated into WS-SecurityPolicy language. This approach uses a high abstraction level to specify security concerns in services composition, but does not enforce these properties at runtime, which contrasts with our approach where specification and runtime are both addressed, and for a wide range of concerns.

In BPEL4J [3] Java code snippets are inserted into orchestration specified in BPEL. BPEL4J gives BPEL complete calculus capability, but the Java code is tangled into the process definition and makes it orchestration models difficult to understand, concerns are coded in ad-hoc way into each orchestration. BPEL4People in [11] adds process interaction with human beings, adding a composition between orchestration and resource domains. These solutions require language extensions such as introducing new activity types, which is difficult to implement and is not supported by different implementations.

7. Conclusion

The fundamental concept of separation of concern is widely accepted, and indeed used in a number of engineering disciples. In software engineering, it is well known that no single decomposition can afford for modularizing correctly all the relevant concerns. Aspect Oriented Programming (AOP) is a low level technique for separation of concern implementation in which arbitrary concerns (aspects) can be composed (weaved) with a central reference program.

Our goal is to modularize each concern inside a domain, and to compose an arbitrary number of domains at a conceptual level, establishing metalinks between concepts and links between models. In this work we are focusing on services orchestration applications. For that reason, the core FOCAS platform is an orchestration system based on the predefined composition of three domains (control, service and data). The composition technology we propose makes that any other concern (domain) can be composed with the core orchestration, extending the FOCAS platform to the support of "any" business of technological concern.

This method does not work for domains in which models cannot be defined in isolation, generally the non-functional domains. These domains, called abstract domains, require a new composition technique called annotation. In FOCAS, the orchestration model can be simply annotated with concepts coming from the abstract domain metamodel, and the corresponding code is generated, conferring new properties to the orchestration application. The approach allows including a number of non-functional concerns into an existing service orchestration application. In contrast with other approaches, both domain composition, and abstract domain composition are fully respectful of the base domain, and require no change at all of the existing tools domains and models.

The FOCAS system has shown not only the feasibility of the approach, but through real experiments, it has shown that concern composition can be undertaken at a high conceptual level and that it is a viable and promising approach.

Future work includes extending the validation, developing other concrete and abstract domains, and developing service based application in various contexts. More fundamentally, we will try to identify which class of non-functional properties cannot be supported by our approach, and to address in this context, the difficult issue of concern interactions.

We believe that this new technology is a step forward in the application of separation of concern approach to process driven application, and software engineering in general.

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, H.: Web Services - Concepts, Architectures and Applications. Springer Verlag (2003)
2. Arkin, A.: Business Process Modeling Language. Intalio, Specification available at http://www.bpmi.org/ (2002)

3. Blow, M., Goland, Y., Kloppmann, M., Leymann, F., Pfau, G., Roller, D., Rowley, M.: BPEL4J: BPEL for java. A Joint White Paper by BEA and IBM (2004)
4. Charfi, A.., Mezini, M.: Hybrid Web Service Composition: Business Processes Meet Business Rules. ICSOC'04: Proceedings of the 2nd International Conference on Service Oriented Computing, New York, ACM Press (2004) 30–38
5. Colombo, M., Nitto, E. D., Mauri, M.: SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules. ICSOC'06: Proceedings of the 4th International Conference on Service Oriented Computing, Chicago, Springer Berlin (2006) 191–202
6. Cubera, F. et al.: Web Services Business Process Execution Language. Specification available at http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf (2007)
7. Della-Libera, G.et al: Web Services Security Policy Language. Specification available at http://specs.xmlsoap.org/ws/2005/07/securitypolicy/ws-securitypolicy.pdf (2003)
8. Estublier, J., Dami, S., Amiour, M.: APEL: A graphical yet executable formalism for process modeling. Automated Software Engineering: An International Journal **5**(1) (1998) 61–96
9. Estublier, J., Ionita, A.D., Vega, G.: A Domain Composition Approach, Las Vegas, International Workshop on Applications of UML/MDA to Software Systems, CSREA (2005) 1–7
10. Estublier, J., Villalobos, J., Tuyet LE, A., Sanlaville, S., Vega, G.: An Approach and Framework for Extensible Process Support System. Lecture Notes in Computer Science **2786**(2003) 46–61
11. Kloppmann, M., Koening, D., Leymann, F., Pfau, G., Rickayzen, A., von Riegen, C., Schmidt, P., Trickovic, I.: WS-BPEL Extension for People – BPEL4People. A Joint White Paper by IBM and SAP (2005)
12. Leymann, F.: Web Service Flow Language (WSFL 1.0). IBM, Specification available at http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf (2001)
13. Nadalin, A. et al: Web Services Security: SOAP Message Security 1.1. Specification available at http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf (2006)
14. OSGi Alliance: OSGi 4.0 release. Specification available at http://www.osgi.org/ (2005)
15. Ossher, H., Tarr, P.: Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In: Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development (2000).
16. Papazoglou, M.P., Heuvel, W.: Service oriented architectures: approaches, technologies and research issues. The VLDB Journal **16**(3) (2007) 389–415
17. Tatsubori, M., Imamura, T., Nakamura, Y.: Best-Practice Patterns and Tool Support for Configuring Secure Web Services Messaging. ICWS'04: International Conference on Web Services, San Diego, IEEE Computer Society(2004) 244–251
18. Thatte, S.: XLANG: Web Services for Business Process Design. Microsoft, Specification available at http://www.gotdotnet.com/team/xml_wsspecs/xlangc/default.htm (2001)
19. Didonet Del Fabro, M. and F. Jouault. *Model Transformation and Weaving in the AMMA Platform*. in *Workshop on Generative and Transformational Techniques in Software Engineering (GTTSE)*. 2005. Braga, Portugal.
20. Melnik, S., E. Rahm, and P.A. Bernstein. *Rondo: A Programming Platform for Generic Model Management*. in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. 2003.
21. Kolovos, D.S., R.F. Paige, and F.A.C. Polack. *Eclipse Development Tools for Epsilon*. in *Eclipse Summit Europe, Eclipse Modeling Symposium*. October 2006. Esslingen, Germany.
22. Boronat, A., J.A. Carsi, and I. Ramos. *Automatic Support for Traceability in a Generic Model Management Framework*. in *European Conference on Model Driven Architecture - Foundations and Applications*. November 2005. Nuremberg (Germany).